1. Course Information

Staff

Faculty

Butler Lampson	32-G924	425-703-5925	blampson@microsoft.com
Daniel Jackson	32-G704	8-8471	dnj@mit.edu

Teaching Assistant

David Shin			dshin@mit.edu

Course Secretary

Maria Rebelo	32-G715	3-5895	mr@csail.mit.edu

Office Hours

Messrs. Lampson and Jackson will arrange individual appointments. David Shin will hold scheduled office hours in the 7th floor lounge of the Gates tower in building, Monday 4-6. In addition to holding regularly scheduled office hours, he will also be available by appointment.

Lectures and handouts

Lectures are held on Monday and Wednesday from 1:00 to 2:30PM in room 32-155. Messrs. Lampson and Jackson will split the lectures. The tentative schedule is at the end of this handout.

The source material for this course is an extensive set of handouts. There are about 400 pages of topic handouts that take the place of a textbook; you will need to study them to do well in the course. Since we don't want to simply repeat the written handouts in class, we will hand out the material for each lecture one week in advance. *We expect you to read the day's handouts before the class and come prepared to ask questions, discuss the material, or follow extensions of it or different ways of approaching the topic.*

Seven research papers supplement the topic handouts. In addition there are 5 problem sets, and the project described below. Solutions for each problem set will be available shortly after the due date.

There is a course Web page, at web.mit.edu/6.826/www. Last year's handouts can be found from this page. Current handouts will be placed on the Web as they are produced.

Current handouts will generally be available in lecture. If you miss any in lecture, you can obtain them afterwards from the course secretary. She keeps them in a file cabinet outside her office.

Problem sets

There is a problem set approximately once a week for the first half of the course. Problem sets are handed out on Wednesdays and are due by noon the following Wednesday in the tray on the

course secretary's desk. They normally cover the material discussed in class during the week they are handed out. Delayed submission of the solutions will be penalized, and no solutions will be accepted after Thursday 5:00PM.

Students in the class will be asked to help grade the problem sets. Each week a team of students will work with the TA to grade the week's problems. This takes about 3-4 hours. Each student will probably only have to do it once during the term.

We will try to return the graded problem sets, with solutions, within a week after their due date.

Policy on collaboration

We encourage discussion of the issues in the lectures, readings, and problem sets. However, if you collaborate on problem sets, you must tell us who your collaborators are. And in any case, you must write up all solutions on your own.

Project

During the last half of the course there is a project in which students will work in groups of three or so to apply the methods of the course to their own research projects. Each group will pick a real system, preferably one that some member of the group is actually working on but possibly one from a published paper or from someone else's research, and write:

A specification for it.

High-level code that captures the novel or tricky aspects of the actual implementation.

The abstraction function and key invariants for the correctness of the code. This is not optional; if you can't write these things down, you don't understand what you are doing.

Depending on the difficulty of the specification and code, the group may also write a correctness proof for the code.

Projects may range in style from fairly formal, like handout 18 on consensus, in which the 'real system' is a simple one, to fairly informal (at least by the standards of this course), like the section on copying file systems in handout 7. These two handouts, along with the ones on naming, sequential transactions, concurrent transactions, and caching, are examples of the appropriate size and possible styles of a project.

The result of the project should be a write-up, in the style of one of these handouts. During the last two weeks of the course, each group will give a 25-minute presentation of its results. We have allocated four class periods for these presentations, which means that there will be twelve or fewer groups.

The projects will have five milestones. The purpose of these milestones is not to assign grades, but to make it possible for the instructors to keep track of how the projects are going and give everyone the best possible chance of a successful project

1. We will form the groups around March 2, to give most of the people that will drop the course a chance to do so.
2. Each group will write up a 2-3 page project proposal, present it to one of the instructors around spring break, and get feedback about how appropriate the project is and suggestions

on how to carry it out. Any project that seems to be seriously off the rails will have a second proposal meeting a week later.

3. Each group will submit a 5-10 page interim report in the middle of the project period.
4. Each group will give a presentation to the class during the last two weeks of classes.
5. Each group will submit a final report, which is due on Friday, May 14, the last day allowed by MIT regulations. Of course you are free to submit it early.

Half the groups will be 'early' ones; the other half will be 'late' ones that give their presentations one week later. The due dates of proposals and interim reports will be spread out over two weeks in the same way. See the schedule later in this handout for precise dates.

Grades

There are no exams. Grades are based 30% on the problem sets, 50% on the project, and 20% on class participation and quality and promptness of grading.

Course mailing list

A mailing list for course announcements—6.826-students@mit.edu—has been set up to include all students and the TA. If you do not receive any email from this mailing list within the first week, check with the TA. Another mailing list, 6.826-staff@mit.edu, sends email to the entire 6.826 staff.

Course Schedule

Date	No	By	HO	Topic	PS out	PS due
Wed., Feb. 8	1	L		**Overview**. The Spec language. State machine semantics. Examples of specifications and code.	1	
			1	Course information		
			2	Background		
			3	Introduction to Spec		
			4	Spec reference manual		
			5	Examples of specs and code		
Mon., Feb. 13	2	J		**Spec and code** for sequential programs. Correctness notions and proofs. Proof methods: abstraction functions and invariants.		
			6	Abstraction functions		
Wed., Feb. 15	3	L		**File systems 1**: Disks, simple sequential file system, caching, logs for crash recovery.	2	1
			7	Disks and file systems		
Tues., Feb. 21	4	L		**File systems 2**: Copying file system.		
Wed., Feb.22	5	L		**Proof methods**: History and prophecy variables; abstraction relations.	3	2
			8	History variables		
Mon., Feb. 27	6	S		**Semantics and proofs**: Formal sequential semantics of Spec.		
			9	Atomic semantics of Spec		
Wed., Mar. 1	7	J		**Naming**: Specs, variations, and examples of hierarchical naming.	Form groups	
			12	Naming		
			13	Paper: David Gifford et al, Semantic file systems, *Proc.13th ACM Symposium on Operating System Principles*, October 1991, pp 16-25.		
Mon., Mar. 6	8	L		**Performance**: How to get it, how to analyze it.	4	3
			10	Performance		
			11	Paper: Michael Schroeder and Michael Burrows, Performance of Firefly RPC, *ACM Transactions on Computer Systems* **8**, 1, February 1990, pp 1-17.		
Wed., Mar. 8	9	J		**Concurrency 1**: Practical concurrency, easy and hard. Easy concurrency using locks and condition variables. Problems with it: scheduling, deadlock.	5	4
			14	Practical concurrency		
			15	Concurrent disks		
			16	Paper: Andrew Birrell, An Introduction to Programming with C# Threads, Microsoft Research Technical Report MSR-TR-2005-68, May 2005.		

Date	No	By	HO	Topic	PS out	PS due
Mon., Mar. 13	10	S		**Concurrency 2:** Concurrency in Spec: threads and non-atomic semantics. Big atomic actions. Safety and liveness. Examples of concurrency.		
			17	Formal concurrency		
Wed., Mar. 15	11	S		**Concurrency 3:** Proving correctness of concurrent programs: assertional proofs, model checking		5
Mon., Mar. 20	12	L		**Distributed consensus 1**. Paxos algorithm for asynchronous consensus in the presence of faults.		
			18	Consensus		
Wed., Mar. 22	13	L		**Distributed consensus** 2.	Early proposals	
Mar. 27-31				Spring Break		
Mon., Apr. 3	14	J		**Sequential transactions** with caching.		
			19	Sequential transactions		
Wed., Apr. 5	15	J		**Concurrent transactions**: Specs for serializability. Ways to code the specs.	Late proposals	
			20	Concurrent transactions		
Mon., Apr. 10	16	J		**Distributed transactions**: Commit as a consensus problem. Two-phase commit. Optimizations.		
			27	Distributed transactions		
Wed., Apr. 12	17	L		**Introduction to distributed systems**: Characteristics of distributed systems. Physical, data link, and network layers. Design principles. **Networks 1**: Links. Point-to-point and broadcast networks.		
			21	Distributed systems		
			22	Paper: Michael Schroeder et al, Autonet: A high-speed, self-configuring local area network, *IEEE Journal on Selected Areas in Communications* **9**, 8, October 1991, pp 1318-1335.		
			23	Networks: Links and switches		
Mon., Apr. 17				Patriot's Day, no class		
Wed., Apr. 19	18	L		**Networks 2**: Links cont'd: Ethernet. Token Rings. Switches. Coding switches. Routing. Learning topologies and establishing routes.	Early interim reports	
Mon., Apr. 24	19	J		**Networks 3**: Network objects and remote procedure call (RPC).		
			24	Network objects		

Date	No	By	HO	Topic	PS out	PS due
			25	Paper: Andrew Birrell et al., Network objects, *Proc.14th ACM Symposium on Operating Systems Principles*, Asheville, NC, December 1993.		
Wed., Apr. 26	20	L		**Networks 4**: Reliable messages. 3-way handshake and clock code. TCP as a form of reliable messages.		Late interim reports
			26	Paper: Butler Lampson, Reliable messages and connection establishment. In *Distributed Systems*, ed. S. Mullender, Addison-Wesley, 1993, pp 251-281.		
Mon., May 1	21	J		**Replication and availability**: Coding replicated state machines using consensus. Applications to replicated storage.		
			28	Replication		
			29	Paper: Jim Gray and Andreas Reuter, Fault tolerance, in *Transaction Processing: Concepts and Techniques*, Morgan Kaufmann, 1993, pp 93-156.		
Wed., May 3	22	J		**Caching**: Maintaining coherent memory. Broadcast (snoopy) and directory protocols. Examples: multiprocessors, distributed shared memory, distributed file systems.		
			30	Concurrent caching		
Mon., May 8	23			**Early project presentations**		
Wed., May 10	24			**Early project presentations**		
Mon., May 15	25			**Late project presentations**		
Wed., May 17	26			**Late project presentations**		
Fri., May 19				**Final reports due**		
May 22-26				Finals week. There is no final for 6.826.		

2. Overview and Background

This is a course for computer system designers and builders, and for people who want to really understand how systems work, especially concurrent, distributed, and fault-tolerant systems.

The course teaches you
how to write precise specifications for any kind of computer system,
what it means for code to satisfy a specification, and
how to prove that it does.
It also shows you how to use the same methods less formally, and gives you some suggestions for deciding how much formality is appropriate (less formality means less work, and often a more understandable spec, but also more chance to overlook an important detail).

The course also teaches you a lot about the topics in computer systems that we think are the most important: persistent storage, concurrency, naming, networks, distributed systems, transactions, fault tolerance, and caching. The emphasis is on
careful specifications of subtle and sometimes complicated things,
the important ideas behind good code, and
how to understand what makes them actually work.
We spend most of our time on specific topics, but we use the general techniques throughout. We emphasize the ideas that different kinds of computer system have in common, even when they have different names.

The course uses a formal language called Spec for writing specs and code; you can think of it as a very high level programming language. There is a good deal of written introductory material on Spec (explanations and finger exercises) as well as a reference manual and a formal semantics. We introduce Spec ideas in class as we use them, but we do not devote class time to teaching Spec per se; we expect you to learn it on your own from the handouts. The one to concentrate on is handout 3, which has an informal introduction to the main features and lots of examples. Section 9 of handout 4, the reference manual, should also be useful. The rest of the reference manual is for reference, not for learning. Don't overlook the one page summary at the end of handout 3.

Because we write specs and do proofs, you need to know something about logic. Since many people don't, there is a concise treatment of the logic you will need at the end of this handout.

This is not a course in computer architecture, networks, operating systems, or databases. We will not talk in detail about how to code pipelines, memory interconnects, multiprocessors, routers, data link protocols, network management, virtual memory, scheduling, resource allocation, SQL, relational integrity, or TP monitors, although we will deal with many of the ideas that underlie these mechanisms.

Topics

General

Specifications as state machines.
The Spec language for describing state machines (writing specs and code).
What it means to implement a spec.
Using abstraction functions and invariants to prove that a program implements a spec.

What it means to have a crash.
What every system builder needs to know about performance.

Specific

Disks and file systems.
Practical concurrency using mutexes (locks) and condition variables; deadlock.
Hard concurrency (without locking): models, specs, proofs, and examples.
Transactions: simple, cached, concurrent, distributed.
Naming: principles, specs, and examples.
Distributed systems: communication, fault-tolerance, and autonomy.
Networking: links, switches, reliable messages and connections.
Remote procedure call and network objects.
Fault-tolerance, availability, consensus and replication.
Caching and distributed shared memory.

Previous editions of the course have also covered security (authentication, authorization, encryption, trust) and system management, but this year we are omitting these topics in order to spend more time on concurrency and semantics and to leave room for project presentations.

Prerequisites

There are no formal prerequisites for the course. However, we assume some knowledge both of computer systems and of mathematics. If you have taken 6.033 and 6.042, you should be in good shape. If you are missing some of this knowledge you can pick it up as we go, but if you are missing a lot of it you can expect to have serious trouble. It's also important to have a certain amount of maturity: enough experience with systems and mathematics to feel comfortable with the basic notions and to have some reliable intuition.

If you know the meaning of the following words, you have the necessary background. If a lot of them are unfamiliar, this course is probably not for you.

Systems

Cache, virtual memory, page table, pipeline
Process, scheduler, address space, priority
Thread, mutual exclusion (locking), semaphore, producer-consumer, deadlock
Transaction, commit, availability, relational data base, query, join
File system, directory, path name, striping, RAID
LAN, switch, routing, connection, flow control, congestion
Capability, access control list, principal (subject)

If you have not already studied Lampson's paper on hints for system design, you should do so as background for this course. It is Butler Lampson, Hints for computer system design, *Proceedings of the Ninth ACM Symposium on Operating Systems Principles*, October 1983, pp 33-48. There is a pointer to it on the course Web page.

Programming

Invariant, precondition, weakest precondition, fixed point
Procedure, recursion, stack
Data type, sub-type, type-checking, abstraction, representation

Object, method, inheritance
Data structures: list, hash table, binary search, B-tree, graph

Mathematics

Function, relation, set, transitive closure
Logic: proof, induction, de Morgan's laws, implication, predicate, quantifier
Probability: independent events, sampling, Poisson distribution
State machine, context-free grammar
Computational complexity, unsolvable problem

If you haven't been exposed to formal logic, you should study the summary at the end of this handout.

References

These are places to look when you want more information about some topic covered or alluded to in the course, or when you want to follow current research. You might also wish to consult Prof. Saltzer's bibliography for 6.033, which you can find on the course web page.

Books

Some of these are fat books better suited for reference than for reading cover to cover, especially Cormen, Leiserson, and Rivest, Jain, Mullender, Hennessy and Patterson, and Gray and Reuter. But the last two are pretty easy to read in spite of their encyclopedic character.

Specification: Leslie Lamport, *Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers,* Addison-Wesley, 2002. TLA+ is superficially quite different from Spec, but the same underneath. Lamport's approach is somewhat more mathematical than ours, but in the same spirit. You can find this book at http://research.microsoft.com/users/lamport/tla/book.html.

Systems programming: Greg Nelson, ed., *Systems Programming with Modula-3*, Prentice-Hall, 1991. Describes the language, which has all the useful features of C++ but is much simpler and less error-prone, and also shows how to use it for concurrency (a version of chapter 4 is a handout in this course), an efficiently customizable I/O streams package, and a window system.

Performance: Jon Bentley, *Writing Efficient Programs*, Prentice-Hall, 1982. Short, concrete, and practical. Raj Jain, *The Art of Computer Systems Performance Analysis*, Wiley, 1991. Tells you much more than you need to know about this subject, but does have a lot of realistic examples.

Algorithms and data structures: Robert Sedgwick, *Algorithms*, Addison-Wesley, 1983. Short, and usually tells you what you need to know. Tom Cormen, Charles Leiserson, and Ron Rivest, *Introduction to Algorithms*, McGraw-Hill, 1989. Comprehensive, and sometimes valuable for that reason, but usually tells you a lot more than you need to know.

Distributed algorithms: Nancy Lynch, *Distributed Algorithms*, Morgan Kaufmann, 1996. The bible for distributed algorithms. Comprehensive, but a much more formal treatment than in this course. The topic is algorithms, not systems.

Computer architecture: John Hennessy and David Patterson, *Computer Architecture: A Quantitative Approach*, 2nd edition, Morgan Kaufmann, 1995. The bible for computer architecture.

The second edition has lots of interesting new material, especially on multiprocessor memory systems and interconnection networks. There's also a good appendix on computer arithmetic; it's useful to know where to find this information, though it has nothing to do with this course.

Transactions, data bases, and fault-tolerance: Jim Gray and Andreas Reuter, *Transaction Processing: Concepts and Techniques*, Morgan Kaufmann, 1993. The bible for transaction processing, with much good material on data bases as well; it includes a lot of practical information that doesn't appear elsewhere in the literature.

Networks: Radia Perlman, *Interconnections: Bridges and Routers*, Addison-Wesley, 1992. Not exactly the bible for networking, but tells you nearly everything you might want to know about how packets are actually switched in computer networks.

Distributed systems: Sape Mullender, ed., *Distributed Systems*, 2nd ed., Addison-Wesley, 1993. A compendium by many authors that covers the field fairly well. Some chapters are much more theoretical than this course. Chapters 10 and 11 are handouts in this course. Chapters 1, 2, 8, and 12 are also recommended. Chapters 16 and 17 are the best you can do to learn about real-time computing; unfortunately, that is not saying much.

User interfaces: Alan Cooper, *About Face*, IDG Books, 1995. Principles, lots of examples, and opinionated advice, much of it good, from the original designer of Visual Basic.

Journals

You can find all of these in the CSAIL reading room in 32-G882. The cryptic strings in brackets are call numbers there. You can also find the ACM publications in the ACM digital library at www.acm.org.

For the current literature, the best sources are the proceedings of the following conferences. 'Sig' is short for "Special Interest Group", a subdivision of the ACM that deals with one field of computing. The relevant ones for systems are SigArch for computer architecture, SigPlan for programming languages, SigOps for operating systems, SigComm for communications, SigMod for data bases, and SigMetrics for performance measurement and analysis.

Symposium on Operating Systems Principles (SOSP; published as special issues of ACM SigOps *Operating Systems Review*; fall of odd-numbered years) [P4.35.06]

Operating Systems Design and Implementation (OSDI; Usenix Association, now published as special issues of ACM *SigOps Review*; fall of even-numbered years, except spring 1999 instead of fall 1998) [P4.35.U71]

Architectural Support for Programming Languages and Operating Systems (ASPLOS; published as special issues of ACM SigOps *Operating Systems Review*, SigArch *Computer Architecture News*, or *SigPlan Notices*; fall of even-numbered years) [P6.29.A7]

Applications, Technologies, Architecture, and Protocols for Computer Communication, (SigComm conference; published as special issues of ACM SigComm *Computer Communication Review*; annual) [P6.24.D31]

Principles of Distributed Computing (PODC; ACM; annual) [P4.32.D57]

Very Large Data Bases (VLDB; Morgan Kaufmann; annual) [P4.33.V4]

International Symposium on Computer Architecture (ISCA; published as special issues of ACM SigArch *Computer Architecture News*; annual) [P6.20.C6]

Less up to date, but more selective, are the journals. Often papers in these journals are revised versions of papers from the conferences listed above.

ACM Transactions on Computer Systems

ACM Transactions on Database Systems

ACM Transactions on Programming Languages and Systems

There are often good survey articles in the less technical IEEE journals:

IEEE *Computer*, *Networks*, *Communication*, *Software*

The Internet Requests for Comments (RFC's) can be reached from

`http://www.cis.ohio-state.edu/hypertext/information/rfc.html`

Rudiments of logic

Propositional logic

The basic type is `Bool`, which contains two elements `true` and `false`. Expressions in these operators (and the other ones introduced later) are called 'propositions'.

Basic operators. These are ∧ (and), ∨ (or), and ~ (not).[1] The meaning of these operators can be conveniently given by a 'truth table' which lists the value of `a op b` for each possible combination of values of `a` and `b` (the operators on the right are discussed later) along with some popular names for certain expressions and their operands.

		negation	conjunction	disjunction	equality		implication
		not	and	or			implies
a	b	~a	a ∧ b	a ∨ b	a = b	a ≠ b	a ⇒ b
T	T	F	T	T	T	F	T
T	F		F	T	F	T	F
F	T	T	F	T	F	T	T
F	F		F	F	T	F	T
name of a			conjunct	disjunct			antecedent
name of b			conjunct	disjunct			consequent

Note: In Spec we write ==> instead of the ⇒ that mathematicians use for implication. Logicians write ⊃ for implication, which looks different but is shaped like the > part of ⇒.

Since the table has only four rows, there are only 16 Boolean operators, one for each possible arrangement of `T` and `F` in a column. Most of the ones not listed don't have common names, though 'not and' is called 'nand' and 'not or' is called 'nor' by logic designers.

The ∧ and ∨ operators are
commutative and
associative and
distribute over each other.
That is, they are just like * (times) and + (plus) on integers, except that + doesn't distribute over *:

```
a + (b * c) ≠ (a + b) * (a + c)
```

but ∨ does distribute over ∧:

```
a ∨ (b ∧ c) = (a ∨ b) ∧ (a ∨ c)
```

An operator that distributes over ∧ is called 'conjunctive'; one that distributes over ∨ is called 'disjunctive'. Both ∧ and ∨ are both conjunctive and disjunctive. This takes some getting used to.

The relation between these operators and ~ is given by DeMorgan's laws (sometimes called the "bubble rule" by logic designers), which say that you can push ~ inside ∧ or ∨ (or pull it out) by flipping from one to the other:

```
~ (a ∧ b)  =  ~a ∨ ~b
~ (a ∨ b)  =  ~a ∧ ~b
```

[1] It's possible to write all three in terms of the single operator 'nor' or 'nand', but our goal is clarity, not minimality.

To put a complex expression into " disjunctive normal form" replace terms in = and ⇒ with their equivalents in ∧, ∨, and ~ (given below), use DeMorgan's laws to push all the ~'s in past ∧ and ∨ so that they apply to variables, and then distribute ∧ over ∨ so that the result looks like

`(a₁ ∧ ~a₂ ∧ ...) ∨ (~b₁ ∧ b₂ ∧ ...) ∨ ...`

The disjunctive normal form is unique (up to ordering, since ∧ and ∨ are commutative). Of course, you can also distribute ∨ over ∧ to get a unique "conjunctive normal form".

If you want to find out whether two expressions are equal, one way is to put them both into disjunctive (or conjunctive) normal form, sort the terms, and see whether they are identical. Another way is to list all the possible values of the variables (if there are *n* variables, there are 2^n of them) and tabulate the values of the expressions for each of them; we saw this 'truth table' for some two-variable expressions above.

Because `Bool` is the result type of relations like =, you can write expressions that mix up relations with other operators in ways that are impossible for any other type. Notably

`(a = b) = ((a ∧ b) ∨ (~a ∧ ~b))`

Some people feel that the outer = in this expression is somehow different from the inner one, and write it ≡. Experience suggests, however, that this is often a harmful distinction to make.

Implication. We can define an ordering on `Bool` with `false > true`, that is, `false` is greater than `true`. The non-strict version of this ordering is called 'implication' and written ⇒ (rather than ≥ or >= as we do with other types; logicians write it ⊃, which also looks like an ordering symbol). So `(true ⇒ false) = false` (read this as: "`true` is greater than or equal to `false`" is false) but all other combinations are `true`. The expression `a ⇒ b` is pronounced "`a` implies `b`", or "if `a` then `b`".[2]

There are lots of rules for manipulating expressions containing ⇒; the most useful ones are given below. If you remember that ⇒ is an ordering you'll find it easy to remember most of the rules, but if you forget the rules or get confused, you can turn the ⇒ into ∨ by the rule

`(a ⇒ b) = ~a ∨ b`

and then just use the simpler rules for ∧, ∨, and ~. So remember this even if you forget everything else.

The point of implication is that it tells you when one proposition is stronger than another, in the sense that if the first one is true, the second is also true (because if both `a` and `a ⇒ b` are `true`, then `b` must be `true` since it can't be `false`).[3] So we use implication all the time when reasoning from premises to conclusions. Two more ways to pronounce `a ⇒ b` are "`a` is stronger than `b`" and "`b` follows from `a`". The second pronunciation suggests that it's sometimes useful to write the operands in the other order, as `b ⇐ a`, which can also be pronounced "`b` is weaker than `a`" or "`b` only if `a`"; this should be no surprise, since we do it with other orderings.

[2] It sometimes seems odd that `false` implies `b` regardless of what `b` is, but the "if ... then" form makes it clearer what is going on: if `false` is `true` you can conclude anything, but of course it isn't. A proposition that implies `false` is called 'inconsistent' because it implies anything. Obviously it's bad to think that an inconsistent proposition is true. The most likely way to get into this hole is to think that each of a collection of innocent looking propositions is true when their conjunction turns out to be inconsistent.

[3] It may also seem odd that `false > true` rather than the other way around, since `true` seems better and so should be bigger. But in fact if we want to conclude lots of things, being close to `false` is better because if `false` is true we can conclude anything, but knowing that `true` is true doesn't help at all. Strong propositions are as close to `false` as possible; this is logical brinkmanship. For example, `a ∧ b` is closer to `false` than `a` (there are more values of the variables `a` and `b` that make it `false`), and clearly we can conclude more things from it than from `a` alone.

Of course, implication has the properties we expect of an ordering:

Transitive: If `a ⇒ b` and `b ⇒ c` then `a ⇒ c`.[4]

Reflexive: `a ⇒ a`.

Anti-symmetric: If `a ⇒ b` and `b ⇒ a` then `a = b`.[5]

Furthermore, ~ reverses the sense of implication (this is called the 'contrapositive'):

`(a ⇒ b) = (~b ⇒ ~a)`

More generally, you can move a disjunct on the right to a conjunct on the left by negating it, or vice versa. Thus

`(a ⇒ b ∨ c) = (a ∧ ~b ⇒ c)`

As special cases in addition to the contrapositive we have

`(a ⇒ b) = (a ∧ ~b ⇒ false ) = ~ (a ∧ ~b) ∨ false = ~a ∨ b`

`(a ⇒ b) = (true ⇒ ~a ∨ b) = false ∨ ~a ∨ b = ~a ∨ b`

since `false` and `true` are the identities for ∨ and ∧.

We say that an operator `op` is 'monotonic' in an operand if replacing that operand with a stronger (or weaker) one makes the result stronger (or weaker). Precisely, "`op` is monotonic in its first operand" means that if `a ⇒ b` then `(a op c) ⇒ (b op c)`. Both ∧ and ∨ are monotonic; in fact, any operator that is conjunctive (distributes over ∧) is monotonic, because if `a ⇒ b` then `a = (a ∧ b)`, so

`a op c = (a ∧ b) op c = a op c ∧ b op c ⇒ b op c`

If you know what a lattice is, you will find it useful to know that the set of propositions forms a lattice with ⇒ as its ordering and (remember, think of ⇒ as "greater than or equal"):

top	= `false`		
bottom	= `true`		
meet	= ∧	least upper bound,	so `(a ∧ b) ⇒ a` and `(a ∧ b) ⇒ b`
join	= ∨	greatest lower bound,	so `a ⇒ (a ∨ b)` and `b ⇒ (a ∨ b)`

This suggests two more expressions that are equivalent to `a ⇒ b`:

`(a ⇒ b) = (a = (a ∧ b))`	'and'ing a weaker term makes no difference, because `a ⇒ b` iff `a =` least upper bound(`a`, `b`).
`(a ⇒ b) = (b = (a ∨ b))`	'or'ing a stronger term makes no difference, because `a ⇒ b` iff `b =` greatest lower bound(`a`, `b`).

Predicate logic

Propositions that have free variables, like `x < 3` or `x < 3 ⇒ x < 5`, demand a little more machinery. You can turn such a proposition into one without a free variable by substituting some value for the variable. Thus if `P(x)` is `x < 3` then `P(5)` is `5 < 3 = false`. To get rid of the free variable without substituting a value for it, you can take the 'and' or 'or' of the proposition for all the possible values of the free variable. These have special names and notation[6]:

`∀ x | P(x) = P(x₁) ∧ P(x₂) ∧ ...`	for all `x`, `P(x)`. In Spec, `(ALL x | P(x))` or `∧ : {x | P(x)}`

[4] We can also write this `((a ⇒ b) ∧ (b ⇒ c)) ⇒ (a ⇒ c)`.

[5] Thus `(a = b) = (a ⇒ b ∧ b ⇒ a)`, which is why `a = b` is sometimes pronounced "`a` if and only if `b`" and written "`a` iff `b`".

[6] There is no agreement on what symbol should separate the `∀ x` or `∃ x` from the `P(x)`. We use '|' here as Spec does, but other people use '.' or ':' or just a space, or write `(∀ x)` and `(∃ x)`. Logicians traditionally write `(x)` and `(∃ x)`.

`∃ x | P(x) = P(x1) ∨ P(x2) ∨ ...` there exists an `x` such that `P(x)`. In Spec, `(EXISTS x | P(x))` or `∨ : {x | P(x)}`

Here the x_i range over all the possible values of the free variables.[7] The first is called 'universal quantification'; as you can see, it corresponds to conjunction. The second is called 'existential quantification' and corresponds to disjunction. If you remember this you can easily figure out what the quantifiers do with respect to the other operators.

In particular, DeMorgan's laws generalize to quantifiers:

```
~ (∀ x | P(x)) = (∃ x | ~P(x))
~ (∃ x | P(x)) = (∀ x | ~P(x))
```

Also, because ∧ and ∨ are conjunctive and therefore monotonic, ∀ and ∃ are conjunctive and therefore monotonic.

It is not true that you can reverse the order of ∀ and ∃, but it's sometimes useful to know that having ∃ first is stronger:

```
∃ y | ∀ x | P(x, y) ⇒ ∀ x | ∃ y | P(x, y)
```

Intuitively this is clear: a `y` that works for every `x` can surely do the job for each particular `x`.

If we think of `P` as a relation, the consequent in this formula says that `P` is total (relates every `x` to some `y`). It doesn't tell us anything about how to find a `y` that is related to `x`. As computer scientists, we like to be able to compute things, so we prefer to have a function that computes `y`, or the set of `y`'s, from `x`. This is called a 'Skolem function'; in Spec you write `P.func` (or `P.setF` for the set). `P.func` is total if `P` is total. Or, to turn this around, if we have a total function `f` such that `∀ x | P(x, f(x))`, then certainly `∀ x | ∃ y | P(x, y)`; in fact, `y = f(x)` will do. Amazing.

Summary of logic

The ∧ and ∨ operators are commutative and associative and distribute over each other.

DeMorgan's laws: `~ (a ∧ b) = ~a ∨ ~b`
`~ (a ∨ b) = ~a ∧ ~b`

Any expression has a unique (up to ordering) disjunctive normal form in which ∨ combines terms in which ∨ combines (possibly negated) variables: $(a_1 \wedge \sim a_2 \wedge ...) \vee (\sim b_1 \wedge b_2 \wedge ...) \vee ...$

Implication: `(a ⇒ b) = ~a ∨ b`

Implication is the ordering in a lattice (a partially ordered set in which every subset has a least upper and a greatest lower bound) with

top	= `false`	so `false ⇒ true`
bottom	= `true`	
meet	= ∧	least upper bound, so `(a ∧ b) ⇒ a`
join	= ∨	greatest lower bound, so `a ⇒ (a ∨ b)`

For all `x`, `P(x)`:

`∀ x | P(x) = P(x1) ∧ P(x2) ∧ ...`

There exists an `x` such that `P(x)`:

`∃ x | P(x) = P(x1) ∨ P(x2) ∨ ...`

[7] In general this might not be a countable set, so the conjunction and disjunction are written in a somewhat misleading way, but this complication won't make any difference to us.

Index for logic

3. Introduction to Spec

This handout explains what the Spec language is for, how to use it effectively, and how it differs from a programming language like C, Pascal, Clu, Java, or Scheme. Spec is very different from these languages, but it is also much simpler. Its meaning is clearer and Spec programs are more succinct and less burdened with trivial details. The handout also introduces the main constructs that are likely to be unfamiliar to a programmer. You will probably find it worthwhile to read it over more than once, until those constructs are familiar. Don't miss the one-page summary of spec at the end. The handout also has an index.

Spec is a language for writing precise descriptions of digital systems, both sequential and concurrent. In Spec you can write something that differs from practical code (for instance, code written in C) only in minor details of syntax. This sort of thing is usually called a program. Or you can write a very high level description of the behavior of a system, usually called a specification. A good specification is almost always quite different from a good program. You can use Spec to write either one, but not the same *style* of Spec. The flexibility of the language means that you need to know the purpose of your Spec in order to write it well.

Most people know a lot more about writing programs than about writing specs, so this introduction emphasizes how Spec differs from a programming language and how to use it to write good specs. It does not attempt to be either complete or precise, but other handouts fill these needs. The *Spec Reference Manual* (handout 4) describes the language completely; it gives the syntax of Spec precisely and the semantics informally. *Atomic Semantics of Spec* (handout 9) describes precisely the meaning of an atomic command; here 'precisely' means that you should be able to get an unambiguous answer to any question. The section "Non-Atomic Semantics of Spec" in handout 17 on formal concurrency describes the meaning of a non-atomic command.

Spec's notation for commands, that is, for changing the state, is derived from Edsger Dijkstra's guarded commands (E. Dijkstra, *A Discipline of Programming*, Prentice-Hall, 1976) as extended by Greg Nelson (G. Nelson, A generalization of Dijkstra's calculus, *ACM TOPLAS* **11**, 4, Oct. 1989, pp 517-561). The notation for expressions is derived from mathematics.

This handout starts with a discussion of specifications and how to write them, with many small examples of Spec. Then there is an outline of the Spec language, followed by three extended examples of specs and code. At the end are two handy tear-out one-page summaries, one of the language and one of the official POCS strategy for writing specs and code.

In the language outline, the parts in small type describe less important features, and you can skip them on first reading.

What is a specification for?

The purpose of a specification is to communicate precisely all the essential facts about the behavior of a system. The important words in this sentence are:

communicate	The spec should tell both the client and the implementer what each needs to know.
precisely	We should be able to prove theorems or compile machine instructions based on the spec.
essential	Unnecessary requirements in the spec may confuse the client or make it more expensive to implement the system.
behavior	We need to know exactly what we mean by the behavior of the system.

Communication

Spec mediates communication between the client of the system and its implementer. One way to view the spec is as a contract between these parties:

> The client agrees to depend only on the system behavior expressed in the spec; in return it only has to read the spec, and it can count on the implementer to provide a system that actually does behave as the spec says it should.
>
> The implementer agrees to provide a system that behaves according to the spec; in return it is free to arrange the internals of the system however it likes, and it does not have to deliver anything not laid down in the spec.

Usually the implementer of a spec is a programmer, and the client is another programmer. Usually the implementer of a program is a compiler or a computer, and the client is a programmer.

Usually the system that the implementer provides is called an implementation, but in this course we will call it *code* for short. It doesn't have to be C or Java code; we will give lots of examples of code in Spec which would still require a lot of work on the details of data structures, memory allocation, etc. to turn it into an executable program. You might wonder what good this kind of high-level code is. It expresses the difficult parts of the design clearly, without the straightforward details needed to actually make it run.

Behavior

What do we mean by behavior? In real life a spec defines not only the functional behavior of the system, but also its performance, cost, reliability, availability, size, weight, etc. In this course we will deal with these matters informally if at all. The Spec language doesn't help much with them.

Spec is concerned only with the possible state transitions of the system, on the theory that the possible state transitions tell the complete story of the functional behavior of a digital system. So we make the following definitions:

> A *state* is the values of a set of names (for instance, `x=3, color=red`).

A *history* is a sequence of states such that each pair of adjacent states is a transition of the system (for instance, `x=1; x=2; x=5` is the history if the initial state is `x=1` and the transitions are "if `x = 1` then `x := x + 1`" and "if `x = 2` then `x := 2 * x + 1`").

A *behavior* is a set of histories (a non-deterministic system can have more than one history, usually at least one for every possible input).

How can we specify a behavior?

One way to do this is to just write down all the histories in the behavior. For example, if the state just consists of a single integer, we might write

```
 1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1
 1  2  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1
...
 1  2  1  2  1  2  1  2  1  2  1  2  1  2  1  2  1  2
....
 1  2  3  4  5  1  2  3  1  2  3  4  5  6  7  8  9 10
```

The example reveals two problems with this approach:

The sequences are long, and there are a lot of them, so it takes a lot of space to write them down. In fact, in most cases of interest the sequences are infinite, so we can't actually write them down.

It isn't too clear from looking at such a set of sequences what is really going on.

Another description of this set of sequences from which these examples are drawn is "18 integers, each one either `1` or one more than the preceding one." This is concise and understandable, but it is not formal enough either for mathematical reasoning or for directions to a computer.

Precise

In Spec the set of sequences can be described in many ways, for example, by the expression

```
{q: SEQ Int |    q.size = 18
             /\ (ALL i: Int | 0 <= i /\ i < q.size ==>
                              q(i) = 1 \/ (i > 0 /\ q(i) = q(i-1) + 1)) }
```

Here the expression in `{...}` is very close to the usual mathematical notation for defining a set. Read it as "The set of all `q` which are sequences of integers such that `q.size = 18` and ...". Spec sequences are indexed from `0`. The `(ALL ...)` is a universally quantified predicate, and `==>` stands for implication, since Spec uses the more familiar `=>` for 'then' in a guarded command. Throughout Spec the '`|`' symbol separates a declaration of some new names and their types from the scope in which they are meaningful.

Alternatively, here is a state machine that generates the sequences we want. We specify the transitions of the machine by starting with primitive *assignment commands* and putting them together with a few kinds of compound commands. Each command specifies a set of possible transitions.

```
VAR i, j |
    << i := 1; j := 1 >> ;
    DO << j < 18 => BEGIN i := 1 [] i := i+1 END; Output(i); j := j+1 >> OD
```

Here there is a good deal of new notation, in addition to the familiar semicolons, assignments, and plus signs.

`VAR i, j |` introduces the local variables `i` and `j` with arbitrary values. Because `;` binds more tightly than `|`, the scope of the variables is the rest of the example.

The `<< ... >>` brackets delimit the atomic actions or transitions of the state machine. All the changes inside these brackets happen as one transition of the state machine.

`j < 18 => ...` is a transition that can only happen when `j < 18`. Read it as "if `j < 18` then `...`". The `j < 18` is called a *guard*. If the guard is false, we say that the entire command *fails*.

`i := 1 [] i := i + 1` is a *non-deterministic* transition which can either set `i` to `1` or increment it. Read `[]` as 'or'.

The `BEGIN ... END` brackets are just brackets for commands, like { ... } in C. They are there because `=>` binds more tightly than the `[]` operator inside the brackets; without them the meaning would be "either set `i` to `1` if `j < 18` or increment `i` and `j` unconditionally".

Finally, the `DO ... OD` brackets mean: repeat the `...` transition as long as possible. Eventually `j` becomes `18` and the guard becomes false, so the command inside the `DO ... OD` fails and can no longer happen.

The expression approach is better when it works naturally, as this example suggests, so Spec has lots of facilities for describing values: sequences, sets, and functions as well as integers and booleans. Usually, however, the sequences we want are too complicated to be conveniently described by an expression; a state machine can describe them much more easily.

State machines can be written in many different ways. When each transition involves only simple expressions and changes only a single integer or boolean state variable, we think of the state machine as a program, since we can easily make a computer exhibit this behavior. When there are transitions that change many variables, non-deterministic transitions, big values like sequences or functions, or expressions with quantifiers, we think of the state machine as a spec, since it may be much easier to understand and reason about it, but difficult to make a computer exhibit this behavior. In other words, large atomic actions, non-determinism, and expressions that compute sequences or functions are hard to code. It may take a good deal of ingenuity to find code that has the same behavior but uses only the small, deterministic atomic actions and simple expressions that are easy for the computer.

Essential

The hardest thing for most people to learn about writing specs is that *a spec is not a program*. A spec defines the behavior of a system, but unlike a program it need not, and usually should not, give any practical method for producing this behavior. Furthermore, it should pin down the behavior of the system only enough to meet the client's needs. Details in the spec that the client doesn't need can only make trouble for the implementer.

The example we just saw is too artificial to illustrate this point. To learn more about the difference between a spec and code consider the following:

```
CONST eps := 10**-8

APROC SquareRoot0(x: Real) -> Real =
    << VAR y : Real | Abs(x - y*y) < eps => RET y >>
```

(Spec as described in the reference manual doesn't have a `Real` data type, but we'll add it for the purpose of this example.)

The combination of `VAR` and `=>` is a very common Spec idiom; read it as "choose a `y` such that `Abs(x - y*y) < eps` and do `RET y`". Why is this the meaning? The `VAR` makes a choice of any `Real` as the value of `y`, but the entire transition on the second line cannot occur unless the guard `Abs(x - y*y) < eps` is true. Hence the `VAR` must choose a value that satisfies the guard.

What can we learn from this example? First, the result of `SquareRoot0(x)` is not completely determined by the value of `x`; any result whose square is within `eps` of `x` is possible. This is why `SquareRoot0` is written as a procedure rather than a function; the result of a function has to be determined by the arguments and the current state, so that the value of an expression like `f(x) = f(x)` will be `true`. In other words, `SquareRoot0` is *non-deterministic*.

Why did we write it that way? First of all, there might not be any `Real` (that is, any floating-point number of the kind used to represent `Real`) whose square exactly equals `x`. We could accommodate this fact of life by specifying the closest floating-point number.[1] Second, however, we may not want to pay for code that gives the closest possible answer. Instead, we may settle for a less accurate answer in the hope of getting the answer faster.

You have to make sure you know what you are doing, though. This spec allows a negative result, which is perhaps not what we really wanted. We could have written (highlighting changes with boxes):

```
APROC SquareRoot1(x: Real) -> Real =
    << VAR y : Real | y >= 0 /\ Abs(x - y*y) < eps => RET y >>
```

to rule that out. Also, the spec produces no result if `x < 0`, which means that `SquareRoot1(-1)` will fail (see the section on commands for a discussion of failure). We might prefer a total function that raises an exception:

```
APROC SquareRoot2(x: Real) -> Real RAISES {undefined} =
    <<     x >= 0 => VAR y : Real | y >= 0 /\ Abs(x - y*y) < eps => RET y
      [*]  RAISE undefined >>
```

The `[*]` is 'else'; it does its second operand iff the first one fails. Exceptions in Spec are much like exceptions in CLU. An exception is contagious: once started by a `RAISE` it causes any containing expression or command to yield the same exception, until it runs into an exception handler (not shown here). The `RAISES` clause of a routine declaration must list all the exceptions that the procedure body can generate, either by `RAISES` or by invoking another routine.

Code for this spec would look quite different from the spec itself. Instead of the existential quantifier implied by the `VAR y`, it would have an algorithm for finding `y`, for instance, Newton's method. In the algorithm you would only see operations that have obvious codes in terms of the load, store, arithmetic, and test instructions of a computer. Probably the code would be deterministic.

Another way to write these specs is as functions that return the set of possible answers. Thus

```
FUNC SquareRoots1(x: Real) -> SET Real =
    RET {y : Real | y >= 0 /\ Abs(x - y*y) < eps}
```

[1] This would still be non-deterministic in the case that two such numbers are equally close, so if we wanted a deterministic spec we would have to give a rule for choosing one of them, for instance, the smaller.

Note that the form inside the `{...}` set constructor is the same as the guard on the `RET`. To get a single result you can use the set's `choose` method: `SquareRoots1(2).choose`.[2]

In the next section we give an outline of the Spec language. Following that are three extended examples of specs and code for fairly realistic systems. At the end is a one-page summary of the language.

An outline of the Spec language

The Spec language has two main parts:

- An *expression* describes how to compute a result (a value or an exception) as a function of other values: either literal constants or the current values of state variables.
- A *command* describes possible transitions of the state variables. Another way of saying this is that a command is a relation on states: it allows a transition from `s1` to `s2` iff it relates `s1` to `s2`.

Both are based on the *state*, which in Spec is a mapping from names to values. The names are called state variables or simply variables: in the sequence example above they are `i` and `j`. Actually a command relates states to *outcomes*; an outcome is either a state (a normal outcome) or a state together with an exception (an exceptional outcome).

There are two kinds of commands:

- An *atomic* command describes a set of possible transitions, or equivalently, a set of pairs of states, or a relation between states. For instance, the command `<< i := i + 1 >>` describes the transitions `i=1`→`i=2`, `i=2`→`i=3`, etc. (Actually, many transitions are summarized by `i=1`→`i=2`, for instance, `(i=1, j=1)`→`(i=2, j=1)` and `(i=1, j=15)`→`(i=2, j=15)`). If a command allows more than one transition from a given state we say it is non-deterministic. For instance, on page 3 the command `BEGIN i := 1 [] i := i + 1 END` allows the transitions `i=2`→`i=1` and `i=2`→`i=3`, with the rest of the state unchanged.
- A *non-atomic* command describes a set of *sequences* of states (by contrast with the set of pairs for an atomic command). More on this below.

A sequential program, in which we are only interested in the initial and final states, can be described by an atomic command.

The meaning of an expression, which is a function from states to values (or exceptions), is much simpler than the meaning of an atomic command, which is a relation between states, for two reasons:

- The expression yields a single value rather than an entire state.
- The expression yields at most one value, whereas a non-deterministic command can yield many final states.

[2] `r := SquareRoots1(x).choose` (using the function) is almost the same as `r := SquareRoot1(x)` (using the procedure). The difference is that because `choose` is a function it always returns the same element (even though we don't know in advance which one) when given the same set, and hence when `SquareRoots1` is given the same argument. The procedure, on the other hand, is non-deterministic and can return different values on successive calls, so that spec is weaker. Which one is more appropriate?

An atomic command is still simple, because its meaning is just a relation between states. The relation may be partial: in some states there may be no way to execute the command. When this happens we say that the command is not *enabled* in those states. As we saw, the relation is not necessarily a function, since the command may be non-deterministic.

A non-atomic command is much more complicated than an atomic command, because:

- Taken in isolation, the meaning of a non-atomic command is a relation between an initial state and a history. A history is a whole sequence of states, much more complicated than a single final state. Again, many histories can stem from a single initial state.
- The meaning of the (parallel) composition of two non-atomic commands is not any simple combination of their relations, such as the union, because the commands can interact if they share any variables that change.

These considerations lead us to describe the meaning of a non-atomic command by breaking it down into its atomic subcommands and connecting these up with a new state variable called a program counter. The details are somewhat complicated; they are sketched in the discussion of atomicity below, and described in handout 17 on formal concurrency.

The moral of all this is that you should use the simpler parts of the language as much as possible: expressions rather than atomic commands, and atomic commands rather than non-atomic ones. To encourage this style, Spec has a lot of syntax and built-in types and functions that make it easy to write expressions clearly and concisely. You can write many things in a single Spec expression that would require a number of C statements, or even a loop. Of course, code with a lot of concurrency will necessarily have more non-atomic commands, but this complication should be put off as long as possible.

Organizing the program

In addition to the expressions and commands that are the core of the language, Spec has four other mechanisms that are useful for organizing your program and making it easier to understand.

- A *routine* is a named computation with parameters, in other words, an abstraction of the computation. Parameters are passed by value. There are four kinds of routine:

  A *function* (defined with `FUNC`) is an abstraction of an expression.

  An *atomic procedure* (defined with `APROC`) is an abstraction of an atomic command.

  A general procedure (defined with `PROC`) is an abstraction of a non-atomic command.

  A *thread* (defined with `THREAD`) is the way to introduce concurrency.

- A *type* is a highly stylized assertion about the set of values that a name or expression can assume. A type is also a convenient way to group and name a collection of routines, called its *methods*, that operate on values in that set.
- An *exception* is a way to report an unusual outcome.
- A *module* is a way to structure the name space into a two-level hierarchy. An identifier `i` declared in a module `m` has the name `m.i` throughout the program. A *class* is a module that can be instantiated many times to create many objects, much like a Java class.

A Spec program is some global declarations of variables, routines, types, and exceptions, plus a set of modules each of which declares some variables, routines, types, and exceptions.

The next two sections describe things about Spec's expressions and commands that may be new to you. They should be enough for the Spec you will read and write in this course, but they don't answer every question about Spec; for those answers, read the reference manual and the handouts on Spec semantics.

Paragraphs in small print contain material that you might want to skip on first reading.

There is a one-page summary of the Spec language at the end of this handout.

Expressions, types, and relations

Expressions are for computing functions of the state.

A Spec expression is	*and its value is*
a constant	the constant
a variable	the current value of the variable
an invocation of a function on an argument that is some sub-expression	the value of the function at the value of the argument

There are no side-effects; those are the province of commands. There is quite a bit of syntactic sugar for function invocations. An expression may be undefined in a state; if a simple command evaluates an undefined expression, the command fails (see below).

Types

A Spec type defines two things:

A set of values; we say that a value *has* the type if it's in the set. The sets are not disjoint. If `T` is a type, `T.all` is its set of values.

A set of functions called the *methods* of the type. There is convenient syntax `v.m` for invoking method `m` on a value `v` of the type. A method `m` of type `T` is lifted to a method `m` of a set of `T`'s, a function `U->T`, or a relation from `U` to `T` in the obvious way, by applying it to the set elements or the result of the function or relation, unless overridden by a different `m` in the definition of the higher type. Thus if `int` has a `square` method, `{2, 3, 4}.square = {4, 9, 16}`. We'll see that this is a form of function composition.

Spec is strongly typed. This means that you are supposed to declare the types of your variables, just as you do in Java. In return the language defines a type for every expression[3] and ensures that the value of the expression always has that type. In particular, the value of a variable always has the declared type. You should think of a type declaration as a stylized comment that has a precise meaning and can be checked mechanically.

If `Foo` is a type, you can omit it in a declaration of the identifiers `foo, foo1, foo'` etc. Thus

```
VAR int1, bool2, char' | ...
```

[3] Note that a value may have many types, but a variable or an expression has exactly one type: for a variable, it's the declared type, and for a complex expression it's the result type of the top-level function in the expression.

is short for

```
VAR int1: Int, bool2: Bool, char': Char | ...
```

Note that this can be confusing in a declaration like `t, u: Int`, where `u` has type `U`, not type `Int`.

If `e IN T.all` then `e AS T` is an expression with the same value and type `T`; otherwise it's undefined. You can write `e IS T` for `e IN T.all`.

Spec has the usual types:

- `Int`, `Nat` (non-negative `Int`), `Bool`
- sets `SET T`
- functions `T->U`
- relations `T->>U`
- records or structs `[f1: T1, f2: T2, ...]`
- tuples `(T1, T2, ...)`
- variable-length arrays called sequences, `SEQ T`

A sequence is actually a function whose domain is `{0, 1, ..., n-1}` for some `n`. A record is actually a function whose domain is the field names, as strings. In addition to the usual functions like `"+"` and `"\/"`, Spec also has some less usual operations on these types, which are valuable when you want to suppress code detail; they are called constructors and combinations and are described below.

You can make a type with fewer values using `SUCHTHAT`. For example,

```
TYPE T = Int SUCHTHAT 0 <= t /\ t <= 4
```

has the value set `{0, 1, 2, 3, 4}`. Here the expression following `SUCHTHAT` is short for `(\ t: Int | 0 <= t /\ t <= 4)`, a lambda expression (with `\` for λ) that defines a function from `Int` to `Bool`, and a value has type `T` if it's an `Int` and the function maps it to `true`. You can write `this` for the argument of `SUCHTHAT` if the type doesn't have a name. The type `IN s`, where `s` has type `SET T`, is short for `SET T SUCHTHAT this IN s`.

Methods

Methods are a convenient way of packaging up some functions with a type so that the functions can be applied to values of that type concisely and without mentioning the type itself. For example, if `s` is a `SEQ T`, `s.head` is `(Sequence[T].Head)(s)`, which is just `s(0)` (which is undefined if `s` is empty). You can see that it's shorter to write `s.head`.[4] Note that when you write `e.m`, the method `m` is determined by the static type of `e`, and *not* by the value as in most object-oriented languages.

You can define your own methods by using `WITH`. For instance, consider

```
TYPE Complex = [re: Real, im: Real] WITH {"+":=Add, mag:=Mag}
```

The `[re: Real, im: Real]` is a record type (a `struct` in C) with fields `re` and `im`. `Add` and `Mag` are ordinary Spec functions that you must define, but you can now invoke them on a `c` which is `Complex` by writing `c + c'` and `c.mag`, which mean `Add(c, c')` and `Mag(c)`. You can use existing operator symbols or make up your own; see section 3 of the reference manual for lexical rules. You can also write `Complex."+"` and `Complex.mag` to denote the functions `Add` and `Mag`; this may be convenient if `Complex` was declared in a different module. Using `Add` as a method does not make it private, hidden, static, local, or anything funny like that.

[4] Of course, `s(0)` is shorter still, but that's an accident; there is no similar alternative for `s.tail`.

When you nest `WITH` the methods pile up in the obvious way. Thus

```
TYPE MoreComplex = Complex WITH {"-":=Sub, mag:=Mag2}
```

has an additional method `"-"`, the same `"+"` as `Complex`, and a different `mag`. Many people call this 'inheritance' and 'overriding'.

A method `m` of type `T` is *lifted* automatically to a method of types `V->T`, `V->>T`, and `SET T` by composing it with the value of the higher-order type. This is explained in detail in the discussion of functions below.

Expressions

The syntax for expressions gives various ways of writing function invocations in addition to the familiar `f(x)`. You can use unary and binary operators, and you can invoke a method with `e1.m(e2)` for `T.m(e1,e2)`, of just `e.m` if there are no other arguments. You can also write a lambda expression `(\ t: T | e)` or a conditional expression `(predicate => e1 [*] e2)`, which yields `e1` if `predicate` is true and `e2` otherwise. If you omit `[*] e2`, the result is undefined if `predicate` is false. Because `=>` denotes if ... then, implication is written `==>`.

Here is a list of all the built-in operators, which also gives their precedence, and a list of the built-in methods. You should read these over so that you know the vocabulary. The rest of this section explains many of these and gives examples of their use.

Note that any lattice (any partially ordered set with least upper bound or max, and greatest lower bound or min, defined on any pair of elements) has operators `/\` (max) and `\/` (min). Booleans, sets, and relations are examples of lattices. Any totally ordered set such as `Int` is a lattice.

Binary operators

Op	*Prec.*	*Argument/result types*	*Operation*
`**`	8	`(Int, Int)->Int`	exponentiate
`*`	7	`(Int, Int)->Int`	multiply
		`(T->U, U->V)->(T->V)`	function or relation composition: `(\t | e2(e1(t))`
`/`	7	`(Int, Int)->Int`	divide
`//`	7	`(Int, Int)->Int`	remainder
`+`	6	`(Int, Int)->Int`	add
		`(SEQ T, SEQ T)->SEQ T`	concatenation
		`(T->U, T->U)->(T->U)`	function overlay: `(\t | (e2!t => e2(t) [*] e1(t))`
`-`	6	`(Int, Int)->Int`	subtract
		`(SET T, SET T)->SET T`	set difference
		`(SEQ T, SEQ T)->SEQ T`	multiset difference
`!`	6	`(T->U, T)->Bool`	function is defined at arg
`!!`	6	`(T->U, T)->Bool`	function defined, no exception at arg
`..`	5	`(Int, Int)->SEQ Int`	subrange: `{e1, e1+1, ..., e2}`
`||`	5	`(SEQ T, SEQ U)->SEQ(T,U)`	zip: pair of sequences to sequence of pairs
`<=`	4	`(Int, Int)->Bool`	less than or equal
		`(SET T, SET T)->Bool`	subset
		`(SEQ T, SEQ T)->Bool`	prefix: `e2.restrict(e1.dom) = e1`
`<`	4	`(T, T)->Bool, T with <=`	less than
`>`	4	`(T, T)->Bool, T with <=`	greater than
`>=`	4	`(T, T)->Bool, T with <=`	greater or equal
`=`	4	`(Any, Any)->Bool`	can't override by `WITH`
`#`	4	`(Any, Any)->Bool`	not equal; can't override by `WITH`
`<<=`	4	`(SEQ T, SEQ T)->Bool`	non-contiguous sub-seq: `(∃s|s<=e2.dom ∧ s.sort*e2=e1)`
`IN`	4	`(T, SET T)->Bool`	membership
`/\`	2	`(Bool, Bool)->Bool`	conditional and*

		`(T, T)->T`	max, for any lattice; example: set/relation intersection
`\/`	1	`(Bool, Bool)->Bool`	conditional or*
		`(T, T)->T`	min, for any lattice; example: set/relation union
`==>`	0	`(Bool, Bool)->Bool`	conditional implies*
`op`	5	`(T, U)->V`	`op` none of the above: `T."op"(`e_1`, `e_2`)`

The "*" on the conditional Boolean operators means that, unlike all other operators, they don't evaluate their second argument if the first one determines the result. Thus `f(x) /\ g(x)` is `false` if `f(x)` is false, even if `g(x)` is undefined.

Unary operators

Op	*Prec.*	*Argument/result types*	*Operation*
`-`	6	`Int->Int`	negation
`~`	3	`Bool->Bool`	complement
		`SET T->SET T`	set complement
		`(T->>U)->(T->>U)`	relation complement
`op`	5	`T->U`	`op` none of the above: `T."op"(`e_1`)`

Relations

A relation `r` is a generalization of a function: an arbitrary set of ordered pairs, defined by a predicate, a total function from pairs to `Bool`. Thus `r` can relate an element of its domain to any number of elements of its range (including none). Like a function, `r` has `dom`, `rng`, and `inv` methods (the inverse is obtained just by flipping the ordered pairs), and you can compose relations with `*`. Note that in general `r * r.inv` is not the identity; for this reason many people prefer to call it the "transpose" or "converse". You can also take the complement, union, and intersection of two relations that have the same type, and compare relations with <= and its friends. These all work like the same operators on the sets of ordered pairs. The `pToR` method converts a predicate on pairs to a relation.

Examples:

The relation < on `Int`. Its domain and range are `Int`, and its inverse is >.

The relation `r` given by the set of ordered pairs `s = {("a", 1), ("b", 2), ("a", 3)}`; `r = s.pred.pToR`; that is, turn the set into a predicate on ordered pairs and the predicate into a relation. Its inverse `r.inv = {(1, "a"), (2, "b"), (3, "a")}`, which is the sequence `{"a", "b", "a"}`. Its domain `r.dom = {"a", "b"}`; its range `r.rng = {1, 2, 3}`.

The advantage of relations is simplicity and generality; for example, there's no notion of "undefined" for relations. The drawback is that you can't write `r(x)` (although you can write `{x} * r` for the set of values related to `x` by `r`; see below).

A relation `r` has methods

`r.setF` to turn it into a set function: `r.setF(x)` is the set of elements that `r` relates to `x`. This is total. `Int."<".setF = (\i | {j: Int | j < i})`, and in the second example, `r.setF` maps `"a"` to `{1, 4}`) and `"b"` to `{2}`. The inverse of `setF` is the `setRel` method for a function whose values are sets: `r.setF.setRel = r`, and `f.setRel.setF = f` if `f` yields sets.

`r.fun` to turn it into a function: `r.fun(x)` is undefined unless `r` relates `x` to exactly one value. Thus `r.fun = r.setF.one`.

If `s` is a set, `s.id` relates every member of the set to itself, and `s.rel` is a relation that relates `true` to each member of the set; thus it is `s.pred.inv.restrict({true})`. The relation's `rng` method inverts this: `s.rel.rng = s`.

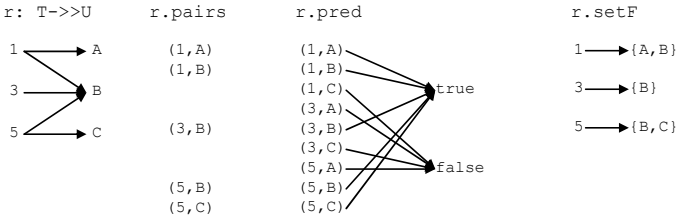
Viewing a set as a relation, you can compose it with a relation (or a function viewed as a relation); the result is the image of the set under the relation: `s * r = (s.rel * r).rng`. Note that this is never undefined, unlike sequence composition.

A relation `r: T->>U` can be viewed as a set `r.pairs` of pairs `(T,U)`, or as a total function `r.pred` on `(T,U)` that is `true` on the pairs that are in the relation, or as a function `r.setF` from `T` to `SET U`.

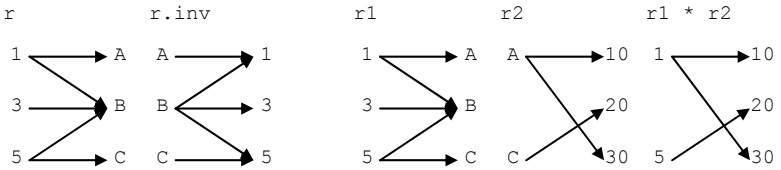
A method `m` of `U` is lifted to `SET U` and to relations to `U` just as it is to functions to `U` (see below), so that `r.m = r * U.m.rel`, as long as the set or relation doesn't have a `m` method.

The Boolean, set, and relational operators are extended to relations, so that `r1 \/ r2` is the union of the relations, `r1 /\ r2` the intersection, and `~r` the complement, and `r1 <= r2` iff `r1.pairs <= r2.pairs`.

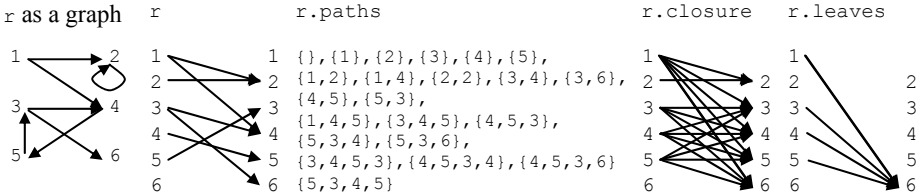
`T = {1,2,3,4,5}; U = {A,a,B,b,C}`



You can compute the inverse of a relation, and compose two relations by matching up the range of the first with the domain of the second.



If a relation `T->>T` has the same range and domain types it represents a graph, on which it makes sense to define the paths through the graph, and the transitive closure of the relation.



The partial inverse of `paths` is `pRel`; it takes a sequence to the relation that holds exactly between adjacent elements. So `\/ :r.paths.pRel = r`, and if the elements of `q` are distinct, `q` is the longest element of `q.pRel.paths`. The set `r.reachable(s)` is the elements reachable through `r` from a starting set `s`.

Method call	*Result type*	*Definition*
`r.pred`	`(T,U) ->Bool`	definition; `(\t,u | u IN r.setF(r))`
`r.pairs`	`SET (T,U)`	`{true} * r.pred.inv`
`r.set`	`SET T`	`r.rng`; only for `R = Bool->>T`
`r * rr`	`T->>V`	`(\t,v | (EXISTS u | r.pred(t,u) /\ rr.pred(u,v))).pToR` where `rr: U->>V`; works for `f` as well as `rr`
`r.dom`	`SET T`	`U.all * r.inv`
`r.rng`	`SET U`	`T.all * r`
`r.inv`	`U->>T`	`(\t,u | r.pred(u,t)).pToR`
`r.restrict(s)`	`T->>U`	`s.id * r` where `s: SET T`
`r.setF`	`T-> SET U`	`(\t | {t} * r)`
`r.fun`	`T->U`	`r.setF.one` (`one` is lifted from `SET U` to `T->SET U`)
`r.paths`	`SET SEQ T`	`{q:SEQ T | (ALL i IN q.dom-{0}|r.pred(q(i-1),q(i))) /\ ( q.rng.size=q.size \/ (q.head=q.last /\ q.rng.size=q.size-1))}` only for `R=T->>T`; paths self-intersect only at endpoints. See `q.pRel` for inverse.
`r.closure`	`T->>T`	`{q IN r.paths | q.size>1 || (q.head, q.last)}.pred.pToR` only for `R=T->>T`; there's a non-trivial path from `t1` to `t2`
`r.leaves`	`T->>T`	`r * (r.rng - r.dom).id` only for `R=T->>T`; there's a direct path from `t1` to `t2`, but nothing beyond.
`r.reachable(s)`	`SET T`	`(+ :{q :IN r.paths - {} | q.head IN s}).set`

Sets

A set has methods for

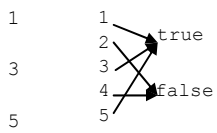
computing union, intersection, and set difference (lifted from `Bool`; see note 3 in section 4), and adding or removing an element, testing for membership and subset;

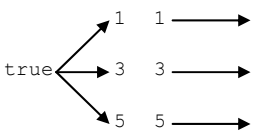
choosing (deterministically) a single element from a set, or a sequence with the same members, or a maximum or minimum element, and turning a set into its characteristic predicate (the inverse is the predicate's `set` method);

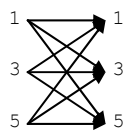
composing a set with a function or relation, and converting a set into a relation from `nil` to the members of the set (the inverse of this is just the range of the relation).

A set `s: SET T` can be viewed as a total function `s.pred` on `T` that is `true` on the members of `s` (sometimes called the 'characteristic function'), or as a relation `s.rel` from `true` to the members of the set, or as the identity relation `s.id` that relates each member to itself, or as the universal relation `s.univ` that relates all the members to each other.

`s ={1,3,5}` `s.pred` `s.rel = s.pred.inv .restrict ({true})` `s.id` `s.univ` `s.include`







Error! Objects cannot be created from editing field codes.

You can compose a set `s` with a function or a relation to get another set, which is the image of `s` under the function or relation.

`{1,3,5}` `* square` `=`

`{1,5}` `* r` `=`

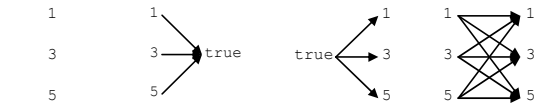
This is just like relational composition on `s.rel`.

`{1,3,5}.rel` `* square` `=`

`{1,5}.rel` `* r` `=`

The universal relation `s.univ` is just the composition of `s.rel` with its inverse:

`s ={1,3,5}` `s.rel.inv` `* s.rel =` `s.univ`

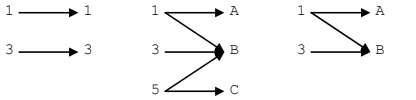


You can compute the range and domain of a relation. An element `t` is in the range if `r` relates something to it, and in the domain if `r` relates it to something. (For clarity, the figures show the relations corresponding to the sets, not the sets themselves.)

`T.all *` `r =` `r.rng` `U.all *` `r.inv =` `r.dom`

You can restrict the domain of a relation or function to a set `s` by composing the identity relation `s.id` with it. To restrict the range to `s`, use the same idea and write `r * s.id`.

`{1,3}.id *` `r =` `r.restrict( {1,3})`



You can convert a set of pairs `s` to a relation with `s.pred.pToR`; there are examples in the section on relations above.

You can pick out one element of a set s with `s.choose`. This is deterministic: `choose` always returns the same value given the same set (a necessary property for it to be a function). It is undefined if the set is empty. A variation of `choose` is `one`: `s.one` is undefined unless `s` has exactly one element, in which case it returns that element.

You can compute the set of all permutations of a set; a permutation is a sequence, explained below. You can sort a set or compute its maximum or minimum; note that the results make an arbitrary choice if the ordering function is not a total order. You can also compute the "leaves" that a

relation computes from a set: the extremal points where the relation takes the elements of the set; here you get them all, so there's no need for an arbitrary choice. If you think of the graph induced by the closure of the relation, starting from the elements of the set, then the leaves are the nodes of the graph that have no outgoing edges (successors).

```
s = {3,1,5}, s.perms ={{3,1,5},{3,5,1},{5,1,3},{5,3,1},{{1,3,5},{1,5,3}},
s.sort = {1,3,5}, s.max = 5, s.min = 3.
```

Method call	*Result type*	*Definition*
`s.pred`	`T->Bool`	definition; `(\t | t IN s)`
`s.rel`	`Bool->>T`	`s.pred.inv`
`s.id`	`T->>T`	`(\ t1,t2 | t1 IN s /\ t1 = t2)`
`s.univ`	`T->>T`	`s.rel.inv * s.rel`
`s.include`	`SET T->>T`	`(\ st: SET T, t | t IN (st /\ s)).pToR`
`t IN s`	`Bool`	`s.pred(t)`
`s1 <= s2`	`Bool`	`s1 /\ s2 = s1`, or equivalently `(∀ t | t IN s1 ==> t IN s2)`
`s1 /\ s2`	`S`	`(\t | t IN s1 /\ t IN s2)` intersection
`s1 \/ s2`	`S`	`(\t | t IN s1 \/ t IN s2)` union
`~ s`	`S`	`(\t | ~(t IN s))`
`s1 - s2`	`S`	`s1 /\ ~ s2`
`s * r`	`SET U`	`(s.rel * r).rng` where `R=T->>U`; works for `f` as well as `r`
`s.size`	`Nat`	`s.seq.dom.max + 1`
`s.choose`	`T`	`?`
`s.one`	`T`	`(s.size = 1 => s.choose)`; undefined if `s#{t}`
`s.perms`	`SET Q`	`{q: SEQ T | q.size = s.size /\ q.rng = s}`
`s.seq`	`Q`	`s.perms.choose`
`s.fsort(f)`	`Q`	`{q IN s.perms|(∀i IN q.dom-{0}|f(q(i),q(i-1)))}.choose`
`s.sort`	`Q`	`s.fsort(T."<=")`
`s.fmax(f)`	`T`	`s.fsort(f).last` and likewise for `fmin`
`s.max`	`T`	`s.sort.last` and likewise for `min`. Note that this is not the same as `/\ : s`, unless `s` is totally ordered.
`s.leaves(r)`	`S`	`r.restrict(s).closure.leaves.rng`; generalizes `max`
`s.combine(f)`	`T`	`s.seq.combine(f)`; useful if `f` is commutative

Functions

A function is a set of ordered pairs; the first element of each pair comes from the function's *domain*, and the second from its *range*. A function produces at most one value for an argument; that is, two pairs can't have the same first element. Thus a function is a relation in which each element of the domain is related to at most one thing. A function may be partial, that is, undefined at some elements of its domain. The expression `f!x` is true if `f` is defined at `x`, false otherwise. Like everything (except types), functions are ordinary values in Spec.

Given a function, you can use a function constructor to make another one that is the same except at a particular argument, as in the `DB` example in the section on constructors below. Another example is `f{x -> 0}`, which is the same as `f` except that it is 0 at `x`. If you have never seen a construction like this one, think about it for a minute. Suppose you had to implement it. If `f` is represented as a table of (argument, result) pairs, the code will be easy. If `f` is represented by code that computes the result, the code for the constructor is less obvious, but you can make a new piece of code that says

```
(\ y: Int | ( (y = x) => 0 [*] f(y) ))
```

Here '`\`' is 'lambda', and the subexpression `( (y = x) => 0 [*] f(y) )` is a conditional, modeled on the conditional commands we saw in the first section; its value is `0` if `y = x` and `f(y)` otherwise, so we have changed `f` just at `0`, as desired. If the else clause `[*] f(y)` is omitted, the condition is undefined if `y # x`. Of course in a running program you probably wouldn't want to construct new functions very often, so a piece of Spec that is intended to be close to practical code must use function constructors carefully.

Functions can return functions as results. Thus `T->U->V` is the type of a function that takes a `T` and returns a function of type `U->V`, which in turn takes a `U` and returns a `V`. If `f` has this type, then `f(t)` has type `U->V`, and `f(t)(u)` has type `V`. Compare this with `(T, U)->V`, the type of a function which takes a `T` and a `U` and returns a `V`. If `g` has this type, `g(t)` doesn't type-check, and `g(t, u)` has type `V`. Obviously `f` and `g` are closely related, but they are not the same. Functions declared with more than one argument are a bit tricky; they are discussed in the section on tuples below.

You can define your own functions either by lambda expressions like the one above, or more generally by function declarations like this one

```
FUNC NewF(y: Int) -> Int = RET ( (y = x) => 0 [*] f(y) )
```

The value of this `NewF` is the same as the value of the lambda expression. To avoid some redundancy in the language, the meaning of the function is defined by a command in which `RET` subcommands specify the value of the function. The command might be syntactically nondeterministic (for instance, it might contain `VAR` or `[]`), but it must specify at most one result value for any argument value; if it specifies no result values for an argument or more than one value, the function is undefined there. If you need a full-blown command in a function constructor, you can write it with `LAMBDA` instead of `\`:

```
(LAMBDA (y: Int) -> Int = RET ( (y = x) => 0 [*] f(y) ))
```

You can *compose* two functions with the `*` operator, writing `f * g`. This means to apply `f` first and then `g`, so you read it "`f` then `g`". It is often useful when `f` is a sequence (remember that a `SEQ T` is a function from `{0, 1, ..., size-1}` to `T`), since the result is a sequence with every element of `f` mapped by `g`. This is Lisp's or Scheme's "map". So:

```
(0 .. 4) * {\ i: Int | i*i} = (SEQ Int){0, 1, 4, 9, 16}
```

since `0 .. 4 = {0, 1, 2, 3, 4}` because `Int` has a method `..` with the obvious meaning: `i .. j = {i, i+1, ..., j-1, j}`. In the section on constructors below we see another way to write

```
(0 .. 4) * {\ i: Int | i*i},
```

as

```
{i :IN 0 .. 4 || i*i}.
```

This is more convenient when the mapping function is defined by an expression, as it is here, but it's less convenient if the mapping function already has a name. Then it's shorter and clearer to write

```
(0 .. 4) * factorial
```

rather than

```
{i :IN 0 .. 4 || factorial(i)}.
```

A function `f` has methods `f.dom` and `f.rng` that yield its domain and range sets, `f.inv` that yields its inverse (which is undefined at `y` unless `f` maps exactly one argument to `y`), and `f.rel` that turns it into a relation (see below). `f.restrict(s)` is the same as `f` on elements of `s` and undefined elsewhere. The *overlay* operator combines two functions, giving preference to the second: `(f1 + f2)(x)` is `f2(x)` if that is defined and `f1(x)` otherwise. So `f{3 -> 24} = f + {3 -> 24}`.

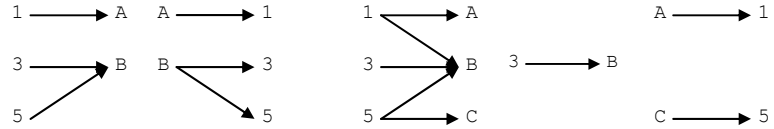
If type `U` has method `m`, then the function type `F = T->U` has a "lifted" method `m` that composes `U.m` with `f`, unless `F` already has a `m` method. `F.m` is defined by

```
(\ f | (\ t | f(t).m))
```

so that `f.m = f * U.m`. For example, `{1, 3, 5}.square = {1, 9, 25}`. If `m` takes a second argument of type `W`, then `F.m` takes a second argument of the same type and uses it uniformly. This also works for sets and relations.

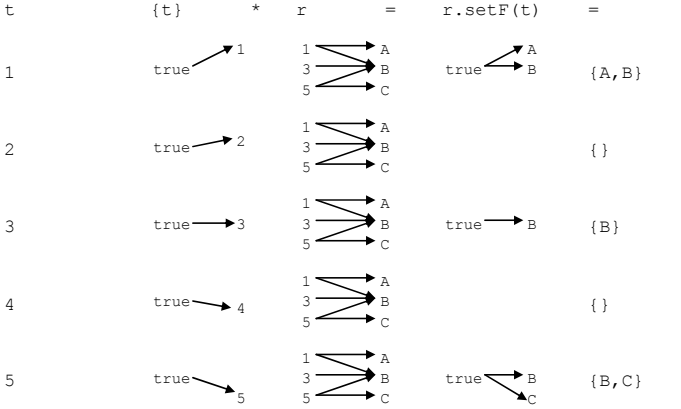
You can turn a relation into a function by discarding all the pairs whose first element is related to more than one thing

`f` `f.inv` (not a function) `r` `r.fun` `r.inv.fun`

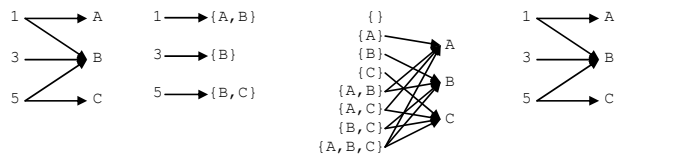


You can go back and forth between a relation `T->>U` and a function `T->SET U` with the `setF` and `setRel` methods.

```
r.setF = (\t | {t} * r)
```




```
f.setRel = f.rel.include
```



Method call	*Result type*	*Definition*
`f` has type `T->U`		
`f + f'`	`T->U`	`(f.rel \/ (f'.rel * f1.rng.id)).func (\t | (f!t => f (t) [*] f'(t)))`
`f!t`	`Bool`	`t IN f.dom`
`f!!t`	`Bool`	
`f $ t`	`U`	Applies `f` to the tuple `t`; see the section on records below
`f * g`	`T->V`	`(f.rel * g.rel).fun`, where `g:U->V`
`f.rel`	`T->>U`	`(\t,u | f!t /\ f(t) = u).pToR`
`f.setRel`	`T->>V`	`f.rel.include`, only for `F=T->SET V`
`f.set`	`SET T`	`f.restrict({true}).rng`, only for `F=T->Bool`
`f.pToR`	`V->>W`	definition, only for `F=(V,W)->Bool`; `(\v |{w|f(v,w)}).setRel`

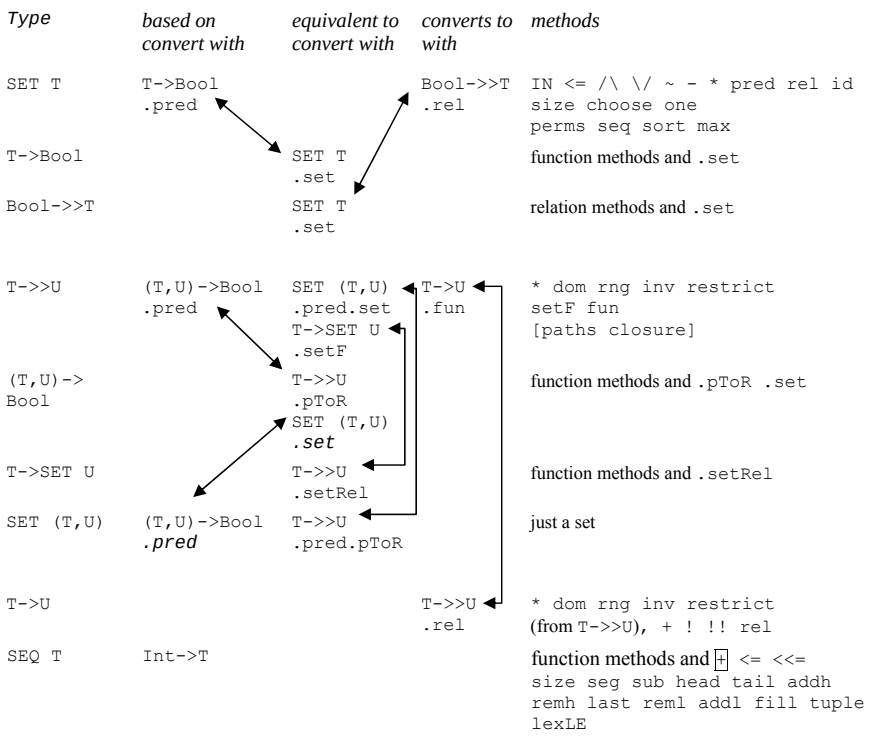
A function type `F = T->U` also has a set of *lifting* methods that turn an `f` into a function on `SET T`, `V->T`, or `V->>T` by composition. This works for `F = (T,W)->U` as well; the lifted method also takes a `W` and uses it uniformly. A relation type `R = T->>U` is also lifted to `SET T`. These are used to automatically supply the higher-order types with lifted methods.

Method	*method `m` of type `T`, with type `F`*	*makes method `m` for type*	*with type*	*by*
`f.liftSet`	`T-> U`	`S =SET T`	`SET T -> SET U`	`s .m=(s * f).set`
`f.liftFun`	`T-> U`	`FF=V-> T`	`(V-> T)->(V-> U)`	`ff.m=ff * f`
`f.liftRel`	`T-> U`	`RR=V->>T`	`(V->>T)->(V->>U)`	`ff.m=rr * f`
`f.liftSet`	`(T,W)->U`	`S =SET T`	`(SET T ,W)->SET U`	`s .m(w)=(s *(\t|f(t,w)).set`
`f.liftFun`	`(T,W)->U`	`FF=V-> T`	`((V-> T),W)->(V-> U)`	`ff.m(w)=ff *(\t|f(t,w))`
`f.liftRel`	`(T,W)->U`	`RR=V->>T`	`((V->>T),W)->(V->>U)`	`ff.m(w)=rr *(\t|f(t,w))`
	with type `R`			
`r.liftSet`	`T->>U`	`S =SET T`	`SET T -> SET U`	`s .m=(s * r).set`

Changing coordinates: relations, predicates, sets, functions, and sequences

As we have seen, there are several ways to view a set or a relation. Which one is best depends on what you want to do with it, and what is familiar and comfortable in your application. Often the choice of representation makes a big difference to the convenience and clarity of your code, just as the choice of coordinate system makes a big difference in a physics problem. The following tables summarize the different representations, the methods they have, and the conversions among them. The players are sets, functions, predicates, and relations.

Method	*Converts*	*to*	*by*	*Inverse*
`.rel`	`F=T->U`	`T->>U`	`(\t,u|f!t/\f(t)=u).pToR`	`.fun`
	`S=SET T`	`Bool->>T`	`s.pred.inv.restrict({true})`	`.set`
`.pred`	`S=SET T`	`T->Bool`	definition; `(\t | t IN s)`	`.set`
	`R=T->>U`	`(T,U)->Bool`	definition; `(\t,u | u IN r.setF(r))`	`.pToR`
`.set`	`F=T->Bool`	`SET T`	`f.restrict({true}).rng`	`.rel`
	`R=Bool->>T`	`SET T`	`r.rng`	`.rel`
`.fun`	`R=T->>U`	`T->U`	`r.setF.one`	`.rel`
`.pToR`	`F=(T,U)->Bool`	`T->>U`	definition; `(\t | {u|f(t,u)}.setRel`	`.pred`
`.setF`	`R=T->>U`	`T->SET U`	`(\t | {t} * r)`	`.setRel`
`.setRel`	`F=T->SET U`	`T->>U`	`f.rel.include`	`.setF`
`.paths`	`T->>T`	`SET SEQ T`	see above	`.pRel`
`.pRel`	`SEQ T`	`T->>T`	`{i :IN q.dom - {0} || (q(i-1), q(i))}.pred.pToR`	`.paths` sort of



Here is another way to look at it. Each of the types that label rows and columns in the following tables is equivalent to the others, and the entries in the table tell how to convert from one form to another.

from \ **to**		*set* SET T	*predicate* T->Bool	*relation* Bool->>T
set	SET T		.pred	.rel
predicate	T->Bool	.set		.inv
relation	Bool->>T	.set	.inv	

from \ **to**		*relation* T->>U	*predicate* (T,U)->Bool	*set function* T->SET U	*set of pairs* SET (T,U)
relation	T->>U		.pred	.setF	.pred.set
predicate	(T,U)->Bool	.pToR		.pToR.setF	.set
set function	T->SET U	.setRel	.setRel.pred		.setRel.pred.set
set of pairs	SET (T,U)	.pred.pToR	.pred	.pred.pToR.setF	

Sequences

A function is called a sequence if its domain is a finite set of consecutive `Int`'s starting at 0, that is, if it has type

```
Q = Int->T SUCHTHAT q.dom = {i: Int | 0<=i /\ i<q.dom.max })
```

We denote this type (with the methods defined below) by `SEQ T`. A sequence inherits the methods of the function (though it overrides `+`), and it also has methods for

detaching or attaching the first or last element,
extracting a segment of a sequence, concatenating two sequences, or finding the size,

making a sequence with all elements the same: `t.Fill(n)`,
testing for prefix or sub-sequence (not necessarily contiguous): `q1<=q2`, `q1<<-q2`,
lexical comparison, permuting, and sorting,
filtering, iterating over, and combining the elements,
making a sequence into a relation that holds exactly between successive elements,
treating a sequence as a multiset with operations to:
count the number of times an element appears: `q.count(t)`,
test membership: `t IN q`,
take differences: `q1 - q2`
("+" is union and `addl` adds an element; to remove an element use `q - {t}`; to test equality use `q1 IN q2.perms`).

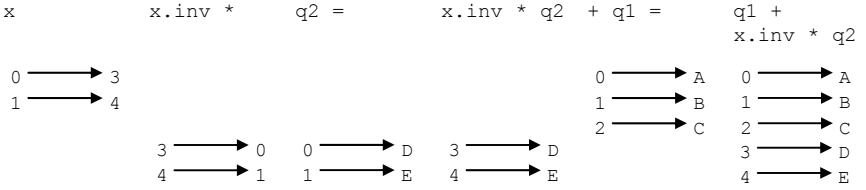
All these operations are undefined if they use out-of-range subscripts, except that a sub-sequence is always defined regardless of the subscripts, by taking the largest number of elements allowed by the size of the sequence.

The value of `i .. j` is the sequence of integers from `i` to `j`.

To apply a function `f` to each of the elements of `q`, just use composition `q * f`.

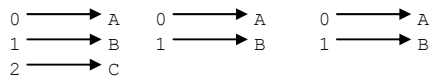
The "+" operator concatenates two sequences.

`q1 + q2 = q1 + x.inv * q2`, where `x = (q1.size .. q1.size+q2.size-1)`
`q1 = {A,B,C}; q2 = {D,E}; x = {3,4}; q1 + q2 = {A,B,C,D,E}`

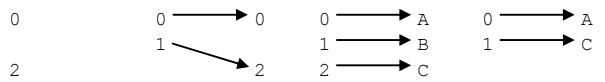


You can test for `q1` being a prefix of `q1` with `q1 <= q2`, and for it being an arbitrary subsequence, not necessarily contiguous, with `q1 <<= q2`.

`q1 <= q2 = (q1 = q2.restrict(q1.dom))`
`q1 = {A,B}; q2 = {A,B,C}`



`q1 <<= q2 = (EXISTS s: SET Int | s <= q2.dom /\ q1 = s.sort * q2`
`q1 = {A,C}; q2 = {A,B,C}`; choose `s = {0,2} <= {0,1,2}`



You can take a subsequence of size `n` starting at `i` with `q.seg(i,n)` and a subsequence from `i1` to `i2` with `q.sub(i1,i2)`.

```
q.seg(i,n) = (i .. i+n-1) * q
q = {A,B,C}; i = 1; n = 3; q.seg(1,3) = {B,C}
```

```
i .. i+n-1   * q          =q.seg(i,n)

 0 ──▶ 1    0 ──▶ A      0 ──▶ B
 1 ──▶ 2    1 ──▶ B      1 ──▶ C
 2 ──▶ 3    2 ──▶ C
```

You can select the elements of `q` that satisfy a predicate `f` with `q.filter(f)`.

```
q.filter(f) = (q * f).set.sort * q
q = {5,4,3,2,1}; f = even
```

```
q            q * f          (q * f).set   .sort          * q =

 0 ──▶ 5     0 ──▶ false                  0 ──▶ 1      0 ──▶ 4
 1 ──▶ 4     1 ──▶ true     1            1 ──▶ 3      1 ──▶ 2
 2 ──▶ 3     2 ──▶ false
 3 ──▶ 2     3 ──▶ true     3
 4 ──▶ 1     4 ──▶ false
```

You can zip up a pair of sequences to get a sequence of pairs with `q1 || q2`. Then you can compose a binary function to get the sequence of results

```
q1={1,2,3,4,5}   q2={6,7,8,9,10}   q1 || q2          (q1 || q2) * Int."+"

 0 ──▶ 1          0 ──▶ 6          0 ──▶ (1,6)      0 ──▶ 7
 1 ──▶ 2          1 ──▶ 7          1 ──▶ (2,7)      1 ──▶ 9
 2 ──▶ 3          2 ──▶ 8          2 ──▶ (3,8)      2 ──▶ 11
 3 ──▶ 4          3 ──▶ 9          3 ──▶ (4,9)      3 ──▶ 13
 4 ──▶ 5          4 ──▶ 10         4 ──▶ (5,10)     4 ──▶ 15
```

Since a pair of `SEQ T` is a function `0..1 -> 0..n -> T` and `SEQ (T, T)` is a function `0..n -> 0..1 -> T`, `zip` just reverses the order of the arguments.

You can apply a combining function `f` successively to the elements of `q` with `q.iterate(f)`. To get the result of combining all the elements of `q` with `f` use `q.combine(f) = q.iterate(f).last`. The syntax `+ : q` is short for `q.combine(T."+")`; it works for any binary operator that yields a `T`.

```
q = {1,2,3,4,5}   q.iterate(Int."+")

 0 ──▶ 1          0 ──▶ 1
 1 ──▶ 2          1 ──▶ 3
 2 ──▶ 3          2 ──▶ 6
 3 ──▶ 4          3 ──▶ 10
 4 ──▶ 5          4 ──▶ 15
```


Method call	*Result type*	*Definition*
`q1 + q2`	`Q`	`q1 + (q1.size .. q1.size+q2.size-1).inv * q2`
`q1 <= q2`	`Bool`	`q1 = q2.restrict(q1.dom)`
`q1 <<= q2`	`Bool`	`(EXISTS s: SET Int | s <= q2.dom /\ q1 = s.sort * q2)`
`q.size`	`Nat`	`q.dom.size`
`q.seg(i,n)`	`Q`	`(i .. i+n-1) * q`
`q.sub(i1,i2)`	`Q`	`(i1 .. i2) * q`
`q.head`	`T`	`q(0)`
`q.tail`	`Q`	`(q # {} => q.sub(1, q.size-1))`
`t.fill(n)`	`Q`	`(0 .. n-1) * {* -> t}`
`q1.lexLE(q2,f)`	`Bool`	`(EXISTS q,n | n=q.size /\ q<=q1 /\ q<=q2 /\ (q=q1 \/ f(q1(n),q2(n)) /\ q1(n)#q2(n)))`
`q.filter(f)`	`Q`	`(q * f).set.sort * q`, where `f: T->Bool`
`q || qU`	`SEQ(T,U)`	`RET (\ i | (i IN (q.dom /\ qU.dom) => (q(i), qU(i))))` where `qU: SEQ U`
`q.iterate(f)`	`Q`	`{qr | qr.size=q.size /\ qr(0)=q(0) /\ (ALL i IN q.dom-{0}|qr(i)=f(qr(i-1),q(i)))}.one` where `f: (T,T)->T`
`q.combine(f)`	`T`	`q.iterate.last`
`t ** n`	`T`	`t.fill(n).combine(T."*")`
`q.pRel`	`T->>T`	`{i :IN q.dom - {0} || (q(i-1), q(i))}.pred.pToR`
`q.count(t)`	`Nat`	`{t' :IN q | t' = t}.size`
`t IN q`	`Bool`	`t IN q.rng`
`q1 - q2`	`Q`	`{q |(ALL t | q.count(t)={q1.count(t)-q2.count(t), 0}.max)}.choose`

`SEQ T` has the same `perms`, `fsort`, `sort`, `fmax`, `fmin`, `max`, and `min` constructors as `SET T`.

Records and tuples

Sets, functions, and sequences are good when you have many values of the same type. When you have values of different types, you need a tuple or a record (they are the same, except that a record allows you to name the different values). In Spec a record is a function from the string names of its fields to the field values, and an *n*-tuple is a function from `0..n-1` to the field values. There is special syntax for declaring records and tuples, and for reading and writing record fields:

`[f: T, g: U]` declares a record with fields `f` and `g` of types `T` and `U`.

`(T, U)` declares a tuple with fields of types `T` and `U`.

`r.f` is short for `r("f")`, and `r.f := e` is short for `r := r{"f"->e}`.

There is also special syntax for constructing record and tuple values, illustrated in the following example. Given the type declaration

```
TYPE Entry = [salary: Int, birthdate: String]
```

we can write a record value

```
Entry{salary := 23000, birthdate := "January 3, 1955"}
```

which is short for the function constructor

```
Entry{"salary" -> 23000, "birthdate" -> "January 3, 1955"}.
```

The constructor `(`

```
23000, "January 3, 1955")
```

yields a tuple of type `(Int, String)`. It is short for

```
{0 -> 23000, 1 -> "January 3, 1955"}
```

This doesn't work for a singleton tuple, since `(x)` has the same value as `x`. However, the sequence constructor `{x}` will do for constructing a singleton tuple, since a singleton `SEQ T` is also a singleton tuple; in fact, this is the only way to write the type of a singleton tuple, since `(T)` is the same as `T` because parentheses are used for grouping in types just as they are in ordinary expressions.

The type of a record is `String->Any SUCHTHAT ...`, and the type of a tuple is `Nat->Any SUCHTHAT ...`. Here the `SUCHTHAT` clauses are of the form `this("f") IS T`; they specify the types of the fields. In addition, a record type has a method called `fields` whose value is the sequence of field names (it's the same for every record). Thus `[f: T, g: U]` is short for

```
String->Any WITH { fields:=(\r: String->Any | (SEQ String){"f", "g"}) }
            SUCHTHAT   this.dom >= {"f", "g"}
                    /\ this("f") IS T /\ this("g") IS U
```

A tuple type works the same way; its `fields` is just `0..n-1` if the tuple has `n` fields. Thus `(T, U)` is short for

```
Int->Any WITH { fields:=(\r: Int->Any | 0..1) }
            SUCHTHAT   this.dom = 0..1
                    /\ this(0) IS T /\ this(1) IS U
```

Thus to convert a record `r` into a tuple, write `r.fields * r`, and to convert a tuple `t` into a record, write `r.fields.inv * t`.

There is no special syntax for tuple fields, since you can just write `t(2)` and `t(2) := e` to read and write the third field, for example (remember that fields are numbered from 0).

Functions declared with more than one argument are a bit tricky: they take a single argument that is a tuple. So `f(x: Int)` takes an `Int`, but `f(x: Int, y: Int)` takes a tuple of type `(Int, Int)`. This convention keeps the tuples in the background as much as possible. The normal syntax for calling a function is `f(x, y)`, which constructs the tuple `(x, y)` and passes it to `f`. However, `f(x)` is treated differently, since it passes `x` to `f`, rather than the singleton tuple `{x}`. If you have a tuple `t` in hand, you can pass it to `f` by writing `f$t` without having to worry about the singleton case; if `f` takes only one argument, then `t` must be a singleton tuple and `f$t` will pass `t(0)` to `f`. Thus `f$(x, y)` is the same as `f(x, y)` and `f${x}` is the same as `f(x)`.

A function declared with names for the arguments, such as

```
(\ i: Int, s: String | i + StringToInt(x))
```

has a type that ignores the names, `(Int, String)->Int`. However, it also has a method `argNames` that returns the sequence of argument names, `{"i", "s"}` in the example, just like a record. This makes it possible to match up arguments by name, as in the following example.

A database is a set `s` of records. A selection query `q` is a predicate that we want to apply to the records. How do we get from the field names, which are strings, to the argument for `q`? Assume that `q` has an `argNames` method. So if `r IN s`, `q.argNames * r` is the tuple that we want to feed to `q`; `q$(q.argNames * r)` is the query, where `$` is the operator that applies a function to a tuple of its arguments.

There is one problem if not all fields are defined in all records: when we try to use `q.argNames * r`, it will be undefined if `r` doesn't have all the fields that `q` wants. We want to apply it only to the records in `s` that have all the necessary fields. That is the set

```
{r :IN s | q.argNames <= r.fields}
```

The answer we want is the subset of records in this set for which `q` is true. That is

```
{r :IN s | q.argNames <= r.fields /\ q$(q.argNames * r)}
```


To project the database, discarding all the fields except the ones in `projection` (a set of strings), write

```
{r :IN s || r.restrict(projection)}
```

Constructors

Functions, sets, and sequences make it easy to toss large values around, and constructors are special syntax to make it easier to define these values. For instance, you can describe a database as a function `db` from names to data records with two fields:

```
TYPE DB = (String -> Entry)
TYPE Entry = [salary: Int, birthdate: Int]
VAR db := DB{}
```

Here `db` is initialized using a function constructor whose value is a function undefined everywhere. The type can be omitted in a variable declaration when the variable is initialized; it is taken to be the type of the initializing expression. The type can also be omitted when it is the upper case version of the variable name, `DB` in this example.

Now you can make an entry with

```
db := db{ "Smith" -> Entry{salary := 23000, birthdate := 1955} }
```

using another function constructor. The value of the constructor is a function that is the same as `db` except at the argument `"Smith"`, where it has the value `Entry{...}`, which is a record constructor. This assignment could also be written

```
db("Smith") := Entry{salary := 23000, birthdate := 1955}
```

which changes the value of the `db` function at `"Smith"` without changing it anywhere else. This is actually a shorthand for the previous assignment. You can omit the field names if you like, so that

```
db("Smith") := Entry{23000, 1955}
```

has the same meaning as the previous assignment. Obviously this shorthand is less readable and more error-prone, so use it with discretion. Another way to write this assignment is

```
db("Smith").salary := 23000; db("Smith").birthdate := 1955
```

A record is actually a function as well, from the strings that represent the field names to the field values. Thus `Entry{salary := 23000, birthdate := 1955}` is a function `r: String->Any` defined at two string values, `"salary"` and `"birthdate"`: `r("salary") = 23000` and `r("birthdate") = 1955`. We could have written it as a function constructor `Entry{"salary" -> 23000, "birthdate" -> 1955}`, and `r.salary` is just a convenient way of writing `r("salary")`.

The set of names in the database can be expressed by a set constructor. It is just

```
{n: String | db!n},
```

in other words, the set of all the strings for which the `db` function is defined ('`!`' is the 'is-defined' operator; that is, `f!x` is true iff `f` is defined at `x`). Read this "the set of strings `n` such that `db!n`". You can also write it as `db.dom`, the domain of `db`; section 9 of the reference manual defines lots of useful built in methods for functions, sets, and sequences. It's important to realize that you can freely use large (possibly infinite) values such as the `db` function. You are writing a spec, and you don't need to worry about whether the compiler is clever enough to turn an expensive-looking manipulation of a large object into a cheap incremental update. That's the implementer's problem (so you may have to worry about whether *she* is clever enough).

If we wanted the set of lengths of the names, we would write

```
{n: String | db!n || n.size}
```

This three part set constructor contains `i` if and only if there exists an `n` such that `db!n` and `i = n.size`. So `{n: String | db!n}` is short for `{n: String | db!n || n}`. You can introduce more than one name, in which case the third part defaults to the last name. For example, if we represent a directed graph by a function on pairs of nodes that returns `true` when there's an edge from the first to the second, then

```
{n1: Node, n2: Node | graph(n1, n2) || n2}
```

is the set of nodes that are the target of an edge, and the "`|| n2`" could be omitted. This is just the range `graph.rng` of the relation `graph` on nodes.

Following standard mathematical notation, you can also write

```
{f :IN openFiles | f.modified}
```

to get the set of all open, modified files. This is equivalent to

```
{f: File | f IN openFiles /\ f.modified}
```

because if `s` is a `SET T`, then `IN s` is a type whose values are the `T`'s in `s`; in fact, it's the type `T SUCHTHAT (\ t | t IN s)`. This form also works for sequences, where the second operand of `:IN` provides the ordering. So if `s` is a sequence of integers, `{x :IN s | x > 0}` is the positive ones, `{x :IN s | x > 0 || x * x}` is the squares of the positive ones, and `{x :IN s || x * x}` is the squares of all the integers, because an omitted predicate defaults to `true`.[5]

To get sequences that are more complicated you can use sequence generators with `BY` and `WHILE`. You can skip this paragraph until you need to do this.

```
{i := 1 BY i + 1 WHILE i <= 5 | true || i}
```

is `{1, 2, 3, 4, 5}`; the second and third parts could be omitted. This is just like the "for" construction in C. An omitted `WHILE` defaults to `true`, and an omitted `:=` defaults to an arbitrary choice for the initial value. If you write several generators, each variable gets a new value for each value produced, but the second and later variables are initialized first. So to get the sums of successive pairs of elements of `s`, write

```
{x := s BY x.tail WHILE x.size > 1 || x(0) + x(1)}
```

To get the sequence of partial sums of `s`, write (eliding `|| sum` at the end)

```
{x :IN s, sum := 0 BY sum + x}
```

Taking `last` of this would give the sum of the elements of `s`. To get a sequence whose elements are reversed from those of `s`, write

```
{x :IN s, rev := {} BY {x} + rev}.last
```

To get the sequence `{e, f(e), f^2(e), ..., f^n(e)}`, write

```
{i :IN 1 .. n, iter := e BY f(iter)}
```

Combinations

A combination is a way to combine the elements of a non-empty sequence or set into a single value using an infix operator, which must be associative, and must be commutative if it is applied to a set. You write "operator `:` sequence or set". This is short for `q.combine(T.operator)`. Thus

```
+ : (SEQ String){"He", "l", "lo"} = "He" + "l" + "lo" = "Hello"
```

because `+` on sequences is concatenation, and

```
+ : {i :IN 1 .. 4 || i**2} = 1 + 4 + 9 + 16 = 30
```

Existential and universal quantifiers make it easy to describe properties without explaining how to test for them in a practical way. For instance, a predicate that is `true` iff the sequence `s` is sorted is

```
(ALL i :IN 1 .. s.size-1   | s(i-1) <= s(i))
```

This is a common idiom; read it as

[5] In the sequence form, `IN s` is not a set type but a special construct; treating it as a set type would throw away the essential ordering information.

"for all `i` in `1 .. s.size-1`, `s(i-1) <= s(i)`".

This could also be written

```
(ALL i :IN (s.dom - {0}) | s(i-1) <= s(i))
```

since `s.dom` is the domain of the function `s`, which is the non-negative integers `< s.size`. Or it could be written

```
(ALL i :IN s.dom | i > 0 ==> s(i-1) <= s(i))
```

Because a universal quantification is just the conjunction of its predicate for all the values of the bound variables, it is simply a combination using `/\` as the operator:

```
(ALL    i | Predicate(i)) = /\ : {i | Predicate(i)}
```

Similarly, an existential quantification is just a similar disjunction, hence a combination using `\/` as the operator:

```
(EXISTS i | Predicate(i)) = \/ : {i | Predicate(i)}
```

Spec has the redundant `ALL` and `EXISTS` notations because they are familiar.

If you want to get your hands on a value that satisfies an existential quantifier, you can construct the set of such values and use the `choose` method to pick out one of them:

```
{i | Predicate(i)}.choose
```

The `VAR` command described in the next section on commands is another form of existential quantification that lets you get your hands on the value, but it is non-deterministic.

Commands

Commands are for changing the state. Spec has a few simple commands, and seven operators for combining commands into bigger ones. The main simple commands are assignment and routine invocation. There are also simple commands to raise an exception, to return a function result, and to `SKIP`, that is, do nothing. If a simple command evaluates an undefined expression, it fails (see below).

You can write `i + := 3` instead of `i := i + 3`, and similarly with any other binary operator.

The operators on commands are:

- A conditional operator: `predicate => command`, read "if `predicate` then `command`". The predicate is called a *guard*.
- Choice operators: `c1 [] c2` and `c1 [*] c2`, read 'or' and 'else'.
- Sequencing operators: `c1 ; c2` and `c1 EXCEPT handler`. The `handler` is a special form of conditional command: `exception => command`.
- Variable introduction: `VAR id: T | command`, read "choose `id` of type `T` such that `command` doesn't fail".
- Loops: `DO command OD`.

Section 6 of the reference manual describes commands. *Atomic Semantics of Spec* gives a precise account of their semantics. It explains that the meaning of a command is a *relation* between a state and an outcome (a state plus an optional exception), that is, a set of possible state-to-outcome transitions.

Conditionals and choice

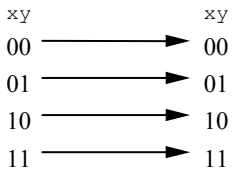
Combining commands

The figure below (copied from Nelson's paper) illustrates conditionals and choice with some very simple examples. Here is how they work:

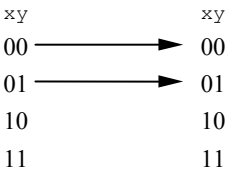
The command

```
p => c
```

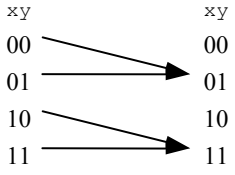
means to do `c` if `p` is true. If `p` is false this command fails; in other words, it has no outcome. More precisely, if `s` is a state in which `p` is false or undefined, this command does not relate `s` to any outcome.



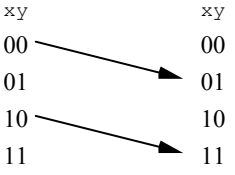
SKIP



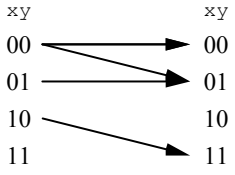

```
x = 0 => SKIP
```
(partial)



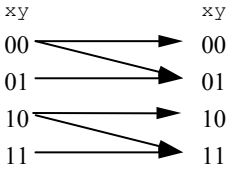

```
y := 1
```




```
y = 0 => y := 1
```
(partial)




```
  x = 0 => SKIP
[] y = 0 => y := 1
```
(partial, non-deterministic)




```
  SKIP
[] y = 0 => y := 1
```
(non-deterministic)

What good is such a command? One possibility is that `p` will be true some time in the future, and then the command will have an outcome and allow a transition. Of course this can only happen in a concurrent program, where there is something else going on that can make `p` true. Even if there's no concurrency, there might be an alternative to this command. For instance, it might appear in the larger command


```
    p  => c
[]  p' => c'
```

in which you read `[]` as 'or'. This fails only if each of `p` and `p'` is false or undefined. If both are true (as in the `00` state in the south-west corner of the figure), it means to do either `c` or `c'`; the choice is non-deterministic. If `p'` is `~p` then they are never both false, and if `p` is defined this command is equivalent to

```
    p => c
[*] c'
```

in which you read `[*]` as 'else'. On the other hand, if `p` is undefined the two commands differ, because the first one fails (since neither guard can be evaluated), while the second does `c'`.

Both `c1 [] c2` and `c1 [*] c2` fail only if *both* `c1` and `c2` fail. If you think of a Spec program operationally (that is, as executing one command after another), this means that if the execution makes some choice that leads to failure later on, it must 'back-track' and try the other alternatives until it finds a set of choices that succeed. For instance, no matter what `x` is, after

```
    y = 0 => x := x - 1; x < y => x := 1
[]  y > 0 => x := 3    ; x < y => x := 2
[*] SKIP
```

if `y = 0` initially, `x = 1` afterwards, if `y > 3` initially, `x = 2` afterwards, and otherwise `x` is unchanged. If you think of it relationally, `c1 [] c2` has all the transitions of `c1` (there are none if `c1` fails, several if it is non-deterministic) as well as all the transitions of `c2`. Both failure and non-determinism can arise from deep inside a complex command, not just from a top-level `[]` or `VAR`.

This is sometimes called 'angelic' non-determinism, since the code finds all the possible transitions, yielding an outcome if *any* possible non-deterministic choice yield that outcome. This is usually what you want for a spec or high-level code; it is not so good for low-level code, since an operational implementation requires backtracking. The other kind of non-determinism is called 'demonic'; it yields an outcome only if *all* possible non-deterministic choice yield that outcome. To do a command `C` and check that all outcomes satisfy some predicate `p`, write `<< C; ~ p => abort >> [*] C`. The command before the `[*]` does `abort` if some outcome does not satisfy `p`; if every outcome satisfies `p` it fails (doing nothing), and the else clause does `C`.

The precedence rules for commands are

`EXCEPT`	binds tightest
`;`	next
`=> |`	next (for the right operand; the left side is an expression or delimited by `VAR`)
`[] [*]`	bind least tightly.

These rules minimize the need for parentheses, which are written around commands in the ugly form `BEGIN ... END` or the slightly prettier form `IF ... FI`; the two forms have the same meaning, but as a matter of style, the latter should only be used around guarded commands. So, for example,

```
p => c1; c2
```

is the same as

```
p => BEGIN c1; c2 END
```

and means to do `c1` followed by `c2` if `p` is true. To guard only `c1` with `p` you must write

```
IF p => c1 [*] SKIP FI; c2
```

which means to do `c1` if `p` is true, and then to do `c2`. The `[*] SKIP` ensures that the command before the `";"` does not fail, which would prevent `c2` from getting done. Without the `[*] SKIP`, that is in

```
IF p => c1 FI; c2
```

if p is false the IF ... FI fails, so there is no possible outcome from which c2 can be done and the whole thing fails. Thus IF p => c1 FI; c2 has the same meaning as p => BEGIN c1; c2 END, which is a bit surprising.

Sequencing

A c1 ; c2 command means just what you think it does: first c1, then c2. The command c1 ; c2 gets you from state s1 to state s2 if there is an intermediate state s such that c1 gets you from s1 to s and c2 gets you from s to s2. In other words, its relation is the composition of the relations for c1 and c2; sometimes ';' is called 'sequential composition'. If c1 produces an exception, the composite command ignores c2 and produces that exception.

A c1 EXCEPT ex => c2 command is just like c1 ; c2 except that it treats the exception ex the other way around: if c1 produces the exception ex then it goes on to c2, but if c1 produces a normal outcome (or any other exception), the composite command ignores c2 and produces that outcome.

Variable introduction

VAR gives you more dramatic non-determinism than []. The most common use is in the idiom

```
VAR x: T | P(x) => c
```

which is read "choose some x of type T such that P(x), and do c". It fails if there is no x for which P(x) is true and c succeeds. If you just write

```
VAR x: T | c
```

then VAR acts like ordinary variable declaration, giving an arbitrary initial value to x.

Variable introduction is an alternative to existential quantification that lets you get your hands on the bound variable. For instance, you can write

```
IF  VAR n: Nat, x: Nat, y: Nat, z: Nat |
    (n > 2 /\ x**n + y**n = z**n) => out := n
[*] out := 0
FI
```

which is read: choose integers n, x, y, z such that n > 2 and $x^n + y^n = z^n$, and assign n to out; if there are no such integers, assign 0 to out.[6] The command before the [*] succeeds iff

```
(EXISTS n: Int, x: Int, y: Int, z: Int | n > 2 /\ x**n + y**n = z**n),
```

but if we wrote that in a guard there would be no way to set out to one of the n's that exist. We could also write

```
VAR s := { n: Int, x: Int, y: Int, z: Int
         | n > 2 /\ x**n + y**n = z**n
        || (n, x, y, z)}
```

to construct the set of all solutions to the equation. Then if s # {}, s.choose yields a tuple (n, x, y, z) with the desired property.

You can use VAR to describe all the transitions to a state that has an arbitrary relation R to the current state: VAR s' | R(s, s') => s := s' if there is only one state variable s.

The precedence of | is higher than [], which means that you can string together different VAR commands with [] or [*], but if you want several alternatives within a VAR you have to use BEGIN ... END or IF ... FI. Thus

[6] A correctness proof for an implementation of this spec defied the best efforts of mathematicians between Fermat's time and 1993.


```
    VAR x: T | P(x) => c1
[]  q => c2
```

is parsed the way it is indented and is the same as

```
    BEGIN VAR x: T | P(x) => c1 END
[]  BEGIN q => c2 END
```

but you must write the brackets in

```
VAR x: T |
    IF P(x) => c1
    [] Q(x) => c2
    FI
```

which might be formatted more concisely as

```
VAR x: T |
    IF  P(x) => c1
    []  R(x) => c2 FI
```

or even

```
VAR x: T | IF P(x) => c1 [] R(x) => c2 FI
```

You are supposed to indent your programs to make it clear how they are parsed.

Loops

You can always write a recursive routine, but sometimes a loop is clearer . In Spec you use DO ... OD for this. These are brackets, and the command inside is repeated as long as it succeeds. When it fails, the repetition is over and the DO ... OD is complete. The most common form is

```
DO P => C OD
```

which is read "while P is true do C". After this command, P must be false. If the command inside the DO ... OD succeeds forever, the outcome is a looping exception that cannot be handled. Note that this is not the same as a failure, which means no outcome at all.

For example, you can zero all the elements of a sequence s with

```
VAR i := 0 | DO i < s.size => s(i) := 0; i - := 1 OD
```

or the simpler form (which also avoids fixing the order of the assignments)

```
DO VAR i | s(i) # 0 => s(i) := 0 OD
```

This is another common idiom: keep choosing an i as long as you can find one that satisfies some predicate. Since s is only defined for i between 0 and s.size-1, the guarded command fails for any other choice of i. The loop terminates, since the s(i) := 0 definitely reduces the number of i's for which the guard is true. But although this is a good example of a loop, it is bad style; you should have used a sequence method or function composition:

```
s := 0.fill(s.size)
```

or

```
s := {x :IN s || 0}
```

(a sequence just like s except that every element is mapped to 0), remembering that Spec makes it easy to throw around big things. Don't write a loop when a constructor will do, because the loop is more complicated to think about. Even if you are writing code, you still shouldn't use a loop here, because it's quite clear how to write C code for the constructor.

To zero all the elements of s that satisfy some predicate P you can write

```
DO VAR i: Int | (s(i) # 0 /\ P(s(i))) => s(i) := 0 OD
```

Again, you can avoid the loop by using a sequence constructor and a conditional expression

```
s := {x :IN s || (P(x) => 0 [*] x) }
```

Atomicity

Each <<...>> command is atomic. It defines a single transition, which includes moving the program counter (which is part of the state) from before to after the command. If a command is not inside <<...>>, it is atomic only if there's no reasonable way to split it up: SKIP, HAVOC, RET, RAISE. Here are the reasonable ways to split up the other commands:

- An assignment has one internal program counter value, between evaluating the right hand side expression and changing the left hand side variable.
- A guarded command likewise has one, between evaluating the predicate and the rest of the command.
- An invocation has one after evaluating the arguments and before the body of the routine, and another after the body of the routine and before the next transition of the invoking command.

Note that evaluating an expression is always atomic.

Modules and names

Spec's modules are very conventional. Mostly they are for organizing the name space of a large program into a two-level hierarchy: module.id. It's good practice to declare everything except a few names of global significance inside a module. You can also declare CONST's, just like VAR's.

```
MODULE foo EXPORT i, j, Fact =

CONST c := 1

VAR i := 0
    j := 1

FUNC Fact(n: Int) -> Int =
     IF   n <= 1 => RET 1
     [*]  RET n * Fact(n - 1)
     FI

END foo
```

You can declare an identifier id outside of a module, in which case you can refer to it as id everywhere; this is short for Global.id, so Global behaves much like an extra module. If you declare id at the top level in module m, id is short for m.id inside of m. If you include it in m's EXPORT clause, you can refer to it as m.id everywhere. All these names are in the *global* state and are shared among all the atomic actions of the program. By contrast, names introduced by a declaration inside a routine are in the *local* state and are accessible only within their scope.

The purpose of the EXPORT clause is to define the external interface of a module. This is important because module T implements module S iff T's behavior at its external interface is a subset of S's behavior at its external interface.

The other feature of modules is that they can be parameterized by types in the same style as CLU clusters. The memory systems modules in handout 5 are examples of this.

You can also declare a *class*, which is a module that can be instantiated many times. The Obj class produces a global Obj type that has as its methods the exported names of the class plus a new procedure that returns a new, initialized instance of the class. It also produces a ObjMod

module that contains the declaration of the Obj type, the code for the methods, and a state variable indexed by Obj that holds the state records of the objects. In a method you can refer to the current object instance by self. For example:

```
CLASS Stat EXPORT add, mean, variance, reset =

VAR n           :  Int := 0
    sum         :  Int := 0
    sumsq       :  Int := 0

PROC add(i: Int) = n + := 1; sum + := i; sumsq + := i**2
FUNC mean() -> Int = RET sum/n
FUNC variance() -> Int = RET sumsq/n - self.mean**2
PROC reset() = n := 0; sum := 0; sumsq := 0

END Stat
```

Then you can write

```
    VAR s: Stat | s := s.new(); s.add(x); s.add(y); Print(s.variance)
```

In abstraction functions and invariants we also write obj.n for field n in obj's state.

Section 7 of the reference manual deals with modules. Section 8 summarizes all the uses of names and the scope rules. Section 9 gives several modules used to define the methods of the built-in data types such as functions, sets, and sequences.

This completes the language summary; for more details and greater precision consult the reference manual. The rest of this handout consists of three extended examples of specs and code written in Spec: topological sort, editor buffers, and a simple window system.

Example: Topological sort

Suppose we have a directed graph whose n+1 vertexes are labeled by the integers 0 .. n, represented in the standard way by a relation g; g(v1, v2) is true if v2 is a successor of v1, that is, if there is an edge from v1 to v2. We want a topological sort of the vertexes, that is, a sequence that is a permutation of 0 .. n in which v2 follows v1 whenever v2 is a successor of v1 in the relation g. Of course this possible only if the graph is acyclic.

```
MODULE TopologicalSort EXPORT V, G, Q, TopSort =

TYPE V = IN 0 .. n                                          % Vertex
     G = (V, V) -> Bool                                     % Graph
     Q = SEQ V

PROC TopSort(g) -> Q RAISES {cyclic} =
     IF   VAR q | q IN (0 .. n).perms /\ IsTSorted(q, g) => RET q
     [*]  RAISE cyclic                                      % g must be cyclic
     FI

FUNC IsTSorted(q, g) -> Bool =
% Not tsorted if v2 precedes v1 in q but is also a child
     RET ~ (EXISTS v1 :IN q.dom, v2 :IN q.dom | v2 < v1 /\ g(q(v1), q(v2))

END TopologicalSort
```

Note that this solution checks for a cyclic graph. It allows any topologically sorted result that is a permutation of the vertexes, because the VAR q in TopSort allows any q that satisfies the two

conditions. The perms method on sets and sequences is defined in section 9 of the reference manual; the dom method gives the domain of a function. TopSort is a procedure, not a function, because its result is non-deterministic; we discussed this point earlier when studying SquareRoot. Like that one, this spec has no internal state, since the module has no VAR. It doesn't need one, because it does all its work on the input argument.

The following code is from Cormen, Leiserson, and Rivest. It adds vertexes to the front of the output sequence as depth-first search returns from visiting them. Thus, a child is added before its parents and therefore appears after them in the result. Unvisited vertexes are white, nodes being visited are grey, and fully visited nodes are black. Note that all the descendants of a black node must be black. The grey state is used to detect cycles: visiting a grey node means that there is a cycle containing that node.

This module has state, but you can see that it's just for convenience in programming, since it is reset each time TopSort is called.

```
MODULE TopSortImpl EXPORT V, G, Q, TopSort =        % implements TopSort

TYPE Color = ENUM[white, grey, black]               % plus the spec's types

VAR  out  : Q
     color: V -> Color                              % every vertex starts white

PROC TopSort(g) -> Q RAISES {cyclic} = VAR i := 0 |
     out := {}; color := {* -> white}
     DO VAR v | color(v) = white => Visit(v, g) OD; % visit every unvisited vertex
     RET out

PROC Visit(v, g) RAISES {cyclic} =
     color(v) := grey;
     DO VAR v' | g(v, v') /\ color(v') # black =>   % pick an successor not done
          IF   color(v') = white => Visit(v', g)
          [*]  RAISE cyclic                         % grey — partly visited
          FI
     OD;
     color(v) := black; out := {v} + out            % add v to front of out
```

The code is as non-deterministic as the spec: depending on the order in which TopSort chooses v and Visit chooses v', any topologically sorted sequence can result. We could get deterministic code in many ways, for example by using min to take the smallest node in each case:

```
     VAR v  := {v0 | color(v0) = white}.min              in TopSort
     VAR v' := {v0 | g(v, v0) /\ color(v') # black }.min in Visit
```

Code in C would do something like this; the details would depend on the representation of G.

Example: Editor buffers

A text editor usually has a *buffer* abstraction. A buffer is a mutable sequence of C's. To get started, suppose that C = Char and a buffer has two operations,

Get(i) to get character i

Replace to replace a subsequence of the buffer by a subsequence of an argument of type SEQ C, where the subsequences are defined by starting position and size.

We can make this spec precise as a Spec class.


```
CLASS Buffer EXPORT B, C, X, Get, Replace =

TYPE X = Nat                                        % indeX in buffer
     C = Char
     B = SEQ C                                      % Buffer contents

VAR  b : B := {}                                    % Note: initially empty

FUNC Get(x) -> C = RET b(x)                         % Note: defined iff 0<=x<b.size

PROC Replace(from: X, size: X, b': B, from': X, size': X) =
% Note: fails if it touches C's that aren't there.
     VAR b1, b2, b3 | b = b1 + b2 + b3 /\ b1.size = from /\ b2.size = size =>
       b := b1 + b'.seg(from', size') + b3

END Buffer
```

We can implement a buffer as a sorted array of *pieces* called a 'piece table'. Each piece contains a SEQ C, and the whole buffer is the concatenation of all the pieces. We use binary search to find a piece, so the cost of Get is at most logarithmic in the number of pieces. Replace may require inserting a piece in the piece table, so its cost is at most linear in the number of pieces.[7] In particular, neither depends on the number of C's. Also, each Replace increases the size of the array of pieces by at most two.

A piece is a B (in C it would be a pointer to a B) together with the sum of the length of all the previous pieces, that is, the index in Buffer.b of the first C that it represents; the index is there so that the binary search can work. There are internal routines Locate(x), which uses binary search to find the piece containing x, and Split(x), which returns the index of a piece that starts at x, if necessary creating it by splitting an existing piece. Replace calls Split twice to isolate the substring being removed, and then replaces it with a single piece. The time for Replace is linear in pt.size because on the average half of pt is moved when Split or Replace inserts a piece, and in half of pt, p.x is adjusted if size' # size.

```
CLASS BufImpl EXPORT B,C,X, Get, Replace =          % implements Buffer

TYPE                                                % Types as in Buffer, plus
     N   = X                                        % iNdex in piece table
     P   = [b, x]                                   % Piece: x is pos in Buffer.b
     PT  = SEQ P                                    % Piece Table

VAR  pt  := PT{}

ABSTRACTION FUNCTION buffer.b = + : {p :IN pt || p.b}
% buffer.b is the concatenation of the contents of the pieces in pt

INVARIANT (ALL n :IN pt.dom |  pt(n).b # {}
                            /\ pt(n).x = + :{i :IN 0 .. n-1 || pt(i).b.size})
% no pieces are empty, and x is the position of the piece in Buffer.b, as promised.

FUNC Get(x) -> C = VAR p := pt(Locate(x)) | RET p.b(x - p.x)

PROC Replace(from: X, size: X, b': B, from': X, size': X) =
     VAR n1 := Split(from); n2 := Split(from + size),
         new := P{b := b'.seg(from', size'), x := from} |
```

[7] By using a tree of pieces rather than an array, we could make the cost of Replace logarithmic as well, but to keep things simple we won't do that. See FSImpl in handout 7 for more on this point.

```
        pt :=  pt.sub(0, n1 - 1)
             + NonNull(new)
             + pt.sub(n2, pt.size - 1) * AdjustX(size' - size )

PROC Split(x) -> N =
% Make pt(n) start at x, so pt(Split(x)).x = x.  Fails if x > b.size.
% If pt=abcd|efg|hi, then Split(4) is RET 1 and Split(5) is pt:=abcd|e|fg|hi; RET 2
    IF   pt = {} /\ x = 0 => RET 0
    [*] VAR n := Locate(x), p := pt(n), b1, b2 |
        p.b = b1 + b2 /\ p.x + b1.size = x =>
            VAR frag1 := p{b := b1}, frag2 := p{b := b2, x := x} |
                pt :=  pt.sub(0, n - 1)
                     + NonNull(frag1) + NonNull(frag2)
                     + pt.sub(n + 1, pt.size - 1);
                RET (b1 = {} => n [*] n + 1)
    FI

FUNC Locate(x) -> N = VAR n1 := 0, n2 := pt.size - 1 |
% Use binary search to find the piece containing x. Yields 0 if pt={},
% pt.size-1 if pt#{} /\ x>=b.size; never fails.  The loop invariant is
%  pt={} \/ n2 >= n1 /\ pt(n1).x <= x /\ ( x < pt(n2).x \/ x >= pt.last.x )
% The loop terminates because n2 - n1 > 1 ==> n1 < n < n2, so n2 - n1 decreases.
    DO n2 - n1 > 1 =>
        VAR n := (n1 + n2)/2 | IF pt(n).x <= x => n1 := n [*] n2 := n FI
    OD; RET (x < pt(n2).x => n1 [*] n2)

FUNC NonNull(p) -> PT = RET (p.b # {} => PT{p} [*] {})

FUNC AdjustX(dx: Int) -> (P -> P) = RET (\ p | p{x + := dx})

END BufImpl
```

If subsequences were represented by their starting and ending positions, there would be lots of extreme cases to worry about.

Suppose we now want each `C` in the buffer to have not only a character code but also some additional properties, for instance the font, size, underlining, etc; that is, we are changing the definition of `C` to include the new properties. `Get` and `Replace` remain the same. In addition, we need a third exported method `Apply` that applies to each character in a subsequence of the buffer a `map` function `C -> C`. Such a function might make all the `C`'s italic, for example, or increase the font size by 10%.

```
PROC Apply(map: C->C, from: X, size: X) =
    b :=   b.sub(0, from-1)
         +  b.seg(from, size) * map
         +  b.sub(from + size, b.size-1)
```

Here is code for `Apply` that takes time linear in the number of pieces. It works by changing the representation to add a `map` function to each piece, and in `Apply` composing the `map` argument with the `map` of each affected piece. We need a new version of `Get` that applies the proper `map` function, to go with the new representation.

```
TYPE P    = [b, x, map: C->C]                          % x is pos in Buffer.b

ABSTRACTION FUNCTION buffer.b = + :{p :IN pt || p.b * p.map}
% buffer.b is the concatenation of the pieces in p with their map's applied.
% This is the same AF we had before, except for the addition of * p.map.

FUNC Get(x) -> C = VAR p := pt(Locate(x)) | RET p.map(p.b(x - p.x))
```



```
PROC Apply(map: C->C, from: X, size: X) =
  VAR n1 := Split(from), n2 := Split(from + size) |
    pt :=  pt.sub(0 , n1 - 1)
         + pt.sub(n1, n2 - 1) * (\ p | p{map := p.map * map})
         + pt.sub(n2, pt.size - 1)
```

Note that we wrote `Split` so that it keeps the same `map` in both parts of a split piece. We also need to add `map := (\ c | c)` to the constructor for `new` in `Replace`.

This code was used in the Bravo editor for the Alto, the first what-you-see-is-what-you-get editor. It is still used in Microsoft Word.

Example: Windows

A window (the kind on your computer screen, not the kind in your house) is a map from points to colors. There can be lots of windows on the screen; they are ordered, and closer ones block the view of more distant ones. Each window has its own coordinate system; when they are arranged on the screen, an offset says where each window's origin falls in screen coordinates.

```
MODULE Window EXPORT Get, Paint =

TYPE I         = Int
     Coord     = Nat
     Intensity = IN (0 .. 255).rng
     P         = [x: Coord, y: Coord] WITH {"-":=PSub}       % Point
     C         = [r: Intensity, g: Intensity, b: Intensity]  % Color
     W         = P -> C                                      % Window

FUNC PSub(p1, p2) -> P = RET P{x := p1.x - p2.x, y := p1.y - p2.y}
```

The shape of the window is determined by the points where it is defined; obviously it need not be rectangular in this very general system. We have given a point a "-" method that computes the vector distance between two points; we somewhat confusingly represent the vector as a point.

A 'window system' consists of a sequence of `[w, offset: P]` pairs; we call a pair a `V`. The sequence defines the ordering of the windows (windows closer to the top come first in the sequence); it is indexed by 'window number' `WN`. The `offset` gives the screen coordinate of the window's `(0, 0)` point, which we think of as its upper left corner. There are two main operations: `Paint(wn, p, c)` to set the value of `P` in window `wn`, and `Get(p)` to read the value of `p` in the topmost window where it is defined (that is, the first one in the sequence). The idea is that what you see (the result of `Get`) is the result of painting the windows from last to first, offsetting each one by its `offset` component and using the color that is painted later to completely overwrite one painted earlier. Of course real window systems have other operations to change the shape of windows, add, delete, and move them, change their order, and so forth, as well as ways for the window system to suggest that newly exposed parts of windows be repainted, but we won't consider any of these complications.

First we give the spec for a window system initialized with `n` empty windows. It is customary to call the coordinate system used by `Get` the *screen* coordinates. The `v.offset` field gives the screen coordinate that corresponds to `{0, 0}` in `v.w`. The `v.c(p)` method below gives the value of `v`'s window at the point corresponding to `p` after adjusting by `v`'s offset. The state `ws` is just the sequence of `V`'s. For simplicity we initialize them all with the same offset `{10, 5}`, which is not too realistic.

`Get` finds the smallest `WN` that is defined at `p` and uses that window's color at `p`. This corresponds to painting the windows from last (biggest `WN`) to first with opaque paint, which is what we wanted. `Paint` uses window rather than screen coordinates.

The state (the `VAR`) is a single sequence of windows on the screen, called `V`'s..

```
TYPE WN          =  IN 0 .. n-1                                  % Window Number
     V           =  [w, offset: P]                               % window on the screen
                    WITH {c:=(\ v, p | v.w(p - v.offset))}       % C of a screen point p

VAR  ws: SEQ V   := {i :IN 0..n-1 || V{{}, P{10,5}}}             % the Window System

FUNC Get(p) -> C = VAR wn := {wn' | V.c!(ws(wn'), p)}.min | RET ws(wn).c(p)

PROC Paint(wn, p, c) = ws(wn).w(p) := c

END Window
```

Now we give code that only keeps track of the visible color of each point (that is, it just keeps the pixels on the screen, not all the pixels in windows that are covered up by other windows). We only keep enough state to handle `Get` and `Paint`, so in this code windows can't move or get smaller. In a real window system an "expose" event tells a window to deliver the color of points that become newly visible.

The state is one `W` that represents the screen, plus an `exposed` variable that keeps track of which window is exposed at each point, and the offsets of the windows. This is sufficient to implement `Get` and `Paint`; to deal with erasing points from windows we would need to keep more information about what other windows are defined at each point, so that `exposed` would have a type `P -> SET WN`. Alternatively, we could keep track for each window of where it is defined. Real window systems usually do this, and represent `exposed` as a set of visible regions of the various windows. They also usually have a 'background' window that covers the whole screen, so that every point on the screen has some color defined; we have omitted this detail from the spec and the code.

We need a history variable `wH` that contains the `w` part of all the windows. The abstraction function just combines `wH` and `offset` to make `ws`. Note that the abstract state `ws` is a sequence, that is, a function from window number to `V` for the window. The abstraction function gives the value of the `ws` function in terms of the code variables `wH` and `offset`; that is, it is a function from `wH` and `offset` to `ws`. By convention, we don't write this as a function explicitly.

The important properties of the code are contained in the invariant, from which it's clear that `Get` returns the answer specified by `Window.Get`. Another way to do it is to have a history variable `wsH` that is equal to `ws`. This makes the abstraction function very simple, but then we need an invariant that says `offset(wn) = wsH(n).offset`. This is perfectly correct, but it's usually better to put as little stuff in history variables as possible.

`MODULE WinImpl` `EXPORT Get, Paint =`

```
VAR  w       :=  W{}                                 % no points defined
     exposed :   P -> WN := {}                       % which wn shows at p
     offset  :=  {i :IN 0..n-1 || P(5, 10)}          %
     wH      :=  {i :IN 0..n-1 || W{}}               % history variable

ABSTRACTION FUNCTION ws = (\ wn | V{w := wH(wn), offset := offset(wn)})
```



```
INVARIANT
     (ALL p |    w!p = exposed!p
            /\ (w!p ==>    {wn | V.c!(ws(wn), p)}.min = exposed(p)
                        /\ w(p) = ws(exposed(p)).c(p) ) )
```

The invariant says that each visible point comes from some window, `exposed` tells the topmost window that defines it, and its color is the color of the point in that window. Note that for convenience the invariant uses the abstraction function; of course we could have avoided this by expanding it in line, but there is no reason to do so, since the abstraction function is a perfectly good function.

```
FUNC Get(p) -> C = RET w(p)

PROC Paint(wn, p, c) =
     VAR p0 | p = p0 - offset(wn) =>                    % the screen coordinate
       IF wn <= exposed(p0) => w(p0) := c; exposed(p0) := wn [*] SKIP FI;
       wH(wn)(p) := c                                   % update the history var
END WinImpl
```

4. Spec Reference Manual

Spec is a language for writing specifications and the first few stages of successive refinement towards practical code. As a specification language it includes constructs (quantifiers, backtracking or non-determinism, some uses of atomic brackets) which are impractical in final code; they are there because they make it easier to write clear, unambiguous and suitably general specs. If you want to write a practical program, avoid them.

This document defines the syntax of the language precisely and the semantics informally**. You should read the** *Introduction to Spec* (handout 3) **before trying to read this manual.** In fact, this manual is intended mainly for reference; rather than reading it carefully, skim through it, and then use the index to find what you need. For a precise definition of the atomic semantics read *Atomic Semantics of Spec* (handout 9). Handout 17 on *Formal Concurrency* gives the non-atomic semantics semi-formally.

1. Overview

Spec is a notation for writing specs for a discrete system. What do we mean by a spec? It is the allowed sequences of transitions of a state machine. So Spec is a notation for describing sequences of transitions of a state machine.

Expressions and commands

The Spec language has two essential parts:

An *expression* describes how to compute a value as a function of other values, either constants or the current values of state variables.

A *command* describes possible transitions, or changes in the values of the state variables.

Both are based on the *state*, which in Spec is a mapping from names to values. The names are called state variables or simply variables: in the examples below they are `i` and `j`.

There are two kinds of commands:

An *atomic* command describes a set of possible transitions. For instance, the command `<< i := i + 1 >>` describes the transitions `i=1`→`i=2`, `i=2`→`i=3`, etc. (Actually, many transitions are summarized by `i=1`→`i=2`, for instance, `(i=1, j=1)`→`(i=2, j=1)` and `(i=1, j=15)`→`(i=2, j=15)`). If a command allows more than one transition from a given state we say it is *non-deterministic*. For instance, the command, `<< i := 1 [] i := i + 1 >>` allows the transitions `i=2`→`i=1` and `i=2`→`i=3`. More on this in *Atomic Semantics of Spec*.

A *non-atomic* command describes a set of sequences of states. More on this in *Formal Concurrency*.

A sequential program, in which we are only interested in the initial and final states, can be described by an atomic command.

Spec's notation for commands, that is, for changing the state, is derived from Edsger Dijkstra's guarded commands (E. Dijkstra, *A Discipline of Programming*, Prentice-Hall, 1976) as extended

by Greg Nelson (G. Nelson, A generalization of Dijkstra's calculus, *ACM TOPLAS* **11**, 4, Oct. 1989, pp 517-561). The notation for expressions is derived from mathematics.

Organizing a program

In addition to the expressions and commands that are the core of the language, Spec has four other mechanisms that are useful for organizing your program and making it easier to understand.

A *routine* is a named computation with parameters (passed by value). There are four kinds:

A *function* is an abstraction of an expression.

An *atomic procedure* is an abstraction of an atomic command.

A general procedure is an abstraction of a non-atomic command.

A *thread* is the way to introduce concurrency.

A *type* is a stylized assertion about the set of values that a name can assume. A type is also an easy way to group and name a collection of routines, called its *methods*, that operate on values in that set.

An *exception* is a way to report an unusual outcome.

A *module* is a way to structure the name space into a two-level hierarchy. An identifier `i` declared in a module `m` is known as `i` in `m` and as `m.i` throughout the program. A *class* is a module that can be instantiated many times to create many objects.

A Spec program is some global declarations of variables, routines, types, and exceptions, plus a set of modules each of which declares some variables, routines, types, and exceptions.

Outline

This manual describes the language bottom-up:
- Lexical rules
- Types
- Expressions
- Commands
- Modules

At the end there are two sections with additional information:
- Scope rules
- Built-in methods for set, sequence, and routine types.

There is also an index. The *Introduction to Spec* has a one-page language summary.

2. Grammar rules

Nonterminal symbols are in lower case; terminal symbols are punctuation other than `::=`, or are quoted, or are in upper case.

Alternative choices for a nonterminal are on separate lines.

`symbol*` denotes zero of more occurrences of `symbol`.

The symbol `empty` denotes the empty string.

If `x` is a nonterminal, the nonterminal `xList` is defined by

```
xList       ::= x
                x , xList
```

A comment in the grammar runs from `%` to the end of the line; this is just like Spec itself.

A [n] in a comment means that there is an explanation in a note labeled [n] that follows this chunk of grammar.

3. Lexical rules

The symbols of the language are literals, identifiers, keywords, operators, and the punctuation `( ) [ ] { } , ; : . | << >> := => -> [] [*]`. Symbols must not have embedded white space. They are always taken to be as long as possible.

A *literal* is a decimal number such as `3765`, a quoted character such as `'x'`, or a double-quoted string such as `"Hello\n"`.

An *identifier* (`id`) is a letter followed by any number of letters, underscores, and digits followed by any number of ' characters. Case is significant in identifiers. By convention type and procedure identifiers begin with a capital letter. An identifier may not be the same as a keyword. The *predefined* identifiers `Any, Bool, Char, Int, Nat, Null, String, true, false`, and `nil` are declared in every program. The meaning of an identifier is established by a declaration; see section 8 on scope for details. Identifiers cannot be redeclared.

By convention *keywords* are written in upper case, but you can write them in lower case if you like; the same strings with mixed case are not keywords, however. The keywords are

ALL	APROC	AS	BEGIN	BY	CLASS
CONST	DO	END	ENUM	EXCEPT	EXCEPTION
EXISTS	EXPORT	FI	FUNC	HAVOC	IF
IN	IS	LAMBDA	MODULE	OD	PROC
RAISE	RAISES	RET	SEQ	SET	SKIP
SUCHTHAT	THREAD	TYPE	VAR	WHILE	WITH

An *operator* is any sequence of the characters `!@#$^&*-+=:.<>?/\|~` except the sequences `: . | << >> := => ->` (these are punctuation), or one of the keyword operators `AS`, `IN`, and `IS`.

A comment in a Spec program runs from a `%` outside of quotes to the end of the line. It does not change the meaning of the program.

4. Types

A type defines a set of values; we say that a value `v` has type `T` if `v` is in `T`'s set. The sets are not disjoint, so a value can belong to more than one set and therefore can have more than one type. In addition to its value set, a type also defines a set of routines (functions or procedures) called its *methods*; a method normally takes a value of the type as its first argument.

An expression has exactly one type, determined by the rules in section 5; the result of the expression has this type unless it is an exception.

The picky definitions given on the rest of this page are the basis for Spec's type-checking. You can skip them on first reading, or if you don't care about type-checking.

About unions: If the expression `e` has type `T` we say that `e` has a routine type `W` if `T` is a routine type `W` or if `T` is a union type and exactly one type `W` in the union is a routine type. Note that this covers sequence, tuple, and record types. Under corresponding conditions we say that `e` has a set type.

Two types are *equal* if their definitions are the same (that is, have the same parse trees) after all type names have been replaced by their definitions and all `WITH` clauses have been discarded. Recursion is allowed; thus the expanded definitions might be infinite. Equal types define the same value set. Ideally the reverse would also be true, but type equality is meant to be decided by a type checker, whereas the set equality is intractable.

A type `T` *fits* a type `U` if the type-checker thinks it's OK to use a `T` where a `U` is required. This is true if the type-checker thinks they may have some non-trivial values in common. This can only happen if they have the same structure, and each part of `T` fits the corresponding part of `U`. 'Fits' is an equivalence relation. Precisely, `T` fits `U` if:

`T = U`.

`T` is `T' SUCHTHAT F` or `(... + T' + ...)` and `T'` fits `U`, or vice versa. There may be no values in common, but the type-checker can't analyze the `SUCHTHAT` clauses to find out. There's a special case for the `SUCHTHAT` clauses of record and tuple types, which the type-checker *can* analyze: `T`'s `SUCHTHAT` must imply `U`'s.

`T=T1->T2 RAISES EXt` and `U=U1->U2 RAISES EXu`, or one or both `RAISES` are missing, and `U1` fits `T1` and `T2` fits `U2`. Similar rules apply for `PROC` and `APROC` types. This also covers sequences. Note that the test is reversed for the argument types.

`T=SET T'` and `U=SET U'` and `T'` fits `U'`.

`T` *includes* `U` if the same conditions apply with "fits" replaced by "includes", all the "vice versa" clauses dropped, and in the `->` rule "`U1` fits `T1`" replaced by "`U1` includes `T1` and `EXt` is a superset of `EXu`". If `T` includes `U` then `T`'s value set includes `U`'s value set; again, the reverse is intractable.

An expression `e` fits a type `U` in state `s` if `e`'s type fits `U` and the result of `e` in state `s` has type `U` or is an exception; in general this can only be checked at runtime unless `U` includes `e`'s type. The check that `e` fits `T` is required for assignment and routine invocation; together with a few other checks it is called *type-checking*. The rules for type-checking are given in sections 5 and 6.

```
type          ::= name                          % name of a type
                  "Any"                         % every value has this type
                  "Null"                        % with value set {nil}
                  "Bool"                        % with value set {true, false}
                  "Char"                        % like an enumeration
                  "String"                      % = SEQ Char
                  "Int"                         % integers
                  "Nat"                         % naturals: non-negative integers
                  SEQ type                      % sequence [1]
                  SET type                      % set
                  [ declList ]                  % record with declared fields [7]
                  ( typeList )                  % tuple; (T) is the same as T [8]
                   ( union )                    % union of the types
                  aType ->  type raises         % function [2]
                  aType ->> type raises         % relation [2]
                  APROC aType returns raises    % atomic procedure [2]
                  PROC  aType returns raises    % non-atomic procedure [2]
                  type WITH { methodDefList }   % attach methods to a type [3]
                  type SUCHTHAT primary         % restrict the value set [4]
                  IN exp                        % = T SUCHTHAT (\ t: T | t IN exp)
                                                % where exp's type has an IN method
                  id [ typeList ] . id          % type from a module [5]

name          ::= id . id                       % the first id denotes a module
                  id                            % short for m.id if id is declared
                                                % in the current module  m, and for
                                                % Global.id if id is declared globally
                  type . id                     % the id method of type
decl          ::= id : type                     % id has this type
                  id                            % short for id: Id [6]

union         ::= type  + type
                  union + type

aType         ::= ()
                  type
returns       ::= empty                         % only for procedures
                  -> type
raises        ::= empty
                  RAISES exceptionSet           % the exceptions it can return

exceptionSet  ::= { exceptionList }             % a set of exceptions
                  name                          % declared as an exception set
                  exceptionSet \/ exceptionSet  % set union
                  exceptionSet - exceptionSet   % set difference
exception     ::= id                            % means "id"

method        ::= id
                  stringLiteral                 % the string must be an operator
                                                % other than "=" or "#" (see section 3)
methodDef     ::= method := name                % name is a routine
```


The ambiguity of the type grammar is resolved by taking `->` to be right associative and giving `WITH` and `RAISES` higher precedence than `->`.

[1] A `SEQ T` is just a function from `0..size-1` to `T`. That is, it is short for

```
(Int->T) SUCHTHAT (\ f: Int->T | (EXISTS size: Int | f.dom = 0..size-1))
        WITH { see section 9 }.
```

This means that invocation, `!`, and `*` work for a sequence just as they do for any function. In addition, there are many other useful operators on sequences; see section 9. The `String` type is just `SEQ Char`; there are `String` literals, defined in section 5.

[2] A `T->U` value is a partial function from a state and a value of type `T` to a value of type `U`. A `T->U RAISES xs` value is the same except that the function may raise the exceptions in `xs`.

A function or procedure declared with names for the arguments, such as

```
(\ i: Int, s: String | i + StringToInt(x))
```

has a type that ignores the names, `(Int, String)->Int`. However, it also has a method `argNames` that returns the sequence of argument names, `{"i", "s"}` in the example, just like a record. This makes it possible to match up arguments by name, as in the following example.

A database is a set `s` of records. A selection query `q` is a predicate that we want to apply to the records. How do we get from the field names, which are strings, to the argument for `q`? Assume that `q` has an `argNames` method. So if `r IN s`, `q.argNames * r` is the tuple that we want to feed to `q`; `q$(q.argNames * r)` is the query, where `$` is the operator that applies a function to a tuple of its arguments.

[3] We say `m` is a *method* of `T` defined by `f`, and denote `f` by `T.m`, if

- `T = T' WITH {..., m := f, ...}` and `m` is an identifier or is `"op"` where `op` is an operator (the construct in braces is a `methodDefList`), or
- `T = T' WITH { methodDefList }`, `m` is not defined in `methodDefList`, and `m` is a method of `T'` defined by `f`, or
- `T = (... + T' + ...)`, `m` is a method of `T'` defined by `f`, and there is no other type in the union with a method `m`.

There are two special forms for invoking methods: `e1 infixOp e2` or `prefixOp e`, and `e1.id(e2)` or `e.id` or `e.id()`. They are explained in notes [1] and [3] to the expression grammar in the next section. This notation may be familiar from object-oriented languages. Unlike many such languages, Spec makes no provision for varying the method in each object, though it does allow inheritance and overriding.

A method doesn't have to be a routine, though the special forms won't type-check unless the method is a routine. Any method `m` of `T` can be referred to by `T.m`.

If type `U` has method `m`, then the function type `V = T->U` has a *lifted* method `m` that composes `U.m` with `v`, unless `V` already has a `m` method. `V.m` is defined by

```
(\ v | (\ t | v(t).m))
```

so that `v.m = v * U.m`. For example, `{"a", "ab", "b"}.size = {1, 2, 1}`. If `m` takes a second argument of type `W`, then `V.m` takes a second argument of type `VV = T->W` and is defined on the intersection of the domains by applying `m` to the two results. Thus in this case `V.m` is

```
(\ v, vv | (\ t :IN v.dom /\ vv.dom | v(t).m(vv(t))))
```

Lifting also works for relations to `U`, and therefore also for `SET U`. Thus if `R = (T,U)->Bool` and `m` returns type `X`, `R.m` is defined by

```
(\ r | (\ t, x | x IN {u | r(t, u) || u.m}))
```

so that `r.m = r * U.m.rel`. If `m` takes a second argument, then `R.m` takes a second argument of type `RR = T->W`, and `r.m(rr)` relates `t` to `u.m(w)` whenever `r` relates `t` to `u` and `rr` relates `t` to `w`. In other words, `R.m` is defined by

```
(\ r, rr | (\ t, x | x IN {u, w | r(t, u) /\ rr(t, w) || u.m(w)}))
```

If `U` doesn't have a method `m` but `Bool` does, then the lifting is done on the function that defines the relation, so that `r1 \/ r2` is the union of the relations, `r1 /\ r2` the intersection, `r1 - r2` the difference, and `~r` the complement.

[4] In `T SUCHTHAT E`, `E` is short for a predicate on `T`'s, that is, a function `(T -> Bool)`. If the context is `TYPE U = T SUCHTHAT E` and `this` doesn't occur free in `E`, the predicate is `(\ u: T | E)`, where `u` is `U` with the first letter changed to lower-case; otherwise the predicate is `(\ this: T | E)`. The type `T SUCHTHAT E` has the same methods as `T`, and its value set is the values of `T` for which the predicate is true. See section 5 for `primary`.

[5] If a type is defined by `m[typeList].id` and `m` is a parameterized module, the meaning is `m'.id` where `m'` is defined by `MODULE m' = m[typeList] END m'`. See section 7 for a full discussion of this kind of type.

[6] `Id` is the `id` of a type, obtained from `id` by dropping trailing ' characters and digits, and capitalizing the first letter or all the letters (it's an error if these capitalizations yield different identifiers that are both known at this point).

[7] The type of a record is `String->Any SUCHTHAT ...`. The `SUCHTHAT` clauses are of the form `this("f") IS T`; they specify the types of the fields. In addition, a record type has a method called `fields` whose value is the sequence of field names (it's the same for every record). Thus `[f: T, g: U]` is short for

```
String->Any WITH { fields:=(\r: String->Any | (SEQ String){"f", "g"}) }
            SUCHTHAT   this.dom >= {"f", "g"}
                       /\ this("f") IS T /\ this("g") IS U
```

[8] The type of a tuple is `Nat->Any SUCHTHAT ...`. As with records, the `SUCHTHAT` clauses are of the form `this("f") IS T`; they specify the types of the fields. In addition, a tuple type has a method called `fields` whose value is `0..n-1` if the tuple has `n` fields. Thus `(T, U)` is short for

```
   Int->Any WITH { fields:=(\r: Int->Any | 0..1) }
               SUCHTHAT   this.dom = 0..1
                          /\ this(0) IS T /\ this(1) IS U
```

Thus to convert a record `r` into a tuple, write `r.fields * r`, and to convert a tuple `t` into a record, write `r.fields.inv * t`.

There is no special syntax for tuple fields, since you can just write `t(2)` and `t(2) := e` to read and write the third field, for example (remember that fields are numbered from 0).

5. Expressions

An expression is a partial function from states to results; results are values or exceptions. That is, an expression computes a result for a given state. The state is a function from names to values. This state is supplied by the command containing the expression in a way explained later. The meaning of an expression (that is, the function it denotes) is defined informally in this section. The meanings of invocations and lambda function constructors are somewhat tricky, and the informal explanation here is supplemented by a formal account in *Atomic Semantics of Spec*. Because expressions don't have side effects, the order of evaluation of operands is irrelevant (but see [5] and [13]).

Every expression has a type. The result of the expression is a member of this type if it is not an exception. This property is guaranteed by the *type-checking* rules, which require an expression used as an argument, the right hand side of an assignment, or a routine result to fit the type of the formal, left hand side, or routine range (see section 4 for the definition of 'fit'). In addition, expressions appearing in certain contexts must have *suitable* types: in `e1(e2)`, `e1` must have a routine type; in `e1+e2`, `e1` must have a type with a `"+"` method, etc. These rules are given in detail in the rest of this section. A union type is suitable if exactly one of the members is suitable. Also, if `T` is suitable in some context, so are `T WITH {...}` and `T SUCHTHAT f`.

An expression can be a literal, a variable known in the scope that contains the expression, or a function invocation. The form of an expression determines both its type and its result in a state:

`literal` has the type and value of the literal.

`name` has the declared type of `name` and its value in the current state, `state("name")`. The form `T.m` (where `T` denotes a type) is also a name; it denotes the `m` method of `T`. Note that if `name` is `id` and `id` is declared in the current module `m`, then it is short for `m.id`.

invocation `f(e)`: `f` must have a function (not procedure) type `U->T RAISES EX` or `U->T` (note that a sequence is a function), and `e` must fit `U`; then `f(e)` has type `T`. In more detail, if `f` has result `rf` and `e` has type `U'` and result `re`, then `U'` must fit `U` (checked statically) and `re` must have type `U` (checked dynamically if `U'` involves a union or `SUCHTHAT`; if the dynamic check fails the result is a fatal error). Then `f(e)` has type `T`.

If either `rf` or `re` is undefined, so is `f(e)`. Otherwise, if either is an exception, that exception is the result of `f(e)`; if both are, `rf` is the result.

If both `rf` and `re` are normal, the result of `rf` at `re` can be:

A normal value, which becomes the result of `f(e)`.

An exception, which becomes the result of `f(e)`. If `rf` is defined by a function body that loops, the result is a special looping exception that you cannot handle.

Undefined, in which case `f(e)` is undefined and the command containing it fails (has no outcome) — failure is explained in section 6.

A function invocation in an expression never affects the state. If the result is an exception, the containing command has an exceptional outcome; for details see section 6.

The other forms of expressions (`e.id`, `constructor`s, prefix and infix operators, combinations, and quantifications) are all syntactic sugar for function invocations, and their results are obtained by the rule used for invocations. There is a small exception for conditionals [5] and for the conditional logical operators `/\`,`\/`, and `==>` that are defined in terms of conditionals [13].

```
exp          ::= primary
                 prefixOp exp                  % [1]
                 exp infixOp exp               % [1]
                 infixOp : exp                 % exp's elements combined by op [2]
                 exp IS type                   % (EXISTS x: type | exp = x)
                 exp AS type                   % error unless (exp IS type) [14]

primary      ::= literal
                 name
                 primary . id                  % method invocation [3] or record field
                 primary arguments             % function invocation
                 constructor
                 ( exp )
                 ( quantif declList | pred )   % /\:{d | p} for ALL, \/ for EXISTS [4]
                 ( pred => exp1 [*] exp2 )     % if pred then exp1 else exp2 [5]
                 ( pred => exp1 )              % undefined if pred is false

literal      ::= intLiteral                    % sequence of decimal digits
                 charLiteral                   % 'x', x a printing character
                 stringLiteral                 % "xxx", with \ escapes as in C

arguments    ::= ( expList )                   % the arg is the tuple (expList)
                 ( )

constructor  ::= { }                           % empty function/sequence/set [6]
                 { expList }                   % sequence/set constructor [6]
                 ( expList )                   % tuple constructor
                 name    { }                   % name denotes a func/seq/set type [6]
                 name    { expList }           % name denotes a seq/set/record type [6]
                 primary { fieldDefList  }     % record constructor [7]
                 primary { exp -> result }     % function or sequence constructor [8]
                 primary { *   -> result }     % function constructor [8]
                 ( LAMBDA signature = cmd )    % function with the local state [9]
                 ( \ declList | exp )          % short for (LAMBDA(d)->T=RET exp) [9]
                 { declList   | pred || exp }  % set constructor [10]
                 { seqGenList | pred || exp }  % sequence constructor [11]

fieldDef     ::= id := exp

result       ::= empty                         % the function is undefined
                 exp                           % the function yields exp
                 RAISE exception               % the function yields exception

seqGen       ::= id := exp BY exp WHILE exp    % sequence generator [11]
                 id :IN exp

pred         ::= exp                           % predicate, of type Bool
quantif      ::= ALL
                 EXISTS
```


		(precedence)	***argument/result types***		***operation***
infixOp	::=	** % (8)	(Int, Int)->Int		exponentiate
		* % (7)	(Int, Int)->Int		multiply
		%	(T->U, U->V)->(T->V)	[12]	function composition
		/ % (7)	(Int, Int)->Int		divide
		// % (7)	(Int, Int)->Int		remainder
		+ % (6)	(Int, Int)->Int		add
		%	(SEQ T, SEQ T)->SEQ T	[12]	concatenation
		%	(T->U, T->U)->(T->U)	[12]	function overlay
		- % (6)	(Int, Int)->Int		subtract
		%	(SET T, SET T)->SET T	[12]	set difference;
		%	(SEQ T, SEQ T)->SEQ T	[12]	multiset difference
		! % (6)	(T->U, T)->Bool	[12]	function is defined
		!! % (6)	(T->U, T)->Bool	[12]	func has normal value
		$ % (6)	(T->U, T)->U	[15]	apply func to tuple
		.. % (5)	(Int, Int)->SEQ Int	[12]	subrange
		<= % (4)	(Int, Int)->Bool		less than or equal
		%	(SET T, SET T)->Bool	[12]	subset
		%	(SEQ T, SEQ T)->Bool	[12]	prefix
		< % (4)	(T, T)->Bool, T with <=		less than
		%	e1<e2 = (e1<=e2 /\ e1#e2)		
		> % (4)	(T, T)->Bool, T with <=		greater than
		%	e1>e2 = e2<e1		
		>= % (4)	(T, T)->Bool, T with <=		greater or equal
		%	e1>=e2 = e2<=e1		
		= % (4)	(Any, Any)->Bool	[1]	equal
		# % (4)	(Any, Any)->Bool		not equal
		%	e1#e2 = ~ (e1=e2)		
		<<= % (4)	(SEQ T, SEQ T)->Bool	[12]	non-contiguous sub-seq
		IN % (4)	(T, SET T)->Bool	[12]	membership
		/\ % (2)	(Bool, Bool)->Bool	[13]	conditional and
		%	(SET T, SET T)->SET T	[12]	intersection
		\/ % (1)	(Bool, Bool)->Bool	[13]	conditional or
		%	(SET T, SET T)->SET T	[12]	union
		==> % (0)	(Bool, Bool)->Bool	[13]	conditional implies
		op % (5)	not one of the above	[1]	
prefixOp	::=	- % (6)	Int->Int		negation
		~ % (3)	Bool->Bool		complement
		op % (5)	not one of the above	[1]	

The ambiguity of the expression grammar is resolved by taking the `infixOps` to be left associative and using the indicated precedences for the `prefixOps` and `infixOps` (with 8 for `IS` and `AS` and 5 for `:` or any operator not listed); higher numbers correspond to tighter binding. The precedence is determined by the operator symbol and doesn't depend on the operand types.

[1] The meaning of `prefixOp e` is `T."prefixOp"(e)`, where `T` is `e`'s type, and of `e1 infixOp e2` is `T1."infixOp"(e1, e2)`, where `T1` is `e1`'s type. The built-in types `Int` (and `Nat` with the same operations), `Bool`, sequences, sets, and functions have the operations given in the grammar. Section 9 on built-in methods specifies the operators for built-in types other than `Int` and `Bool`. Special case: `e1 IN e2` means `T2."IN"(e1, e2)`, where `T2` is `e2`'s type.

Note that the `=` operator does not require that the types of its arguments agree, since both are `Any`. Also, `=` and `#` cannot be overridden by `WITH`. To define your own abstract equality, use a different operator such as `"=="`.

[2] The `exp` must have type `SEQ T` or `SET T`. The value is the elements of `exp` combined into a single value by `infixOp`, which must be associative and have an identity, and must also be commutative if `exp` is a set. Thus

```
+ : {i: Int | 0<i /\ i<5 | i**2} = 1 + 4 + 9 + 16 = 30,
```

and if `s` is a sequence of strings, `+ : s` is the concatenation of the strings. For another example, see the definition of quantifications in [4]. Note that the entire set is evaluated; see [10].

[3] Methods can be invoked by dot notation.

The meaning of `e.id` or `e.id()` is `T.id(e)`, where `T` is `e`'s type.

The meaning of `e1.id(e2)` is `T.id(e1, e2)`, where `T` is `e1`'s type.

Section 9 on built-in methods gives the methods for built-in types other than `Int` and `Bool`.

[4] A quantification is a conjunction (if the quantifier is `ALL`) or disjunction (if it is `EXISTS`) of the `pred` with the `id`'s in the `declList` bound to every possible value (that is, every value in their types); see section 4 for `decl`. Precisely, `(ALL d | p) = /\ : {d | p}` and `(EXISTS d | p) = \/ : {d | p}`. All the expressions in these expansions are evaluated, unlike `e2` in the expressions `e1 /\ e2` and `e1 \/ e2` (see [10] and [13]).

[5] A conditional `(pred => e1 [*] e2)` is not exactly an invocation. If `pred` is true, the result is the result of `e1` even if `e2` is undefined or exceptional; if `pred` is false, the result is the result of `e2` even if `e1` is undefined or exceptional. If `pred` is undefined, so is the result; if `pred` raises an exception, that is the result. If `[*] e2` is omitted and `pred` is false, the result is undefined.

[6] In a `constructor {expList}` each `exp` must have the same type `T`, the type of the `constructor` is `(SEQ T + SET T)`, and its value is the sequence containing the values of the `exp`s in the given order, which can also be viewed as the set containing these values.

If `expList` is empty the type is the union of all function, sequence and set types, and the value is the empty sequence or set, or a function undefined everywhere. If desired, these constructors can be prefixed by a `name` denoting a suitable set or sequence type.

A constructor `T{e1, ..., en}`, where `T` is a record type `[f1: T1, ..., fn: Tn]`, is short for a record constructor (see [7]) `T{f1:=e1, ..., fn:=en}`.

[7] The `primary` must have a record type, and the constructor has the same type as its `primary` and denotes the same value except that the fields named in the `fieldDefList` have the given values. Each value must fit the type declared for its `id` in the record type. The `primary` may also denote a record type, in which case any fields missing from the `fieldDefList` are given arbi-

trary (but deterministic) values. Thus if `R=[a: Int, b: Int]`, `R{a := 3, b := 4}` is a record of type `R` with `a=3` and `b=4`, and `R{a := 3, b := 4}{a := 5}` is a record of type `R` with `a=5` and `b=4`. If the record type is qualified by a `SUCHTHAT`, the fields get values that satisfy it, and the constructor is undefined if that's not possible.

[8] The `primary` must have a function or sequence type, and the constructor has the same type as its `primary` and denotes a value equal to the value denoted by the `primary` except that it maps the argument value given by `exp` (which must fit the domain type of the function or sequence) to `result` (which must fit the range type if it is an `exp`). For a function, if `result` is `empty` the constructed function is undefined at `exp`, and if `result` is `RAISE exception`, then `exception` must be in the `RAISES` set of `primary`'s type. For a sequence `result` must not be `empty` or `RAISE`, and `exp` must be in `primary.dom` or the constructor expression is undefined.

In the `*` form the `primary` must be a function type or a function, and the value of the constructor is a function whose result is `result` at every value of the function's domain type (the type on the left of the `->`). Thus if `F=(Int->Int)` and `f=F{*->0}`, then `f` is zero everywhere and `f{4->1}` is zero except at `4`, where it is `1`. If this value doesn't have the function type, the constructor is undefined; this can happen if the type has a `SUCHTHAT` clause. For example, the type can't be a sequence.

[9] A `LAMBDA constructor` is a statically scoped function definition. When it is invoked, the meaning of the body is determined by the local state when the `LAMBDA` was evaluated and the global state when it is invoked; this is ad-hoc but convenient. See section 7 for `signature` and section 6 for `cmd`. The `returns` in the `signature` may not be `empty`. Note that a function can't have side effects.

The form `(\ declList | exp)` is short for `(LAMBDA (declList) -> T = RET exp)`, where `T` is the type of `exp`. See section 4 for `decl`.

[10] A set `constructor { declList | pred || exp }` has type `SET T`, where `exp` has type `T` in the current state augmented by `declList`; see section 4 for `decl`. Its value is a set that contains `x` iff `(EXISTS declList | pred /\ x = exp)`. Thus

```
{i: Int | 0<i /\ i<5 || i**2} = {1, 4, 9, 16}
```

and both have type `SET Int`. If `pred` is omitted it defaults to `true`. If `| exp` is omitted it defaults to the last `id` declared:

```
{i: Int | 0<i /\ i<5} = {1, 2, 3, 4 }
```

Note that if `s` is a set or sequence, `IN s` is a type (see section 4), so you can write a constructor like `{i :IN s | i > 4}` for the elements of `s` greater than 4. This is shorter and clearer than

```
{i | i IN s /\ i > 4}
```

If there are any values of the declared `id`'s for which `pred` is undefined, or `pred` is true and `exp` is undefined, then the result is undefined. If nothing is undefined, the same holds for exceptions; if more than one exception is raised, the result exception is an arbitrary choice among them.

[11] A sequence constructor `{ seqGenList | pred || exp }` has type `SEQ T`, where `exp` has type `T` in the current state augmented by `seqGenList`, as follows. The value of

```
{x1 := e01 BY e1 WHILE p1, ... , xn := e0n BY en WHILE pn | pred || exp}
```

is the sequence which is the value of `result` produced by the following program. Here `exp` has type `T` and `result` is a fresh identifier (that is, one that doesn't appear elsewhere in the program). There's an informal explanation after the program.

```
VAR x2 := e02, ..., xn := e0n, result := T{}, x1 := e01 |
    DO p1 => x2 := e2; p2 => ... => xn := en; pn =>
```

```
            IF pred => result := result + {exp} [*] SKIP FI;
            x1 := e1
        OD
```

However, `e0i` and `ei` are not allowed to refer to `xj` if `j > i`. Thus the `n` sequences are unrolled in parallel until one of them ends, as follows. All but the first are initialized; then the first is initialized and all the others computed, then all are computed repeatedly. In each iteration, once all the `xi` have been set, if `pred` is true the value of `exp` is appended to the result sequence; thus `pred` serves to filter the result. As with set constructors, an omitted `pred` defaults to `true`, and an omitted `|| exp` defaults to `|| xn`. An omitted `WHILE pi` defaults to `WHILE true`. An omitted `:= e0i` defaults to

```
    := {x: Ti | true}.choose
```

where `Ti` is the type of `ei`; that is, it defaults to an arbitrary value of the right type.

The generator `xi :IN ei` generates the elements of the sequence `ei` in order. It is short for

```
    j := 0 BY j + 1 WHILE j < ei.size, xi BY ei(j)
```

where `j` is a fresh identifier. Note that if the `:IN` isn't the first generator then the first element of `ei` is skipped, which is probably not what you want. Note that `:IN` in a sequence constructor overrides the normal use of `IN s` as a type (see [10]).

Undefined and exceptional results are handled the same way as in set constructors.

Examples

`{i := 0 BY i+1 WHILE i <= n}`	`= 0..n = {0, 1, ..., n}`
`(r := head BY r.next WHILE r # nil || r.val}`	the `val` fields of a list starting at `head`
`{x :IN s, sum := 0 BY sum + x}`	partial sums of `s`
`{x :IN s, sum := 0 BY sum + x}.last`	`+ : s`, the last partial sum
`{x :IN s, rev := {} BY {x} + rev}.last`	reverse of `s`
`{x :IN s || f(x)}`	`s * f`
`{i :IN 1..n | i // 2 # 0 || i * i}`	squares of odd numbers <= n
`{i :IN 1..n, iter := e BY f(iter)}`	$\{f(e), f^2(e), \ldots, f^n(e)\}$

[12] These operations are defined in section 9.

[13] The conditional logical operators are defined in terms of conditionals:

```
    e1 \/  e2 = (  e1 => true  [*] e2 )
    e1 /\  e2 = ( ~e1 => false [*] e2 )
    e1 ==> e2 = ( ~e1 => true  [*] e2 )
```

Thus the second operand is not evaluated if the value of the first one determines the result.

[14] `AS` changes only the type of the expression, not its value. Thus if `(exp IS type)` the value of `(exp AS type)` is the value of `exp`, but its type is `type` rather than the type of `exp`.

[15] `f$t` applies the function `f` to the tuple `t`. It differs from `f(t)`, which makes a tuple out of the list of expressions in `t` and applies `f` to that tuple.

6. Commands

A command changes the state (or does nothing). Recall that the state is a mapping from names to values; we denote it by `state`. Commands are non-deterministic. An atomic command is one that is inside `<<...>>` brackets.

The meaning of an atomic command is a set of possible transitions (that is, a relation) between a state and an outcome (a state plus an optional exception); there can be any number of outcomes from a given state. One possibility is a looping exceptional outcome. Another is no outcomes. In this case we say that the atomic command *fails*; this happens because all possible choices within it encounter a false guard or an undefined invocation.

If a subcommand fails, an atomic command containing it may still succeed. This can happen because it's one operand of `[]` or `[*]` and the other operand succeeds. If can also happen because a non-deterministic construct in the language that might make a different choice. Leaving exceptions aside, the commands with this property are `[]` and `VAR` (because it chooses arbitrary values for the new variables). If we gave an operational semantics for atomic commands, this situation would correspond to backtracking. In the relational semantics that we actually give (in *Atomic Semantics of Spec*), it corresponds to the fact that the predicate defining the relation is the "or" of predicates for the subcommands. Look there for more discussion of this point.

A non-atomic command defines a collection of possible transitions, roughly one for each `<<...>>` command that is part of it. If it has simple commands not in atomic brackets, each one also defines a possible transition, except for `assignment`s and `invocation`s. An `assignment` defines two transitions, one to evaluate the right hand side, and the other to change the value of the left hand side. An `invocation` defines a transition for evaluating the arguments and doing the call and one for evaluating the result and doing the return, plus all the transitions of the body. These rules are somewhat arbitrary and their details are not very important, since you can always write separate commands to express more transitions, or atomic brackets to express fewer transitions. The motivation for the rules is to have as many transitions as possible, consistent with the idea that an expression is evaluated atomically.

A complete collection of possible transitions defines the possible sequences of states or histories; there can be any number of histories from a given state. A non-atomic command still makes choices, but it does not backtrack and therefore can have histories in which it gets stuck, even though in other histories a different choice allows it to run to completion. For the details, see handout 17 on formal concurrency.

```
cmd          ::= SKIP                          % [1]
                 HAVOC                         % [1]
                 RET                           % [2]
                 RET exp                       % [2]
                 RAISE exception               % [9]
                 CRASH                         % [10]
                 invocation                    % [3]
                 assignment                    % [4]

                 cmd              []  cmd      % or [5]
                 cmd              [*] cmd      % else [5]
                 pred             =>  cmd      % guarded cmd: if pred then cmd [5]
                 VAR declInitList  |  cmd      % variable introduction [6]
                 cmd               ;  cmd      % sequential composition
                 cmd EXCEPT handler             % handle exception [9]

                 << cmd >>                      % atomic brackets [7]
                 BEGIN cmd END                  % just brackets
                 IF cmd FI                      % just brackets [5]
                 DO cmd OD                      % repeat until cmd fails [8]

invocation   ::= primary arguments              % primary has a routine type [3]

assignment   ::= lhs         := exp             % state := state{name -> exp} [4]
                 lhs infixOp := exp             % short for lhs := lhs infixOp exp
                 lhs         := invocation      % of a PROC or APROC
                 ( lhsList ) := exp             % exp a tuple that fits lhsList
                 ( lhsList ) := invocation
lhs          ::= name                          % defined in section 4
                 lhs . id                       % record field [4]
                 lhs arguments                  % function  [4]

declInit     ::= decl                          % initially any value of the type [6]
                 id : type := exp               % initially exp, which must fit type [6]
                 id        := exp               % short for id: T := exp, where
                                                %  T is the type of exp

handler      ::= exceptionSet => cmd           % [9]. See section 4 for exceptionSet
```

The ambiguity of the command grammar is resolved by taking the command composition operations `;`, `[]`, and `[*]` to be left-associative and `EXCEPT` to be right associative, and giving `[]` and `[*]` lowest precedence, `=>` and `|` next (to the right only, since their left operand is an `exp`), `;` next, and `EXCEPT` highest precedence.

[1] The empty command and `SKIP` make no change in the state. `HAVOC` produces an arbitrary outcome from any state; if you want to specify undefined behavior when a precondition is not satisfied, write `~precondition => HAVOC`.

[2] A `RET` may only appear in a routine body, and the `exp` must fit the result type of the routine. The `exp` is omitted iff the `returns` of the routine's `signature` is empty.

[3] For `arguments` see section 5. The argument are passed by value, that is, assigned to the formals of the procedure A function body cannot invoke a `PROC` or `APROC`; together with the rule for assignments (see [7]) this ensures that it can't affect the state. An atomic command can invoke an `APROC` but not a `PROC`. A command is atomic iff it is `<< cmd >>`, a subcommand of an atomic command, or one of the simple commands `SKIP`, `HAVOC`, `RET`, or `RAISE`. The type-checking rule for `invocation`s is the same as for function invocations in expressions.

[4] You can only assign to a name declared with `VAR` or in a `signature`. In an `assignment` the `exp` must fit the type of the `lhs`, or there is a fatal error. In a function body `assignments` must be to names declared in the signature or the body, to ensure that the function can't have side effects.

An assignment to a left hand side that is not a name is short for assigning a constructor to a name. In particular,

`lhs(arguments) := exp` is short for `lhs := lhs{arguments->exp}`, and
`lhs . id := exp` is short for `lhs := lhs{id := exp}`.

These abbreviations are expanded repeatedly until `lhs` is a `name`.

In an assignment the right hand side may be an `invocation` (of a procedure) as well as an ordinary expression (which can only invoke a function). The meaning of `lhs := exp` or `lhs := invocation` is to first evaluate the `exp` or do the `invocation` and assign the result to a temporary variable `v`, and then do `lhs := v`. Thus the assignment command is not atomic unless it is inside `<<...>>`.

If the left hand side of an `assignment` is a `(lhsList)`, the `exp` must be a tuple of the same length, and each component must fit the type of the corresponding `lhs`. Note that you cannot write a tuple constructor that contains procedure invocations.

[5] A guarded command fails if the result of `pred` is undefined or `false`. It is equivalent to `cmd` if the result of `pred` is `true`. A `pred` is just a Boolean `exp`; see section 4.

`S1 [] S2` chooses one of the `Si` to execute. It chooses one that doesn't fail. Usually `S1` and `S2` will be guarded. For example,
`x=1 => y:=0 [] x> 1 => y:=1` sets `y` to `0` if `x=1`, to `1` if `x>1`, and has no outcome if `x<1`. But
`x=1 => y:=0 [] x>=1 => y:=1` might set `y` to `0` or `1` if `x=1`.

`S1 [*] S2` is the same as `S1` unless `S1` fails, in which case it's the same as `S2`.

`IF ... FI` are just command brackets, but it often makes the program clearer to put them around a sequence of guarded commands, thus:

```
IF   x < 0 => y := 3
[]   x = 0 => y := 4
[*]           y := 5
FI
```

[6] In a `VAR` the unadorned form of `declInit` initializes a new variable to an arbitrary value of the declared type. The `:=` form initializes a new variable to `exp`. Precisely,

```
VAR id: T := exp | c
```

is equivalent to

```
VAR id: T | id := exp; c
```

The `exp` could also be a procedure invocation, as in an `assignment`.

Several `declInit`s after `VAR` is short for nested `VAR`s. Precisely,

```
VAR declInit ,     declInitList  | cmd
```

is short for

```
VAR declInit | VAR declInitList | cmd
```

This is unlike a module, where all the names are introduced in parallel.

[7] In an atomic command the atomic brackets can be used for grouping instead of `BEGIN ... END`; since the command can't be any more atomic, they have no other meaning in this context.

[8] Execute `cmd` repeatedly until it fails. If `cmd` never fails, the result is a looping exception that doesn't have a name and therefore can't be handled. Note that this is *not* the same as failure.

[9] Exception handling is as in Clu, but a bit simplified. Exceptions are named by literal strings (which are written without the enclosing quotes). A module can also declare an identifier that denotes a set of exceptions. A command can have an attached exception `handler`, which gets to look at any exceptions produced in the command (by `RAISE` or by an invocation) and not handled closer to the point of origin. If an exception is not handled in the body of a routine, it is raised by the routine's invocation.

An exception `ex` must be in the `RAISES` set of a routine `r` if either `RAISE ex` or an invocation of a routine with `ex` in its `RAISES` set occurs in the body of `r` outside the scope of a handler for `ex`.

[10] `CRASH` stops the execution of any current invocations in the module other than the one that executes the `CRASH`, and discards their local state. The same thing happens to any invocations outside the module from within it. After `CRASH`, no procedure in the module can be invoked from outside until the routine that invokes it returns. `CRASH` is meant to be invoked from within a special `Crash` procedure in the module that models the effects of a failure.

7. Modules

A program is some global declarations plus a set of modules. Each module contains variable, routine, exception, and type declarations.

Module definitions can be parameterized with `mformals` after the module `id`, and a parameterized module can be instantiated. Instantiation is like macro expansion: the formal parameters are replaced by the arguments throughout the body to yield the expanded body. The parameters must be types, and the body must type-check without any assumptions about the argument that replaces a formal other than the presence of a `WITH` clause that contains all the methods mentioned in the formal parameter list (that is, formals are treated as distinct from all other types).

Each module is a separate scope, and there is also a `Global` scope for the identifiers declared at the top level of the `program`. An identifier `id` declared at the top level of a non-parameterized module `m` is short for `m.id` when it occurs in `m`. If it appears in the `exports`, it can be denoted by `m.id` anywhere. When an identifier `id` that is declared globally occurs anywhere, it is short for `Global.id`. `Global` cannot be used as a module `id`.

An exported `id` must be declared in the module. If an exported `id` has a `WITH` clause, it must be declared in the module as a type with at least those methods, and only those methods are accessible outside the module; if there is no `WITH` clause, all its methods and constructors are accessible. This is Spec's version of data abstraction.


```
program      ::= toplevel* module* END

module       ::= modclass id mformals exports = body END id

modclass     ::= MODULE
                 CLASS                          % [4]

exports      ::= EXPORT exportList
export       ::= id
                 id WITH {methodList}           % see section 4 for method

mformals     ::= empty
                 [ mfpList ]
mfp          ::= id                             % module formal parameter
                 id WITH { declList }           % see section 4 for decl

body         ::= toplevel*                      % id must be the module id
                 id [ typeList ]                % instance of parameterized module

toplevel     ::= VAR declInit*                  % declares the decl ids [1]
                 CONST declInit*                % declares the decl ids as constant
                 routineDecl                    % declares the routine id
                 EXCEPTION exSetDecl*           % declares the exception set ids
                 TYPE typeDecl*                 % declares the type ids and any
                                                % ids in ENUMs

routineDecl  ::= FUNC   id signature =  cmd     % function
                 APROC  id signature =<<cmd>>   % atomic procedure
                 PROC   id signature =  cmd     % non-atomic procedure
                 THREAD id signature =  cmd     % one thread for each possible
                                                % invocation of the routine [2]
signature    ::= ( declList ) returns raises    % see section 4 for returns
                 ( )          returns raises    %  and raises

exSetDecl    ::= id = exceptionSet              % see section 4 for exceptionSet

typeDecl     ::= id = type                      % see section 4 for type
                 id = ENUM [ idList ]           % a value is one of the id's [3]
```

[1] The "`:= exp`" in a `declInit` (defined in section 6) specifies an initial value for the variable. The `exp` is evaluated in a state in which each variable used during the evaluation has been initialized, and the result must be a normal value, not an exception. The `exp` sees all the names known in the scope, not just the ones that textually precede it, but the relation "used during evaluation of initial values" on the variables must be a partial order so that initialization makes sense. As in an `assignment`, the `exp` may be a procedure invocation as well as an ordinary expression. It's a fatal error if the `exp` is undefined or the invocation fails.

[2] Instead of being invoked by the client of the module or by another procedure, a thread is automatically invoked in parallel once for every possible value of its arguments. The thread is named by the `id` in the declaration together with the argument values. So

```
VAR sum := 0, count := 0
THREAD P(i: Int) = i IN 0 .. 9 =>
  VAR t | t := F(i); <<sum := sum + t>>; <<count := count + 1>>
```

adds up the values of `F(0) ... F(9)` in parallel. It creates a thread `P(i)` for every integer `i`; the threads `P(0), ..., P(9)` for which the guard is true invoke `F(0), ..., F(9)` in parallel and total the results in `sum`. When `count = 10` the total is complete.

A thread is the only way to get an entire program to do anything (except evaluate initializing expressions, which could have side effects), since transitions only happen as part of some thread.

[3] The `id`'s in the list are declared in the module; their type is the `ENUM` type. There are no operations on enumeration values except the ones that apply to all types: equality, assignment, and routine argument and result communication.

[4] A class is shorthand for a module that declares a convenient object type. The next few paragraphs specify the shorthand, and the last one explains the intended usage.

If the class `id` is `Obj`, the module `id` is `ObjMod`. Each variable declared in a top level `VAR` in the class becomes a field of the `ObjRec` record type in the module. The module exports only a type `Obj` that is also declared globally. `Obj` indexes a collection of state records of type `ObjRec` stored in the module's `objs` variable, which is a function `Obj->ObjRec`. `Obj`'s methods are all the names declared at top level in the class except the variables, plus the `new` method described below; the exported `Obj`'s methods are all the ones that the class exports plus `new`.

To make a class routine suitable as a method, it needs access to an `ObjRec` that holds the state of the object. It gets this access through a `self` parameter of type `Obj`, which it uses to refer to the object state `objs(self)`. To carry out this scheme, each routine in the module, unless it appears in a `WITH` clause in the class, is 'objectified' by giving it an extra `self` parameter of type `Obj`. In addition, in a routine body every occurrence of a variable `v` declared at top level in the class is replaced by `objs(self).v` in the module, and every invocation of an objectified class routine gets `self` as an extra first parameter.

The module also gets a synthesized and objectified `StdNew` procedure that adds a state record to `objs`, initializes it from the class's variable initializations (rewritten like the routine bodies), and returns its `Obj` index; this procedure becomes the `new` method of `Obj` unless the class already has a `new` routine.

A class cannot declare a `THREAD`.

The effect of this transformation is that a variable `obj` of type `Obj` behaves like an object. The state of the object is `objs(obj)`. The invocation `obj.m` or `obj.m(x)` is short for `ObjMod.m(obj)` or `ObjMod.m(obj, x)` by the usual rule for methods, and it thus invokes the method `m`; in `m`'s body each occurrence of a class variable refers to the corresponding field in `obj`'s state. `Obj.new()` returns a new and initialized `Obj` object. The following example shows how a class is transformed into a module.


```
CLASS Obj EXPORT T1, f, p, … =

TYPE T1 = … WITH {add:=AddT}
CONST c := …

VAR  v1:T1:=ei, v2:T2:=pi(v1), …

FUNC f(p1: RT1, …) = … v1 …

PROC p(p2: RT2, …) = … v2 …

FUNC AddT(t1, t2) = …
…

END Obj
```

```
MODULE ObjMod EXPORT Obj WITH {T1, f, p, new} =

TYPE T1 = … WITH {add:=AddT}
CONST c := …

TYPE ObjRec = [v1: T1, v2: T2, …]
     Obj = Int WITH {T1, c, f:=f, p:=p,
                     AddT:=AddT, …,new:=StdNew}
VAR  objs: Obj -> ObjRec := {}

FUNC f(self: Obj, p1: RT1, …) =
    … objs(self).v1 …
PROC p(self: Obj, p2: RT2, …) =
    … objs(self).v2 …
FUNC AddT(t1, t2) = …  % in T1's WITH, so not objectified
…
PROC StdNew(self: Obj) -> Obj =
  VAR obj: Obj | ~ obj IN objs.dom =>
    objs(obj)    := ObjRec{};
    objs(obj).v1 := ei;
    objs(obj).v2 := pi(objs(obj).v1);
    …;
    RET obj

END ObjMod

TYPE Obj = ObjMod.Obj
```

In abstraction functions and invariants we also write `obj.n` for field `n` in `obj`'s state, that is, for `ObjMod.objs(obj).n`.

8. Scope

The declaration of an identifier is known throughout the smallest scope in which the declaration appears (redeclaration is not allowed). This section summarizes how scopes work in Spec; terms defined before section 7 have pointers to their definitions. A scope is one of

- the whole `program`, in which just the predefined (section 3), module, and globally declared identifiers are declared;
- a `module`;
- the part of a `routineDecl` or `LAMBDA` expression (section 5) after the `=`;
- the part of a `VAR declInit | cmd` command after the `|` (section 6);
- the part of a constructor or quantification after the first `|` (section 5).
- a record `type` or `methodDefList` (section 4);

An identifier is declared by

- a module `id`, `mfp`, or `toplevel` (for types, exception sets, `ENUM` elements, and named routines),
- a `decl` in a record `type` (section 4), `| constructor` or quantification (section 5), `declInit` (section 6), routine `signature`, or `WITH` clause of a `mfp`, or
- a `methodDef` in the `WITH` clause of a `type` (section 4).

An identifier may not be declared in a scope where it is already known. An occurrence of an identifier `id` always refers to the declaration of `id` which is known at that point, except when `id` is being declared (precedes a `:`, the `=` of a `toplevel`, the `:=` of a record constructor, or the `:=` or `BY` in a `seqGen`), or follows a dot. There are four cases for dot:

`moduleId . id` — the `id` must be exported from the basic module `moduleId`, and this expression denotes the meaning of `id` in that module.

`record . id` — the `id` must be declared as a field of the record type, and this expression denotes that field of `record`. In an `assignment`'s `lhs` see [7] in section 6 for the meaning.

`typeId . id` — the `typeId` denotes a type, `id` must be a method of this type, and this expression denotes that method.

`primary . id` — the `id` must be a method of `primary`'s type, and this expression, together with any following arguments, denotes an invocation of that method; see [2] in section 5 on expressions.

If `id` refers to an identifier declared by a `toplevel` in the current module `m`, it is short for `m.id`. If it refers to an identifier declared by a `toplevel` in the `program`, it is short for `Global.id`. Once these abbreviations have been expanded, every name in the state is either global (contains a dot and is declared in a `toplevel`), or local (does not contain a dot and is declared in some other way).

Exceptions look like identifiers, but they are actually string literals, written without the enclosing quotes for convenience. Therefore they do not have scope.

9. Built-in methods

Some of the type constructors have built-in methods, among them the operators defined in the expression grammar. The built-in methods for types other than `Int` and `Bool` are defined below. Note that these are not complete definitions of the types; they do not include the constructors.

Sets

A set has methods for

computing union, intersection, and set difference (lifted from `Bool`; see note 3 in section 4), and adding or removing an element, testing for membership and subset;

choosing (deterministically) a single element from a set, or a sequence with the same members, or a maximum or minimum element, and turning a set into its characteristic predicate (the inverse is the predicate's `set` method);

composing a set with a function or relation, and converting a set into a relation from `nil` to the members of the set (the inverse of this is just the range of the relation).

We define these operations with a module that represents a set by its characteristic predicate. Precisely, `SET T` behaves as though it were `Set[T].S`, where


```
MODULE Set[T] EXPORT S =

TYPE S = Any->Bool SUCHTHAT (ALL any | s(any) ==> (any IS T))
% Defined everywhere so that type inclusion will work; see section 4.
                    WITH {"\/":=Union, "/\":=Intersection, "-":=Difference,
                          "IN":=In, "<=":=Subset, choose:=Choose, seq:=Seq,
                           pred:=Pred, rel:=Rel, id:=Id,univ:=Univ,include:=Incl,
                           perms:=Perms,fsort:=FSort,sort:=Sort,combine:=Combine,
                           fmax:=FMax, fmin:=FMin, max:=Max, min:=Min
                           "*":=ComposeF, "**":=ComposeR }

FUNC Union(s1, s2)->S        = RET (\ t | s1(t) \/  s2(t))  %s1 \/ s2
FUNC Intersection(s1, s2)->S = RET (\ t | s1(t) /\  s2(t))  %s1 /\ s2
FUNC Difference(s1, s2)->S   = RET (\ t | s1(t) /\ ~s2(t))  %s1 - s2
FUNC In(s, t)->Bool          = RET s(t)                     %t IN s
FUNC Subset(s1, s2)->Bool    = RET (ALL t| s1(t) ==> s2(t)) %s1 <= s2
FUNC Size(s)->Int            =                              %s.size
    VAR t | s(t) => RET Size(s-{t}) + 1 [*] RET 0
FUNC Choose(s)->T            = VAR t | s(t) => RET t        %s.choose
% Not really, since VAR makes a non-deterministic choice,
% but choose makes a deterministic one. It is undefined if s is empty.
FUNC Seq(s)->SEQ T           =                              %s.seq
% Defined only for finite sets. Note that Seq chooses a sequence deterministically.
    RET {q: SEQ T | q.rng = s /\ q.size = s.size}.choose

FUNC Pred(s)->(T->Bool)      = RET s                        %s.pred
% s.pred is just s. We define pred for symmetry with seq, set, etc.
FUNC Rel(s)->(Bool->>T)      = s.pred.inv
FUNC Id(s)->(T->>T)          = RET {t :IN s || (t, t)}.pred.pToR
FUNC Univ(s)->(T->>T)        = s.rel.inv * s.rel
FUNC Incl(s)->(SET T->>T)    = (\ st: SET T, t | t IN (st /\ s)).pToR

FUNC Perms(s)->SET SEQ T         = RET s.seq.perms          %s.perms
FUNC FSort(s, f: (T,T)->Bool)->S = RET s.seq.fsort(f)       %s.fsort(f); f is compare
FUNC Sort(s)->S                  = RET s.seq.sort           %s.sort; only if T has <=
FUNC Combine(s, f: (T,T)->T)->T  = RET s.seq.combine(f)     %useful if f is commutative
FUNC FMax(s, f: (T,T)->Bool)->T  = RET s.fsort(f).last      %s.fmax(f); a max under f
FUNC FMin(s, f: (T,T)->Bool)->T  = RET s.fsort(f).head      %s.fmin(f); a min under f
FUNC Max(s)->T                   = RET s.fmax(T."<=")       %s.max; only if T has <=
FUNC Min(s)->T                   = RET s.fmin(T."<=")       %s.min; only if T has <=
% Note that these functions are undefined if s is empty. If there are extremal elements not distinguished by f or "<=",
% they make an arbitrary deterministic choice. To get all the choices, use T.f.rel.leaves.
% Note that this is not the same as /\ : s, unless s is totally ordered.
FUNC ComposeF(s, f: T->U)->SET U = RET {t :IN s || f(t)}    %s * f; image of s under f
% ComposeF like sequences, pointwise on the elements. ComposeF(s, f) = ComposeR(s, f.rel)
FUNC ComposeR(s, r:T->>U)->SET U = RET (s.rel * r).rng      %s ** r; image of s under r
% ComposeR is relational composition: anything you can get to by r, starting with a member of s.
% We could have written it explicitly: {t :IN s, u | r(t, u) || u}, or as \/ : (s * r.setF).

END Set
```

There are constructors `{}` for the empty set, `{e1, e2, ...}` for a set with specific elements, and `{declList | pred || exp}` for a set whose elements satisfy a predicate. These constructors are described in [6] and [10] of section 5. Note that `{t | p}.pred = (\ t | p)`, and similarly `(\ t | p).set = {t | p}`. A method on `T` is lifted to a method on `S`, unless the name conflicts with one of `S`'s methods, exactly like lifting on `S.rel`; see note 3 in section 4.

Functions

The function types `T->U` and `T->U RAISES XS` have methods for

composition, overlay, inverse, and restriction;

testing whether a function is defined at an argument and whether it produces a normal (non-exceptional) result at an argument, and for the domain and range;

converting a function to a relation (the inverse is the relation's `func` method) or a function that produces a set to a relation with each element of the set (`setRel`; the inverse is the relation's `setF` method).

In other words, they behave as though they were `Function[T, U].F`, where (making allowances for the fact that `XS` and `V` are pulled out of thin air):

MODULE Function[T, U] EXPORT F =

```
TYPE F = T->U RAISES XS WITH {"*":=Compose, "+":=Overlay,
                              inv:=Inverse, restrict:=Restrict,
                              "!":=Defined, "!!":=Normal,
                              dom:=Domain, rng:=Range, rel:=Rel, setRel:=SetRel}
     R = (T, U) -> Bool

FUNC Compose(f, g: U -> V) -> (T -> V) = RET (\ t | g(f(t)))
% Note that the order of the arguments is reversed from the usual mathematical convention.

FUNC Overlay(f1, f2) -> F = RET (\ t | (f2!t => f2(t) [*] f1(t)))
% (f1 + f2) is f2(x) if that is defined, otherwise f1(x)

FUNC Inverse(f) -> (U -> T) = RET f.rel.inv.func
FUNC Restrict(f, s: SET T) -> F = (s.id * f).func

FUNC Defined(f, t)->Bool =
     IF f(t)=f(t) => RET true [*] RET false FI EXCEPT XS => RET true

FUNC Normal(f, t)->Bool = t IN f.dom

FUNC Domain(f) -> SET T = f.rel.dom
FUNC Range (f) -> SET U = f.rel.rng

FUNC Rel(f) -> R = RET (\ t, u | f(t) = u).pToR
FUNC SetRel(f) -> ((T, V)->Bool) = RET (\ t, v | (f!t ==> v IN f(t) [*] false) )
% if U = SET V, f.setRel relates each t in f.dom to each element of f(t).

END Function
```

Note that there are constructors `{}` for the function undefined everywhere, `T{* -> result}` for a function of type `T` whose value is `result` everywhere, and `f{exp -> result}` for a function which is the same as `f` except at `exp`, where its value is `result`. These constructors are described in [6] and [8] of section 5. There are also lambda constructors for defining a function by a computation, described in [9] of section 5. A method on `U` is lifted to a method on `F`, unless the name conflicts with a method of `F`; see note 3 in section 4.

Functions declared with more than one argument take a single argument that is a tuple. So `f(x: Int)` takes an `Int`, but `f(x: Int, y: Int)` takes a tuple of type `(Int, Int)`. This convention keeps the tuples in the background as much as possible. The normal syntax for calling a function is `f(x, y)`, which constructs the tuple `(x, y)` and passes it to `f`. However, `f(x)` is treated differently, since it passes `x` to `f`, rather than the singleton tuple `{x}`. If you have a tuple `t` in hand, you can pass it to `f` by writing `f$t` without having to worry about the singleton case; if `f` takes only one argument, then `t` must be a singleton tuple and `f$t` will pass `t(0)` to `f`. Thus `f$(x, y)` is the same as `f(x, y)` and `f${x}` is the same as `f(x)`.

A function declared with names for the arguments, such as

```
(\ i: Int, s: String | i + StringToInt(x))
```

has a type that ignores the names, `(Int, String)->Int`. However, it also has a method `argNames` that returns the sequence of argument names, `{"i", "s"}` in the example, just like a record. This makes it possible to match up arguments by name.

A total function `T->Bool` is a predicate and has an additional method to compute the set of `T`'s that satisfy the predicate (the inverse is the set's `pred` method). In other words, a predicate behaves as though it were `Predicate[T].P`, where

MODULE Predicate[T] EXPORT P =

```
TYPE P = T -> Bool WITH {set:=Set, pToR:=PToR}
FUNC Set(p) -> SET T = RET {t | p(t)}
END Predicate
```

A predicate with `T = (U, V)` defines a relation `U ->> V` by

```
FUNC PToR(p: (U, V)->Bool) -> (U ->> V) = RET (\u | {v | p(u, v)}).setRel
```

It has additional methods to turn it into a function `U -> V` or a function `U -> SET V`, and to get its domain and range, invert it or compose it (overriding the methods for a function). In other words, it behaves as though it were `Relation[U, V].R`, where (allowing for the fact that `W` is pulled out of thin air in `Compose`):

MODULE Relation[U, V] EXPORT R =

```
TYPE R = (U, V) -> Bool WITH {pred:=Pred, set:=R.rng, restrict:=Restrict,
                              fun:=Fun, setF:=SetFunc, dom:=Domain, rng :=Range,
                              inv:=Inverse, "*":=Compose}

FUNC Pred(r) -> ((U,V)->Bool) = RET r(u, v)
FUNC Restrict(r, s) -> R = RET s.id * r

FUNC Fun(r) -> (U -> V) =                                  % defined at u iff r relates u to a single
     RET (\ u | (r.setF(u).size = 1 => r.setF(u).choose))
FUNC SetFunc(r) -> (U -> SET V) = RET (\ u | {v | r(u, v)})
% SetFunc(r) is defined everywhere, returning the set of V's related to u.

FUNC Domain(r) -> SET U = RET {u, v | r(u, v) || u}
FUNC Range (r) -> SET V = RET {u, v | r(u, v) || v}

FUNC Inverse(r) -> ((V, U) -> Bool) = RET (\ v, u | r(u, v))
FUNC Compose(r: R, s: (V, W)->Bool) -> (U, W)->Bool = % r * s
     RET (\ u, w | (EXISTS v | r(u, v) /\ s(v, w)) )

END Relation
```

A method on `V` is lifted to a method on `R`, unless there's a name conflict; see note 3 in section 4.

A relation with `U = V` is a graph and has additional methods to yield the sequences of `U`'s that are paths in the graph, and to compute the transitive closure and its restriction to exit nodes. In other words, it behaves as though it were `Graph[U].G`, where

```
MODULE Graph[T] EXPORT G =

TYPE G = T ->> T WITH {paths:=Paths, closure:=Closure, leaves:=Leaves }
     P = SEQ T

FUNC Paths(g) -> SET P = RET {p | (ALL i :IN p.dom - {0} | (g.pred)(p(i-1), p(i))
% Any p of size <= 1 is a path by this definition.
FUNC Closure(g) -> G = RET (\ t1, t2 |
     (EXISTS p | p.size > 1 /\ p.head = t1 /\ p.last = t2 /\ p IN g.paths ))
FUNC Leaves(g) -> G = RET g.closure * (g.rng - g.dom).id

END Graph
```

Records and tuples

A record is a function from the string names of its fields to the field values, and an *n*-tuple is a function from `0..n-1` to the field values. There is special syntax for declaring records and tuples, and for reading and writing record fields:

`[f: T, g: U]` declares a record with fields `f` and `g` of types `T` and `U`. It is short for

```
String->Any WITH { fields:=(\r: String->Any | (SEQ String){"f", "g"}) }
             SUCHTHAT   this.dom >= {"f", "g"}
                        /\ this("f") IS T /\ this("g") IS U
```

Note the `fields` method, which gives the sequence of field names `{"f", "g"}`.

`(T, U)` declares a tuple with fields of types `T` and `U`. It is short for

```
Int->Any WITH { fields:=(\r: nt->Any | 0..1) }
             SUCHTHAT   this.dom >= 0..1
                        /\ this(0) IS T /\ this(1) IS U
```

Note the `fields` method, which gives the sequence of field names `0..1`.

`r.f` is short for `r("f")`, and `r.f := e` is short for `r := r{"f"->e}`.

There is no special syntax for tuple fields, since you can just write `t(2)` and `t(2) := e` to read and write the third field, for example (remember that fields are numbered from 0).

Thus to convert a record `r` into a tuple, write `r.fields * r`, and to convert a tuple `t` into a record, write `r.fields.inv * t`.

There is also special syntax for constructing record and tuple values, illustrated in the following example. Given the type declaration

```
TYPE Entry = [salary: Int, birthdate: String]
```

we can write a record value

```
Entry{salary := 23000, birthdate := "January 3, 1955"}
```

which is short for the function constructor

```
Entry{"salary" -> 23000, "birthdate" -> "January 3, 1955"}.
```

The constructor `(`

```
23000, "January 3, 1955")
```

yields a tuple of type `(Int, String)`. It is short for

```
{0 -> 23000, 1 -> "January 3, 1955"}
```

This doesn't work for a singleton tuple, since `(x)` has the same value as `x`. However, the sequence constructor `{x}` will do for constructing a singleton tuple, since a singleton `SEQ T` has the type `(T)`.

Sequences

A function is called a sequence if its domain is a finite set of consecutive `Int`'s starting at 0, that is, if it has type

```
Q = Int -> T SUCHTHAT (\ q | (EXISTS size: Int | q.dom = (0 .. size-1).rng))
```

We denote this type (with the methods defined below) by `SEQ T`. A sequence inherits the methods of the function (though it overrides `+`), and it also has methods for

`head`, `tail`, `last`, `reml`, `addh`, `addl`: detaching or attaching the first or last element,
`seg`, `sub`: extracting a segment of a sequence,
`+`, `size`: concatenating two sequences, or finding the size,
`fill`: making a sequence with all elements the same,
`zip` or `||`: making a pair of sequences into a sequence of pairs
`<=`, `<<=`: testing for prefix or sub-sequence (not necessarily contiguous),
`**`: composing with a relation (`SEQ T` inherits composing with a function),
lexical comparison, permuting, and sorting,
`iterate`, `combine`: iterating a function over each prefix of a sequence, or the whole sequence
treating a sequence as a multiset, with operations to:
count the number of times an element appears, test membership and multiset equality, take differences, and remove an element (`"+"` or `"\/"` is union and `addl` adds an element).

All these operations are undefined if they use out-of-range subscripts, except that a sub-sequence is always defined regardless of the subscripts, by taking the largest number of elements allowed by the size of the sequence.

We define the sequence methods with a module. Precisely, `SEQ T` is `Sequence[T].Q`, where:

```
MODULE Sequence[T] EXPORTS Q =

TYPE I          =  Int
     Q          =  (I -> T) SUCHTHAT q.dom = (0 .. q.size-1).rng
                   WITH { size:=Size, seg:=Seg, sub:=Sub, "+":=Concatenate,
                          head:=Head, tail:=Tail, addh:=AddHead, remh:=Tail,
                          last:=Last, reml:=RemoveLast, addl:=AddLast,
                          fill:=Fill, zip:=Zip, "||":=Zip,
                          "<=":=Prefix, "<<=":=SubSeq,
                          "**":=ComposeR, lexLE:=LexLE, perms:=Perms,
                          fsorter:=FSorter, fsort:=FSort, sort:=Sort,
                          iterate:=Iterate, combine:=Combine,

                      % These methods treat a sequence as a multiset (or bag).
                          count:=Count, "IN":=In, "==":=EqElem,
                          "\/":=Concatenate, "-":=Diff, set:=Q.rng }

FUNC Size(q)-> Int  = RET q.dom.size

FUNC Sub(q, i1, i2) -> Q =
% q.sub(i1, i2); yields {q(i1),...,q(i2)}, or a shorter sequence if i1 < 0 or i2 >= q.size
     RET ({0, i1}.max .. {i2, q.size-1}.min) * q

FUNC Seg(q, i, n: I) -> Q = RET q.sub(i, i+n-1)        % q.seg(i,n); n I's from q(i)

FUNC Concatenate(q1, q2) -> Q = VAR q |                % q1 + q2
     q.sub(0, q1.size-1) = q1 /\ q.sub(q1.size, q.size-1) = q2 => RET q

FUNC Head(q) -> T = RET q(0)                           % q.head; first element
```

```
FUNC Tail(q) -> Q =                                  % q.tail;  all but first
    q.size > 0 => RET q.sub(1, q.size-1)
FUNC AddHead(q, t) -> Q = RET {t} + q                % q.addh(t)

FUNC Last(q) -> T = RET q(q.size-1)                  % q.last; last element
FUNC RemoveLast(q) -> Q =                            % q.reml; all but last
    q # {} => RET q.sub(0, q.size-2)
FUNC AddLast(q, t) -> Q = RET q + {t}                % q.addl(t)

FUNC Fill(t, n: I) -> Q = RET {i :IN 0 .. n-1 || t}  % yields n copies of t

FUNC Zip(q, qU: SEQ U) -> SEQ (T, U) =               % size is the min
    RET (\ i | (i IN (q.dom /\ qU.dom) => (q(i), qU(i))))

FUNC Prefix(q1, q2) -> Bool =                        % q1 <= q2
    RET (EXISTS q | q1 + q = q2)

FUNC SubSeq(q1, q2) -> Bool =                        % q1 <<= q2
% Are q1's elements in q2 in the same order, not necessarily contiguously.
    RET (EXISTS p: SET Int | p <= q2.dom /\ q1 = p.seq.sort * q2)

FUNC ComposeR(q, r: (T, U)->Bool) -> SEQ U =         % q ** r
% Elements related to nothing are dropped. If an element is related to several things, they appear in arbitrary order.
    RET + : (q * r.setF * (\s: SET U | s.seq))

FUNC LexLE(q1, q2, f: (T,T)->Bool) -> Bool =         % q1.lexLE(q2, f);  f is <=
% Is q1 lexically less than or equal to q2. True if q1 is a prefix of q2,
% or the first element in which q1 differs from q2 is less.
    RET    q1 <= q2
        \/ (EXISTS i :IN q1.dom /\ q2.dom |   q1.sub(0, i-1) = q2.sub(0, i-1)
                                          /\ q1(i) # q2(i)) /\ f(q1(i), q2(i))

FUNC Perms(q)->SET Q =                               % q.perms
    RET {q' | (ALL t | q.count(t) = q'.count(t))}

FUNC FSorter(q, f: (T,T)->Bool)->SEQ Int =           % q.fsorter(f); f is <=
% The permutation that sorts q stably. Note: can't use min to define this, since min is defined using sort.
    VAR  ps := {p :IN q.dom.perms                    % all perms that sort q
              | (ALL i :IN (q.dom - {0}) | f((p*q)(i-1), (p*q)(i))) } |
      VAR p0 :IN ps |                                % the one that reorders the least
         (ALL p :IN ps | p0.lexLE(p, Int."<=")) => RET p0

FUNC FSort(q, f: (T,T)->Bool) -> Q =                 % q.fsort(f); f is <= for the sort
    RET q.fsorter(f) * q
FUNC Sort(q)->Q = RET q.fsort(T."<=")                % q.sort; only if T has <=

FUNC Iterate(q, f: (T,T)->T) -> Q =                  % q.iterate(f)
% Yields qr = {q(0), qr(0) + q(1), qr(1) + q(2), ...), where t1 + t2 is f(t1, t2)
    RET {qr: Q |   qr.size=q.size /\ qr(0) = q(0)
                /\ (ALL i IN q.dom-{0} | qr(i) = f(qr(i-1),q(i)))}.one
FUNC Combine(q, f: T,T)->T) -> T = RET q.iterate(f).last
% Yields q(0) + q(1) + ..., where t1 + t2 is f(t1, t2)

FUNC Count(q, t)->Int = RET {t' :IN q | t' = t}.size % q.count(t)
FUNC In(t, q)->Bool = RET (q.count(t) # 0)           % t IN q
FUNC EqElem(q1, q2) -> Bool = RET q1 IN q2.perms     % q1 == q2; equal as multisets
FUNC Diff(q1, q2) -> Q =                             % q1 - q2
    RET {q | (ALL t | q.count(t) = {q1.count(t) - q2.count(t), 0}.max)}.choose

END Sequence
```


A sequence is a special case of a tuple, in which all the elements have the same type.

`Int` has a method `..` for making sequences: `i .. j = {i, i+1, ..., j-1, j}`. If `j < i`, `i .. j = {}`. You can also write `i .. j` as `{k := i BY k + 1 WHILE k <= j}`; see [11] in section 5. `Int` also has a `seq` method: `i.seq = 0 .. i-1`.

There is a constructor `{e1, e2, ...}` for a sequence with specific elements and a constructor `{}` for the empty sequence. There is also a constructor `q{e1 -> e2}`, which is equal to `q` except at `e1` (and undefined if `e1` is out of range). For the constructors see [6] and [8] of section 5. To generate a sequence there are constructors `{x :IN q | pred || exp}` and `{x := e1 BY e2 WHILE pred1 | pred2 || exp}`. For these see [11] of section 5.

To map each element `t` of `q` to `f(t)` use function composition `q * f`. Thus if `q: SEQ Int`, `q * (\ i: Int | i*i)` yields a sequence of squares. You can also write this `{i :IN q || i*i}`.

Index

5. Examples of Specs and Code

This handout is a supplement for the first two lectures. It contains several example specs and code, all written using Spec.

Section 1 contains a spec for sorting a sequence. Section 2 contains two specs and one code for searching for an element in a sequence. Section 3 contains specs for a read/write memory. Sections 4 and 5 contain code for a read/write memory based on caching and hashing, respectively. Finally, Section 6 contains code based on replicated copies.

1. Sorting

The following spec describes the behavior required of a program that sorts sets of some type `T` with a `"<="` comparison method. We do not assume that `"<="` is antisymmetric; in other words, we can have `t1 <= t2` and `t2 <= t1` without having `t1 = t2`, so that `"<="` is not enough to distinguish values of `T`. For instance, `T` might be the record type `[name:String, salary: Int]` with `"<="` comparison of the `salary` field. Several `T`'s can have different `name`s but the same `salary`.

```
TYPE S = SET T
     Q = SEQ T

APROC Sort(s) -> Q = <<
     VAR q | (ALL t | s.count(t) = q.count(t)) /\ Sorted(q) => RET q >>
```

This spec uses the auxiliary function `Sorted`, defined as follows.

```
FUNC Sorted(q) -> Bool = RET (ALL i :IN q.dom - {0} |  q(i-1) <= q(i))
```

If we made `Sort` a `FUNC` rather than a `PROC`, what would be wrong?[1] What could we change to make it a `FUNC`?

We could have written this more concisely as

```
APROC Sort(s) -> Q =
     << VAR q :IN a.perms | Sorted(q) => RET q >>
```

using the `perms` method for sets that returns a set of sequences that contains all the possible permutations of the set.

[1] Hint: a `FUNC` can't have side effects and must be deterministic (return the same value for the same arguments).

2. Searching

Search spec

We begin with a spec for a procedure to search an array for a given element. Again, this is an APROC rather than a FUNC because there can be several allowable results for the same inputs.

```
APROC Search(q, t) -> Int RAISES {NotFound} =
    << IF  VAR i: Int | (0 <= i /\ i < q.size /\ q(i) = t) => RET i
       [*] RAISE NotFound
       FI >>
```

Or, equivalently but slightly more concisely, and highlighting the changes with boxes:

```
APROC Search(q, t) -> Int RAISES {NotFound} =
    << IF VAR i :IN q.dom | q(i) = t => RET i [*] RAISE NotFound FI >>
```

Sequential search code

Here is code for the Search spec given above. It uses sequential search, starting at the first element of the input sequence.

```
APROC SeqSearch(q, t) -> Int RAISES {NotFound} = << VAR i := 0 |
    DO i < q.size => IF q(i) = t => RET i [*] i + := 1 FI OD; RAISE NotFound >>
```

Alternative search spec

Some searching algorithms, for example, binary search, assume that the input argument sequence is sorted. Such algorithms require a different spec, one that expresses this requirement.

```
APROC Search1(q, t) -> Int RAISES {NotFound} = <<
    IF  ~Sorted(q) => HAVOC
    [*] VAR i :IN q.dom | q(i) = t => RET i
    [*] RAISE NotFound
    FI >>
```

Alternatively, the requirement could go in the type of the q argument:

```
APROC Search1(q: Q SUCHTHAT Sorted(this), t) -> Int RAISES {NotFound} = <<
    ... >>
```

This is farther from code, since proving that a sequence is sorted is likely to be too hard for the code's compiler.

You might consider writing the spec to raise an exception when the array is not sorted:

```
APROC Search2(q, t) -> Int RAISES {NotFound, NotSorted} = <<
    IF  ~Sorted(q) => RAISE NotSorted
    ...
```

This is not a good idea. The whole point of binary search is to obtain O(log *n*) time performance (for a sorted input sequence). But any code for the Search2 spec requires an O(*n*) check, even for a sorted input sequence, in order to verify that the input sequence is in fact sorted.

This is a simple but instructive example of the difference between defensive programming and efficiency. If Search were part of an operating system interface, it would be intolerable to have HAVOC as a possible transition, because the operating system is not supposed to go off the deep

end no matter how it is called (though it might be OK to return the wrong answer if the input isn't sorted; what would that spec be?). On the other hand, the efficiency of a program often depends on assumptions that one part of it makes about another, and it's appropriate to express such an assumption in a spec by saying that you get HAVOC if it is violated. We don't care to be more specific about what happens because we intend to ensure that it doesn't happen. Obviously a program written in this style will be more prone to undetected or obscure errors than one that checks the assumptions, as well as more efficient.

3. Read/write memory

The simplest form of read/write memory is a single read/write register, say of type V (for value), with arbitrary initial value. The following Spec module describes this (a lot of boilerplate for a simple variable, but we can extend it in many interesting ways):

```
MODULE Register [V] EXPORT Read, Write =

VAR m: V                                                    % arbitrary initial value

APROC Read() -> V = << RET m  >>
APROC Write(v)    = << m := v >>

END Register
```

Now we give a spec for a simple addressable memory with elements of type V. This is like a collection of read/write registers, one for each address in a set A. In other words, it's a function from addresses to data values. For variety, we include new Reset and Swap operations in addition to Read and Write.

```
MODULE Memory [A, V] EXPORT Read, Write, Reset, Swap =

TYPE M =  A -> V
VAR  m := Init()

APROC Init() -> M  = << VAR m' | (ALL a | m'!a) => RET m' >>
% Choose an arbitrary function that is defined everywhere.

FUNC  Read(a) -> V = << RET m(a)  >>
APROC Write(a, v)  = << m(a) := v >>

APROC Reset(v)     = << m := M{* -> v} >>
% Set all memory locations to v.

APROC Swap(a, v) -> V = << VAR v' := m(a) | m(a) := v; RET v' >>
% Set location a to the input value and return the previous value.

END Memory
```

The next three sections describe code for Memory.

4. Write-back cache code

Our first code is based on two memory mappings, a main memory `m` and a *write-back cache* `c`. The code maintains the invariant that the number of addresses at which `c` is defined is constant. A real cache would probably maintain a weaker invariant, perhaps bounding the number of addresses at which `c` is defined.

```
MODULE WBCache [A, V] EXPORT Read, Write, Reset, Swap =
% implements Memory

TYPE M          =  A -> V
     C          =  A -> V

CONST Csize     :  Int := ...                          % cache size

VAR  m          := InitM()
     c          := InitC()

APROC InitM() -> M = << VAR m' | (ALL a | m'!a)       => RET m' >>
% Returns a M with arbitrary values.

APROC InitC() -> C = << VAR c' | c'.dom.size = CSize => RET c' >>
% Returns a C that has exactly CSize entries defined, with arbitrary values.

APROC Read(a) -> V = << Load(a); RET c(a) >>
APROC Write(a, v) = << IF ~c!a => FlushOne() [*] SKIP FI; c(a) := v >>
% Makes room in the cache if necessary, then writes to the cache.

APROC Reset(v) = <<...>>                               % exercise for the reader

APROC Swap(a, v) -> V = << Load(a); VAR v' | v' := c(a); c(a) := v; RET v' >>

% Internal procedures.

APROC Load(a) = << IF ~c!a => FlushOne(); c(a) := m(a) [*] SKIP FI >>
% Ensures that address a appears in the cache.

APROC FlushOne() =
% Removes one (arbitrary) address from the cache, writing the data value back to main memory if necessary.
     << VAR a | c!a => IF Dirty(a) => m(a) := c(a) [*] SKIP FI; c := c{a -> } >>

FUNC Dirty(a) -> Bool = RET c!a /\ c(a) # m(a)
% Returns true if the cache is more up-to-date than the main memory.

END WBCache
```

The following Spec function is an abstraction function mapping a state of the `WBCache` module to a state of the `Memory` module. Unlike our usual practice we have written it explicitly as a function from the state of `HashMemory` to the state of `Memory`. It says that the contents of location `a` is `c(a)` if `a` is in the cache, and `m(a)` otherwise.

```
FUNC AF(m, c) -> M = RET (\ a | c!a => c(a) [*] m(a) )
```

We could have written this more concisely as

```
FUNC AF(m, c) -> M = RET m + c
```

That is, override the function `m` with the function `c` wherever `c` is defined.

5. Hash table code

Our second code for `Memory` uses a hash table for the representation. It is different enough from the spec that it wouldn't be helpful to highlight the changes.

```
MODULE HashMemory [A WITH {hf: A->Int}, V] EXPORT Read, Write, Reset, Swap =
% Implements Memory.  Expects that the hash function A.hf is total and that its range is 0 .. n for some n.

TYPE Pair       =  [a, v]
     B          =  SET Pair                            % Bucket in hash table
     HashT      =  SEQ B

CONST nb        := A.hf.rng.max                        % Number of buckets

VAR  m: HashT   := {i :IN 1 .. nb | {}}                % Memory hash table; initially empty
     default    :  V                                   % default value, initially arbitrary

APROC Read(a) -> V = <<
     VAR p :IN m(a.hf) | p.a = a => RET p.v [*] RET default >>

APROC Write(a, v) = << VAR b := m(a.hf) |
     IF  VAR p :IN b | p.a = a => b := b - {p}         % remove a pair with a from b
     [*] SKIP FI;                                     % do nothing if there isn't one
     m(a.hf) := b \/ {Pair{a, v}} >>                  % and add the new pair

APROC Reset(v) = << m := {i :IN 1 .. nb | {}}; default := v >>

APROC Swap(a, v) -> V = << VAR v' | v' := Read(a); Write(a, v); RET v' >>

END HashMemory
```

The following is a key invariant that holds between invocations of the operations of `HashMemory`:

```
FUNC Inv(m: hashT, nb: Int) -> Bool = RET
     (  m.size = nb
     /\ (ALL i :IN m.dom, p :IN m(i) | p.a.hf = i)
     /\ (ALL a | { p :IN m(a.hf) | p.a = a }.size <= 1) )
```

This says that the hash function maps all addresses to actual buckets, that a pair containing address `a` appears only in the bucket at index `a.hf` in `m`, and that at most one pair for an address appears in the bucket for that address. Note that these conditions imply that in any reachable state of `HashMemory`, each address appears in at most one pair in the entire memory.

The following Spec function is an abstraction function between states of the `HashMemory` module and states of the `Memory` module.

```
FUNC AF(m: HashT, default) -> M = RET
     (LAMBDA(a) -> V =
         IF  VAR i :IN m.dom, p :IN m(i) | p.a = a => RET p.v
         [*] RET default FI)
```

That is, the data value for address `a` is any value associated with address `a` in the hash table; if there is none, the data value is the default value. Spec says that a function is undefined at an argument if its body can yield more than one result value. The invariants given above ensure that the `LAMBDA` is actually single-valued for all the reachable states of `HashMemory`.

Of course `HashMemory` is not fully detailed code. In particular, it just treats the variable-length buckets as sets and doesn't explain how to maintain them, which is usually done with a linked list. However, the code does capture all the essential details.

6. Replicated memory

Our final code is based on some number `k ≥ 1` of copies of each memory location. Initially, all copies have the same default value. A `Write` operation only modifies an arbitrary *majority* of the copies. A `Read` reads an arbitrary majority, and selects and returns the most recent of the values it sees. In order to allow the `Read` to determine which value is the most recent, each `Write` records not only its value, but also a sequence number. The crucial property of a majority is that any two majorities have a non-empty intersection; this ensures that a read will see at least one copy written by the most recent write.

For simplicity, we just show the module for a single read/write register. The constant `k` determines the number of copies.

```
MODULE MajReg [V] =                                   % implements Register

CONST k          =  5                                 % 5 copies

TYPE N           =  Nat
     C           =  IN 1 .. k                         % copies, ints between 1 and k
     Maj         =  SET C SUCHTHAT maj.size > k/2     % all majority subsets of C

TYPE P           =  [v, n] WITH {"<=":=PLEq}          % Pair of value and sequence number
     M           =  C -> P                            % Memory (really register) copies
     S           =  SET P

VAR  default     :  V                                 % arbitrary initial value
     m           := M{* -> P{v := default, n := 0}}

APROC Read() -> V = << RET ReadPair().v >>

APROC Write(v) = << VAR n:= ReadPair().n, maj |
% Determines the highest sequence number n, then writes v paired with n+1 to some majority maj of the copies.
     n := n + 1;
     DO VAR j :IN maj | m(j).n # n => m(j) := {v, n} OD >>

% Internal procedures.

APROC ReadPair() -> P = << VAR s := ReadMaj() |
% Returns a pair with the largest sequence number from some majority of the copies.
     VAR p :IN s | p.n = s.max.n => RET p >>

APROC ReadMaj () -> S = << VAR maj | RET maj * m >>
% Returns the set of pairs belonging to some majority of the copies. maj * m  is {c :IN maj || m(c)}

FUNC PLeq(p1, p2) -> Bool = RET p1.n <= p2.n

END MajReg
```

The following is a key invariant for `MajReg`.

```
FUNC Inv(m) -> Bool = RET
        (ALL p :IN m.rng, p' :IN m.rng | p.n = p'.n ==> p.v = p'.v)
     /\ (EXISTS maj | m.rng.max <= (maj * m).min)
```

The first conjunct says that any two pairs with the same sequence number also have the same data. The second says that for some majority of the copies every pair has the highest sequence number.

The following Spec function is an abstraction function. It says that the abstract register data value is the data component of a copy with the highest sequence number. Again, because of the invariants, there is only one `p.v` that will be returned.

```
FUNC AF(m) -> V = RET m.rng.max.v
```

We could have written the body of `ReadPair` as

```
    << VAR s := ReadMaj() | RET s.max >>
```

except that `max` always returns the same maximal `p` from the same `s`, whereas the `VAR` in `ReadPair` chooses one non-deterministically.

6. Abstraction Functions and Invariants

This handout describes the main techniques used to prove correctness of state machines: abstraction functions and invariant assertions. We demonstrate the use of these techniques for some of the `Memory` examples from handout 5.

Throughout this handout, we consider modules all of whose externally invocable procedures are `APROC`s. We assume that the body of each such procedure is executed all at once. Also, we do not consider procedures that modify global variables declared outside the module under consideration.

Modules as state machines

Our methods apply to an arbitrary state machine or automaton. In this course we use a Spec module to define a state machine. Each state of the automaton designates values for all the variables declared in the module. The initial states of the automaton consist of initial values assigned to all the module's variables by the Spec code. The transitions of the automaton correspond to the invocations of `APROC`s together with their result values.

An *execution fragment* of a state machine is a sequence of the form $s_0, \pi_1, s_1, \pi_2, \ldots$, where each s is a state, each π is a label for a transition (an invocation of a procedure), and each consecutive $(s_i, \pi_{i+1}, s_{i+1})$ triple follows the rules specified by the Spec code. (We do not define these rules precisely here—wait for the lecture on atomic semantics.) An *execution* is an execution fragment that begins in an initial state.

The π_i are labels for the transitions; we often call them *actions*. When the state machine is written in Spec, each transition is generated by some atomic command, and we can use some unambiguous identification of the command for the action. At the moment we are studying sequential Spec, in which every transition is the invocation of an exported atomic procedure. We use the name of the procedure, the arguments, and the results as the label.

Figure 1 shows some of the states and transitions of the state machine for the `Memory` module with `A = IN 1 .. 4`, and Figure 2 does likewise for the `WBCache` module with `Csize = 2`. The arrow for each transition is labeled by its π_i, that is, by the procedure name, arguments, and result.

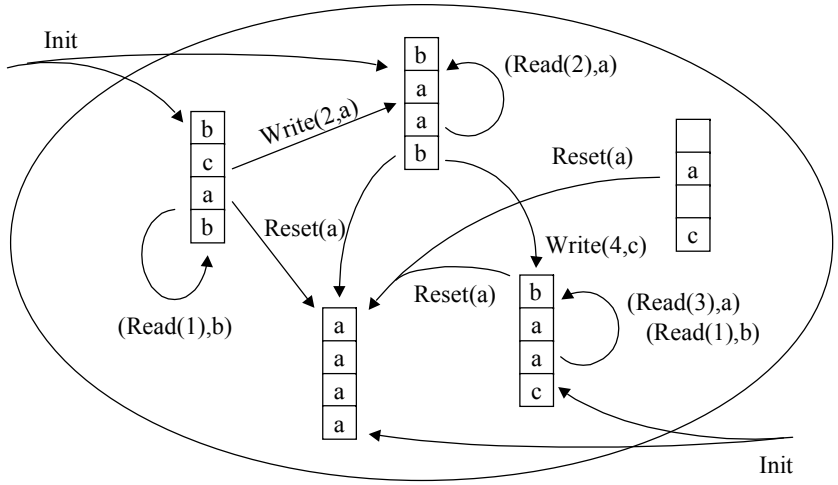


Figure 1: Part of the `Memory` state machine

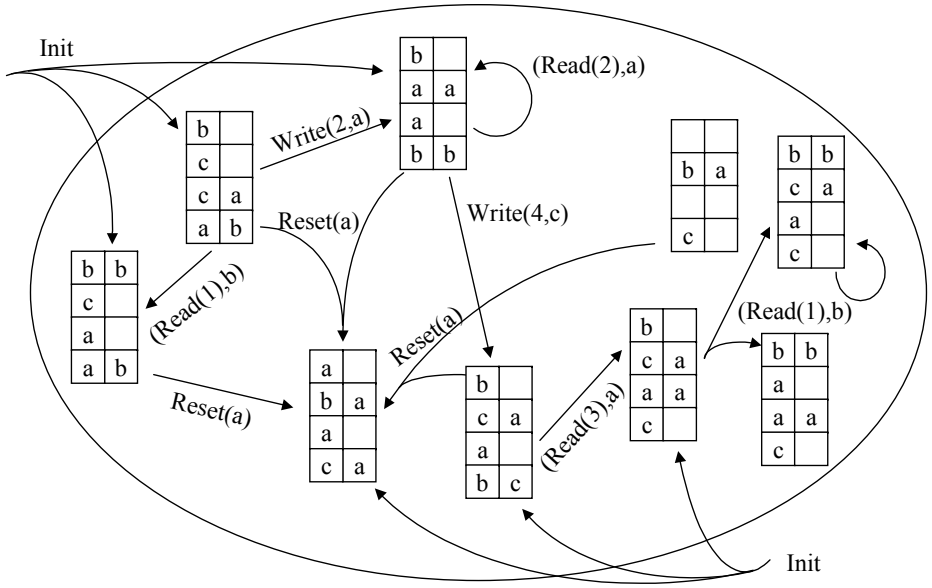


Figure 2: Part of the `WBCache` state machine

External behavior

Usually, a client of a module is not interested in all aspects of its execution, but only in some kind of external behavior. Here, we formalize the external behavior as a set of *traces*. That is, from an execution (or execution fragment) of a module, we discard both the states and the internal actions, and extract the *trace*. This is the sequence of labels π_i for external actions (that is, invocations of exported routines) that occur in the execution (or fragment). Then the external behavior of the module is the set of traces that are obtained from all of its executions.

It's important to realize that in going from the execution to the trace we are discarding a great deal of information. First, we discard all the states, keeping only the actions or labels. Second, we discard all the internal actions, keeping only the external ones. Thus the only information we keep in the trace is the behavior of the state machine at its external interface. This is appropriate, since we want to study state machines that have the same behavior at the external interface; we shall see shortly exactly what we main by 'the same' here. Two machines can have the same traces even though they have very different state spaces.

In the sequential Spec that we are studying now, a module only makes a transition when an exported routine is invoked, so all the transitions appear in the trace. Later, however, we will introduce modules with internal transitions, and then the distinction between the executions and the external behavior will be important.

For example, the set of traces generated by the `Memory` module includes the following trace:

```
(Reset(v),)
(Read(a1),v)
(Write(a2,v'))
(Read(a2),v')
```

However, the following trace is not included if `v # v'`:

```
(Reset(v))
(Read(a1),v')                    should have returned v
(Write(a2,v'))
(Read(a2),v)                     should have returned v'
```

In general, a trace is included in the external behavior of `Memory` if every `Read(a)` or `Swap(a, v)` operation returns the last value written to `a` by a `Write`, `Reset` or `Swap` operation, or returned by a `Read` operation; if there is no such previous operation, then `Read(a)` or `Swap(a, v)` returns an arbitrary value.

Implements relation

In order to understand what it means for one state machine to implement another one, it is helpful to begin by considering what it means for one atomic procedure to implement another. The meaning of an atomic procedure is a relation between an initial state just before the procedure starts (sometimes called a 'pre-state') and a final state just after the procedure has finished (sometimes called a 'post-state'). This is often called an 'input-output relation'. For example, the relation defined by a square-root procedure is that the post-state is the same as the pre-state, except that the square of the procedure result is close enough to the argument. This meaning makes sense for an arbitrary atomic procedure, not just for one in a trace.

We say that procedure *P* implements spec *S* if the relation defined by *P* (considered as a set of ordered pairs of states) is a subset of the relation defined by *S*. This means that *P* never does any-

thing that *S* couldn't do. However, *P* doesn't have to do everything that *S* can do. Code for square root is probably deterministic and always returns the same result for a given argument. Even though the spec allows several results (all the ones that are within the specified tolerance), we don't require code for to be able to produce all of them; instead we are satisfied with one.

Actually this is not enough. The definition we have given allows *P*'s relation to be empty, that is, it allows *P* not to terminate. This is usually called 'partial correctness'. In addition, we usually want to require that *P*'s relation be total on the domain of *S*; that is, *P* must produce some result whenever *S* does. The combination of partial correctness and termination is usually called 'total correctness'.

If we are only interested in external behavior of a procedure that is part of a stateless module, the only state we care about is the arguments and results of the procedure. In this case, a transition is completely described by a single entry in a trace, such as `(Read(a1),v)`.

Now we are ready to consider modules with state. Our idea is to generalize what we did with pairs of states described by single trace entries to sequences of states described by longer traces. Suppose that *T* and *S* are any modules that have the same external interface (set of procedures that are exported and hence may be invoked externally). In this discussion, we will often refer to *S* as the *spec* module and *T* as the *code*. Then we say that *T implements S* if every trace of *T* is also a trace of *S*. That is, the set of traces generated by *T* is a subset of the set of traces generated by *S*.

This says that any external behavior of the code *T* must also be an external behavior of the spec *S*. Another way of looking at this is that we shouldn't be able to tell by looking at the code that we aren't looking at the spec, so we have to be able to explain every behavior of *T* as a possible behavior of *S*.

The reverse, however, is not true. We do not insist that the code must exhibit every behavior allowed by the spec. In the case of the simple memory the spec is completely deterministic, so the code cannot take advantage of this freedom. In general, however, the spec may allow lots of behaviors and the code choose just one. The spec for sorting, for instance, allows any sorted sequence as the result of `Sort`; there may be many such sequences if the ordering relation is not total. The code will usually be deterministic and return exactly one of them, so it doesn't exhibit all the behavior allowed by the spec.

Safety and liveness

Just as with procedures, this subset requirement is not strong enough to satisfy our intuitive notion of code. In particular, it allows the set of traces generated by *T* to be empty; in other word, the code might do nothing at all, or it might do some things and then stop. As we saw, the analog of this for a simple sequential procedure is non-termination. Usually we want to say that the code of a procedure should terminate, and similarly we want to say that the code of a module should keep doing things. More generally, when we have concurrency we usually want the code to be *fair*, that is, to eventually service all its clients, and more generally to eventually make any transition that continues to be enabled.

It turns out that any external behavior (that is, any set of traces) can be described as the intersection of two sets of traces, one defined by a *safety* property and the other defined by a *liveness* property.[1] A safety property says that in the trace nothing bad ever happens, or more precisely, that no bad transition occurs in the trace. It is analogous to partial correctness for a stateless procedure; a state machine that never makes a bad transition can define any safety property. If a trace doesn't satisfy a safety property, you can always find this out by looking at a finite prefix of the trace, in particular, at a prefix that includes the first bad transition.

A liveness property says that in the trace something good *eventually* happens. It is analogous to termination for a stateless procedure. You can never tell that a trace doesn't have a liveness property by looking at a finite prefix, since the good thing might happen later. A liveness property cannot be defined by a state machine. It is usual to express liveness properties in terms of *fairness*, that is, in terms of a requirement that if some transition stays enabled continuously it eventually occurs (weak fairness), or that if some transition stays enabled intermittently it eventually occurs (strong fairness).

With a few exceptions, we don't deal with liveness in this course. There are two reasons for this. First, it is usually not what you want. Instead, you want a result within some time bound, which is a safety property, or you want a result with some probability, which is altogether outside the framework we have set up. Second, liveness proofs are usually hard.

Abstraction functions and simulation

The definition of 'implements' as inclusion of external behavior is a sound formalization of our intuition. It is difficult to work with directly, however, since it requires reasoning about infinite sets of infinite sequences of actions. We would like to have a way of proving that *T* implements *S* that allows us to deal with one of *T*'s actions at a time. Our method is based on *abstraction functions*.

An abstraction function maps each state of the code *T* to a state of the spec *S*. For example, each state of the `WBCache` module gets mapped to a state of the `Memory` module. The abstraction function explains how to interpret each state of the code as a state of the spec. For example, Figure 3 depicts part of the abstraction function from `WBCache` to `Memory`. Here is its definition in Spec, copied from handout 5.

```
FUNC AF() -> M = RET (\ a | c!a => c(a) [*] m(a) )
```

You might think that an abstraction function should map the other way, from states of the spec to states of the code, explaining how to represent each state of the spec. This doesn't work, however, because there is usually more than one way of representing each state of the spec. For example, in the `WBCache` code for `Memory`, if an address is in the cache, then the value stored for that address in memory does not matter. There are also choices about which addresses appear in the cache. Thus, many states of the code can represent the same state of the spec. In other words, the abstraction function is many-to-one.

An abstraction function *F* is required to satisfy the following two conditions.

1. If t is any initial state of T, then $F(t)$ is an initial state of S.
2. If t is a reachable state of T and (t, π, t') is a step of T, then there is a step of S from $F(t)$ to $F(t')$, having the same trace.

[1] B. Alpern and F. Schneider. Recognizing safety and liveness. *Distributed Computing* **2**, 3 (1987), pp 117-126.

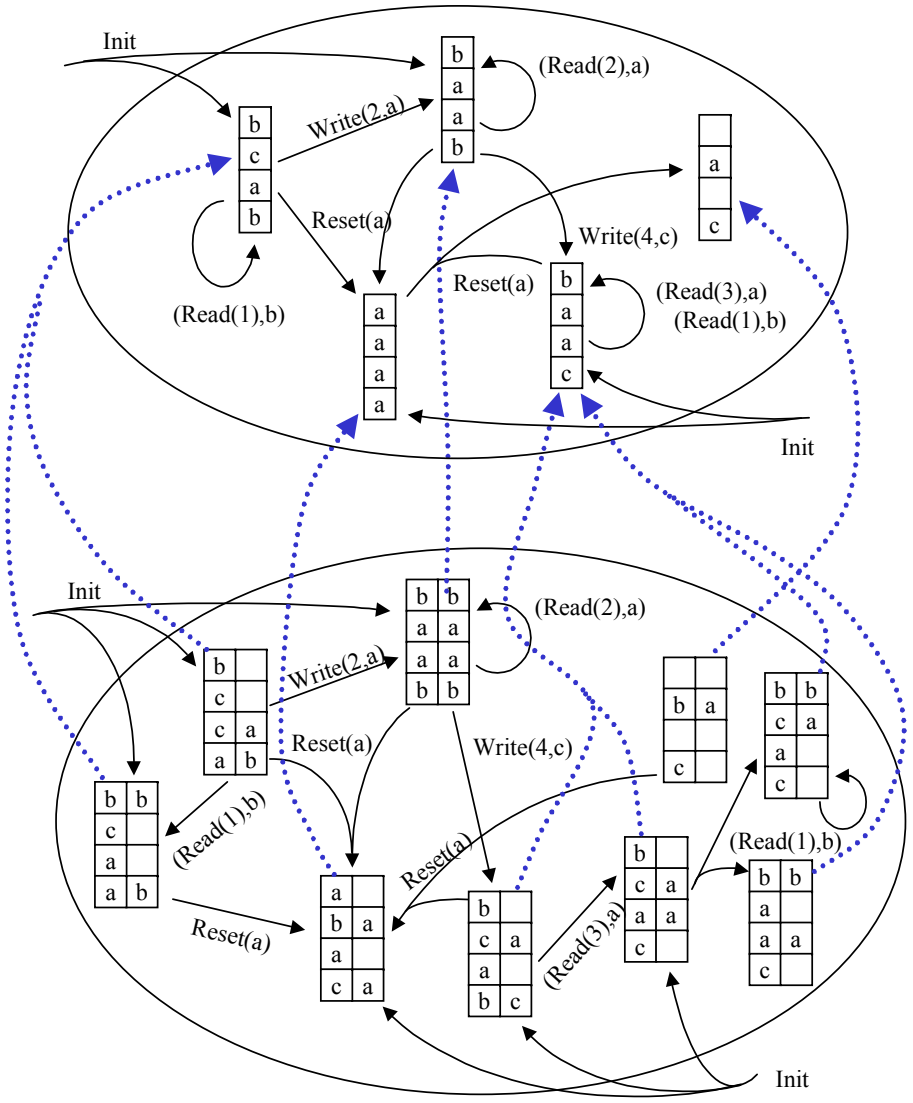


Figure 3: Abstraction function for `WBCache`

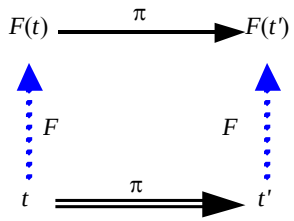


Figure 4: Commutative diagram for correctness

Condition 2 says that *T simulates S*; every step of *T* faithfully copies a step of *S*. It is stated in a particularly simple way, forcing the given step of *T* to simulate a single step of *S*. That is enough for the special case we are considering right now. Later, when we consider concurrent invocations for modules, we will generalize condition 2 to allow any number of steps of *S* rather than just a single step.

The diagram in Figure 4 represents condition 2. The dashed arrows represent the abstraction function *F*, and the solid arrows represent the transitions; if the lower (double) solid arrow exists in the code, the upper (single) solid arrow must exist in the spec. The diagram is sometimes called a "commutative diagram" because if you start at the lower left and follow arrows, you will end up in the same state regardless of which way you go.

An abstraction function is important because it can be used to show that one module implements another:

Theorem 1: If there is an abstraction function from *T* to *S*, then *T* implements *S*, i.e., every trace of *T* is a trace of *S*.

Note that this theorem applies to both finite and infinite traces.

Proof: (Sketch) Let β be any trace of *T*, and let α be any execution of *T* that generates trace β. Use Conditions 1 and 2 above to construct an execution α' of *S* with the same trace. That is, if *t* is the initial state of α, then let *F*(*t*) be the initial state of α'. For each step of α in turn, use Condition 2 to add a corresponding step to α'.

More formally, this proof is an induction on the length of the execution. Condition 1 gives the basis: any initial state of *T* maps to an initial state of *S*. Condition 2 gives the inductive step: if we have an execution of *T* of length *n* that simulates an execution of *S*, any next step by *T* simulates a next step by *S*, so any execution of *T* of length *n*+1 simulates an execution of *S*.

We would like to have an inverse of Theorem 1: if every trace of *T* is a trace of *S*, then there is an abstraction function that shows it. This is not true for the simple abstraction functions and simulations we have defined here. Later on, in handout 8, we will generalize them to a simulation method that is strong enough to prove that *T* implements *S* whenever that is true.

Invariants

An *invariant* of a module is any property that is true of all *reachable* states of the module, i.e., all states that can be reached in executions of the module (starting from initial states). Invariants are

important because condition 2 for an abstraction function requires us to show that the code simulates the spec from every reachable state, and the invariants characterize the reachable states. It usually isn't true that the code simulates the spec from every state.

Here are examples of invariants for the HashMemory and MajReg modules, written in Spec and copied from handout 5.

```
FUNC HashMemory.Inv(nb: Int, m: HashT, default: V) -> Bool = RET
     (  m.size = nb
     /\ (ALL i :IN m.dom, p :IN m(i).rng | p.a.hf = i)
     /\ (ALL a | { p :IN m(a.hf) | p.a = a }.size <= 1) )

FUNC MajReg.Inv(m) -> Bool = RET
        (ALL p :IN m.rng, p' :IN m.rng | p.n = p'.n ==> p.v = p'.v)
     /\ (EXISTS maj | (ALL i :IN maj, p :IN m.rng | m(i).n >= p.n)))
```

For example, for the HashMemory module, the invariant says (among other things) that a pair containing address a appears only in the appropriate bucket a.hf, and that at most one pair for an address appears in the bucket for that address.

The usual way to prove that a property *P* is an invariant is by induction on the length of finite executions leading to the states in question. That is, we must show the following:

(Basis, length = 0) *P* is true in every initial state.

(Inductive step) If (*t*, π, *t*') is a transition and *P* is true in *t*, then *P* is also true in *t*'.

Not all invariants are proved directly by induction, however. It is often better to prove invariants in groups, starting with the simplest invariants. Then the proofs of the invariants in the later groups can assume the invariants in the earlier groups.

Example: We sketch a proof that the property MajReg.Inv is in fact an invariant.

Basis: In any initial state, a single (arbitrarily chosen) default value *v* appears in all the copies, along with the sequence number 0. This immediately implies both parts of the invariant.

Inductive step: Suppose that (*t*, π, *t*') is a transition and Inv is true in *t*. We consider cases based on π. If π is an invocation or response, or the body of a Read procedure, then the step does not affect the truth of Inv. So it remains to consider the case where π is a Write, say Write(v).

In this case, the second part of the invariant for *t* (i.e., the fact that the highest n appears in more than half the copies), together with the fact that the Write reads a majority of the copies, imply that the Write obtains the highest n, say *i*. Then the new n that the Write chooses must be the new highest n. Since the Write writes *i*+1 to a majority of the copies, it ensures the second part of the invariant. Also, since it associates the same v with the sequence number *i*+1 everywhere it writes, it preserves the first part of the invariant.

Proofs using abstraction functions

Example: We sketch a proof that the function WBCache.AF given above is an abstraction function from WBCache to Memory. In this proof, we get by without any invariants.

For Condition 1, suppose that *t* is any initial state of WBCache. Then AF(*t*) is some (memory) state of Memory. But all memory states are allowable in initial states of Memory. Thus, AF(*t*) is an initial state of Memory, as needed. For Condition 2, suppose that *t* and AF(*t*) are states of WBCache and

Memory, respectively, and suppose that (*t*, π, *t*') is a step of WBCache. We consider cases, based on π.

For example, suppose π is Read(a). Then the step of WBCache may change the cache and memory by writing a value back to memory. However, these changes don't change the corresponding abstract memory. Therefore, the memory correspondence given by AF holds after the step. It remains to show that both Reads give the same result. This follows because:

The Read(a) in WBCache returns the value *t*.c(a) if it is defined, otherwise *t*.m(a).

The Read(a) in Memory returns the value of AF(*t*).m(a).

The value of AF(*t*).m(a) is equal to *t*.c(a) if it is defined, otherwise *t*.m(a). This is by the definition of AF.

For another example, suppose π is Write(a,v). Then the step of WBCache writes value v to location a in the cache. It may also write some other value back to memory. Since writing a value back does not change the corresponding abstract state, the only change to the abstract state is that the value in location a is changed to v. On the other hand, the effect of Write(a,v) in Memory is to change the value in location a to v. It follows that the memory correspondence, given by AF, holds after the step.

We leave the other cases, for the other types of operations, to the reader. It follows that AF is an abstraction function from WBCache to Memory. Then Theorem 1 implies that WBCache implements Memory, in the sense of trace set inclusion.

Example: Here is a similar analysis for MajReg, using MajReg.AF as the abstraction function.

```
FUNC AF() -> V = RET m.rng.max.v
```

This time we depend on the invariant MajReg.Inv. Suppose π is Read(a). No state changes occur in either module, so the only thing to show is that the return values are the same in both modules. In MajReg, the Read collects a majority of values and returns a value associated with the highest n from among that majority. By the invariant that says that the highest n appears in a majority of the copies, it must be that the Read in fact obtains the highest n that is present in the system. That is, the Read in MajReg returns a value associated with the highest n that appears in state *t*.

On the other hand, the Read in Register just returns the value of the single variable m in state *s*. Since AF(*t*) = *s*, it must be that *s*.m is a value associated with the highest n in *t*. But the uniqueness invariant says that there is only one such value, so this is the same as the value returned by the Read in MajReg.

Now suppose π is Write(v). Then the key thing to show is that AF(*t*') = *s*'. The majority invariant implies that the Write in MajReg sees the highest n *i* and thus *i*+1 is the new highest n. It writes (*i*+1, v) to a majority of the copies. On the other hand, the Write in Register just sets m to v. But clearly v is a value associated with the largest n after the step, so AF(*t*') = *s*' as required.

It follows that AF is an abstraction function from MajReg to Register. Then Theorem 1 implies that MajReg implements Register.

7. Disks and File Systems

Motivation

The two lectures on disks and file systems are intended to show you a number of things:

Some semi-realistic examples of specs.

Many important coding techniques for file systems.

Some of the tradeoffs between a simple spec and efficient code.

Examples of abstraction functions and invariants.

Encoding: a general technique for representing arbitrary types as byte sequences.

How to model crashes.

Transactions: a general technique for making big actions atomic.

There are a lot of ideas here. After you have read this handout and listened to the lectures, it's a good idea to go back and reread the handout with this list of themes in mind.

Outline of topics

We give the specs of disks and files in the `Disk` and `File` modules, and we discuss a variety of coding issues:

Crashes

Disks

Files

Caching and buffering of disks and files

Representing files by trees and extents

Allocation

Encoding and decoding

Directories

Transactions

Redundancy

Crashes

The specs and code here are without concurrency. However, they do allow for crashes. A crash can happen between any two atomic commands. Thus the possibility of crashes introduces a limited kind of concurrency.

When a crash happens, the volatile global state is reset, but the stable state is normally unaffected. We express precisely what happens to the global state as well as how the module recovers by including a `Crash` procedure in the module. When a crash happens:

1. The `Crash` procedure is invoked. It need not be atomic.
2. If the `Crash` procedure does a `CRASH` command, the execution of the current invocations (if any) stop, and their local state is discarded; the same thing happens to any invocations outside the module from within it. After `CRASH`, no procedure in the module can be invoked from outside until `Crash` returns.
3. The `Crash` procedure may do other actions, and eventually it returns.
4. Normal operation of the module resumes; that is, external invocations are now possible.

You can tell which parts of the state are volatile by looking at what `Crash` does; it will reset the volatile variables.

Because crashes are possible between any two atomic commands, atomicity is important for any operation that involves a change to stable state.

The meaning of a Spec program with this limited kind of concurrency is that each atomic command corresponds to a transition. A hidden piece of state called the program counter keeps track of what transitions are enabled next: they are the atomic commands right after the program counter. There may be several if the command after the program counter has `[]` as its operator. In addition, a crash transition is always possible; it resets the program counter to a null value from which no transition is possible until some external routine is invoked and then invokes the `Crash` routine.

If there are non-atomic procedures in the spec with many atomic commands, it can be rather difficult to see the consequences of a crash. It is therefore clearer to write a spec with as much atomicity as possible, making it explicit exactly what unusual transitions are possible when there's a crash. We don't always follow this style, but we give some examples of it, notably at the end of the section on disks.

Disks

Essential properties of a disk:

Storage is stable across crashes (we discuss error models for disks in the `Disk` spec).
It's organized in blocks, and the only atomic update is to write one block.
Random access is about 100k times slower than random access to RAM (10 ms vs. 100 ns)
Sequential access is 10-100 times slower than to RAM (40 MB/s vs. 400-6000 MB/s)
Costs 50 times less than RAM ($0.75/GB vs. $100/GB) in February 2004.
MTBF 1 million hours = 100 years.

Performance numbers:

Blocks of .5k - 4k bytes
75 MB/sec sequential, sustained (more with parallel disks)
3 ms average rotational delay (10000 rpm = 6 ms rotation time)
7 ms average seek time; 3 ms minimum

It takes 10 ms to get anything at all from a random place on the disk. In another 10 ms you can transfer 750 KB. Hence the cost to get 750 KB is only twice the cost to get 1 byte. By reading from several disks in parallel (called *striping* or *RAID*) you can easily increase the transfer rate by a factor of 5-10.

Performance techniques:

Avoid disk operations: use caching
Do sequential operations: allocate contiguously, prefetch, write to log
Write in background (write-behind)

A spec for disks

The following module describes a disk `Dsk` as a function from a `DA` to a disk block `DB`, which is just a sequence of `DBSize` bytes. The `Dsk` function can also yield `nil`, which represents a permanent read error. The module is a class, so you can instantiate as many `Disk`s as needed. The state is one `Dsk` for each `Disk`. There is a `New` method for making a new disk; think of this as ordering a new disk drive and plugging it in. An extent `E` represents a set of consecutive disk addresses. The main routines are the `read` and `write` methods of `Disk`: `read`, which reads an extent, and `write`, which writes `n` disk blocks worth of data sequentially to the extent `E{da, n}`. The write is not atomic, but can be interrupted by a failure after each single block is written.

Usually a spec like this is written with a concurrent thread that introduces permanent errors in the recorded data. Since we haven't discussed concurrency yet, in this spec we introduce the errors in `read`, using the `AddErrors` procedure. An error sets a block to `nil`, after which any read that includes that block raises the exception `error`. Strictly speaking this is illegal, since `read` is a function and therefore can't call the procedure `AddErrors`. When we learn about concurrency we can move `AddErrors` to a separate thread; in the meantime we take the liberty, since it would be a real nuisance for `read` to be a procedure rather than a function.

Since neither Spec nor our underlying model deals with probabilities, we don't have any way to say how likely an error is. We duck this problem by making `AddErrors` completely nondeterministic; it can do anything from introducing no errors (which we must hope is the usual case) to clobbering the entire disk. Characterizing errors would be quite tricky, since disks usually have at least two classes of error: failures of single blocks and failures of an entire disk. However, any user of this module must assume something about the probability and distribution of errors.

Transient errors are less interesting because they can be masked by retries. We don't model them, and we also don't model errors reported by `write`. Finally, a realistic error model would include the possibility that a block that reports a read error might later be readable after all.

```
CLASS Disk EXPORT Byte, Data, DA, E, DBSize, read, write, size, check, Crash =

TYPE Byte       =  IN 0 .. 255
     Data       =  SEQ Byte
     DA         =  Nat                                  % Disk block Address
     DB         =  SEQ Byte SUCHTHAT db.size = DBSize   % Disk Block
     Blocks     =  SEQ DB
     E          =  [da, size: Nat]                      % Extent, in disk blocks
                   WITH {das:=EToDAs, "IN":=(\ e, da | da IN e.das)}
     Dsk        =  DA -> (DB + Null)                    % a DB or nil (error) for each DA

CONST DBSize    := 1024                                 % bytes in a disk block

VAR  disk       :  Dsk

APROC new(size: Int) -> Disk = <<                       % overrides StdNew
     VAR dsk | dsk.dom = size.seq.rng =>                % size blocks, arbitrary contents
          self := StdNew(); disk := dsk; RET self >>

FUNC read(e) -> Data RAISES {notThere, error} =
     check(e); AddErrors();
     VAR dbs := e.das * disk |                          % contents of the blocks in e
          IF nil IN dbs => RAISE error [*] RET BToD(dbs) FI

PROC write(da, data) RAISES {notThere} =                % fails if data not n * DBsize
     VAR blocks := DToB(data), n := 0 |
     % Atomic by block, and in order
          check(E{da, blocks.size});
          DO blocks!n => WriteBlock(da + n, blocks(n)); n + := 1 OD

APROC WriteBlock(da, db) = << disk(da) := db >>         % the atomic update. PRE: disk!da

FUNC size() -> Int = RET disk.dom.size

APROC check(e) RAISES {notThere} =                      % every DA in e is in disk.dom
     << e.das.rng <= disk.dom => RET [*] RAISE notThere >>

PROC Crash() = CRASH                                    % no global volatile state

FUNC EToDAs(e) -> SEQ DA = RET e.da .. e.da+e.size-1    % e.das

% Internal routines

% Functions to convert between Data and Blocks.
FUNC BToD(blocks) -> Data   = RET + : blocks
FUNC DToB(data  ) -> Blocks = VAR blocks | BToD(blocks) = data => RET blocks
% Undefined if data.size is not a multiple of DBsize

APROC AddErrors() =                                     % clobber some blocks
     << DO RET [] VAR da :IN disk.dom | disk(da) := nil OD >>

END Disk
```

This module doesn't worry about the possibility that a disk may fail in such a way that the client can't tell whether a write is still in progress; this is a significant problem in fault tolerant systems that want to allow a backup processor to start running a disk as soon as possible after the primary fails.

Many disks do not guarantee the order in which blocks are written (why?) and thus do not implement this spec, but instead one with a weaker `write`:

```
PROC writeUnordered(da, data) RAISES {notThere} =
  VAR blocks := DToB(data) |
     % Atomic by block, in arbitrary order; assumes no concurrent writing.
     check(E{da, blocks.size});
     DO VAR n | blocks(n) # disk(da + n) => WriteBlock(da + n, blocks(n)) OD
```

In both specs `write` establishes `blocks = E{da, blocks.size}.das * disk`, which is the same as `data = read(E{da, blocks.size})`, and both change each disk block atomically. `writeUnordered` says nothing about the order of changes to the disk, so after a crash any subset of the blocks being written might be changed; `write` guarantees that the blocks changed are a prefix of all the blocks being written. (`writeUnordered` would have other differences from `write` if concurrent access to the disk were possible, but we have ruled that out for the moment.)

Clarifying crashes

In this spec, what happens when there's a crash is expressed by the fact that `write` is not atomic and changes the disk one block at a time in the atomic `WriteBlock`. We can make this more explicit by making the occurrence of a crash visible inside the spec in the value of the `crashed` variable. To do this, we modify `Crash` so that it temporarily makes `crashed` true, to give `write` a chance to see it. Then `write` can be atomic; it writes all the blocks unless `crashed` is true, in which case it writes some prefix; this will happen only if `write` is invoked between the `crashed := true` and the `CRASH` commands of `Crash`. To describe the changes to the disk neatly, we introduce an internal function `NewDisk` that maps a `dsk` value into another one in which disk blocks at `da` are replaced by corresponding blocks defined in `bs`.

Again, this wouldn't be right if there were concurrent accesses to `Disk`, since we have made all the changes atomically, but it gives the possible behavior if the only concurrency is in crashes.

```
VAR  crashed     : Bool := false

...

APROC write(da, data) RAISES {notThere} = <<                % fails if data not n * DBsize
     VAR blocks := DToB(data) |
          check(E{da, blocks.size});
          IF   crashed =>                                   % if crashed, write some prefix
               VAR n | n < blocks.size => blocks := blocks.sub(0, n)
          []   SKIP FI;
          disk := NewDisk(disk, da, blocks)
     >>

FUNC NewDisk(dsk, da, bs: (Int -> DB)) -> Dsk =             % dsk overwritten with bs at da
     RET dsk + (\ da' | da' - da) * bs

PROC Crash() = crashed := true; CRASH; crashed := false
```

For unordered writes we need only a slight change, to write an arbitrary subset of the blocks if there's a crash, rather than a prefix:

```
          IF   crashed =>                                   % if crashed, write some subset
               VAR w: SET N | w <= blocks.dom => blocks := blocks.restrict(w)
```

Specifying files

This section gives a variety of specs for files. Code follows in later sections.

We treat a file as just a sequence of bytes, ignoring permissions, modified dates and other paraphernalia. Files have names, and for now we confine ourselves to a single directory that maps names to files. We call the name a 'path name' `PN` with an eye toward later introducing multiple directories, but for now we just treat the path name as a string without any structure, so it might as well be an inode number. We package the operations on files as methods of `PN`. The main methods are `read` and `write`; we define the latter initially as `WriteAtomic`, and later introduce less atomic variations `Write` and `WriteUnordered`. There are also boring operations that deal with the size and with file names.

```
MODULE File EXPORT PN, Byte, Data, X, F, Crash =

TYPE PN          =  String                                  % Path Name
                    WITH {read:=Read, write:=WriteAtomic, size:=GetSize,
```



```
                    setSize:=SetSize, create:=Create, remove:=Remove,
                    rename:=Rename}
     N           =  Nat
     Byte        =  IN 0 .. 255
     Data        =  SEQ Byte
     X           =  Nat                                     % byte-in-file indeX
     F           =  Data                                    % File

     D           =  PN -> F                                 % Directory

VAR  d           := D{}                                     % undefined everywhere
```

Note that the only state of the spec is `d`, since files are only reachable through `d`.

There are tiresome complications in `Write` caused by the fact that the arguments may extend beyond the end of the file. These can be handled by imposing preconditions (that is, writing the spec to do `HAVOC` when the precondition isn't satisfied), by raising exceptions, or by defining some sensible behavior. This spec takes the third approach; `NewFile` computes the desired contents of the file after the write. So that it will work for unordered writes as well, it handles sparse `data` by choosing an arbitrary `data'` that agrees with `data` where `data` is defined. Compare it with `Disk.NewDisk`.

```
FUNC Read(pn, x, n) -> Data = RET d(pn).seg(x, n)
% Returns as much data as available, up to n bytes, starting at x.

APROC WriteAtomic(pn, x, data) = << d(pn) := NewFile(d(pn), x, data) >>

FUNC NewFile(f0, x, data: N -> Byte) -> F =
% f is the desired final file. Fill in space between f0 and x with zeros, and undefined data elements arbitrarily.
  VAR z := data.dom.max, z0 := f0.size , f, data' |
       data'.size = z /\ data'.restrict(data.dom) = data
    /\ f.size = {z0, x+z}.max
    /\ (ALL n |     ( n IN 0   .. {x, z0}.min-1 ==> f(n) = f0(n)      )
               /\   ( n IN z0  .. x-1           ==> f(n) = 0          )
               /\   ( n IN x   .. x+z-1         ==> f(n) = data'(n-x) )
               /\   ( n IN x+z .. z0-1          ==> f(n) = f0(n) )    )
    => RET f

FUNC GetSize(pn) -> X = RET d(pn).size

APROC SetSize(pn, x) = << VAR z := pn.size |
     IF   x <= z => << d(pn) := pn.read(0, z) >>            % truncate
     [*]  pn.write(z, 0.fill(x - z + 1))                    % handles crashes like write
     FI >>

APROC Create(pn) = << d(pn) := F{} >>
APROC Remove(pn) = << d := d{pn -> } >>
APROC Rename(pn1, pn2) = << d(pn2) := d(pn1); Remove(pn1) >>

PROC Crash() = SKIP                                         % no volatile state, changes all atomic

END File
```

`WriteAtomic` changes the entire file contents at once, so that a crash can never leave the file in an intermediate state. This would be quite expensive in most code. For instance, consider what is involved in making a write of 20 megabytes to an existing file atomic; certainly you can't overwrite the existing disk blocks one by one. For this reason, real file systems don't implement `WriteAtomic`. Instead, they change the file contents a little at a time, reflecting the fact that the underlying disk writes blocks one at a time. Later we will see how an atomic `Write` could be im-

plemented in spite of the fact that it takes several atomic disk writes. In the meantime, here is a more realistic spec for `Write` that writes the new bytes in order. It is just like `Disk.write` except for the added complication of extending the file when necessary, which is taken care of in `NewFile`.

```
APROC Write(pn, x, data) = <<
    IF   crashed =>                                          % if crashed, write some prefix
         VAR n | n < data.size => data := data.sub(0, n)
    [*] SKIP FI;
    d(pn) := NewFile(d(pn), x, data) >>

PROC Crash() = crashed := true; CRASH; crashed := false
```

This spec reflects the fact that only a single disk block can be written atomically, so there is no guarantee that all of the data makes it to the file before a crash. At the file level it isn't appropriate to deal in disk blocks, so the spec promises only bytewise atomicity. Actual code would probably make changes one page at a time, so it would not exhibit all the behavior allowed by the spec. There's nothing wrong with this, as long as the spec is restrictive enough to satisfy its clients.

`Write` does promise, however, that `f(n)` is changed no later than `f(n+1)`. Some file systems make no ordering guarantee; actually, any file system that runs on a disk without an ordering guarantee probably makes no ordering guarantee, since it requires considerable care, or considerable cost, or both to overcome the consequences of unordered disk writes. For such a file system the following `WriteUnordered` is appropriate; it is just like `Disk.writeUnordered`.

```
APROC WriteUnordered(pn, x, data) = <<
    IF   crashed =>                                          % if crashed, write some subset
         VAR w: SET N | w <= data.dom => data := data.restrict(w)
    [*] SKIP FI;
    d(pn) := NewFile(d(pn), x, data) >>
```

Notice that although writing a file is not atomic, `File`'s directory operations are atomic. This corresponds to the semantics that file systems usually attempt to provide: if there is a failure during a `Create`, `Remove`, or `Rename`, the operation is either completed or not done at all, but if there is a failure during a `Write`, any amount of the data may be written. The other reason for making this choice in the spec is simple: with the abstractions available there's no way to express any sensible intermediate state of a directory operation other than `Rename` (of course sloppy code might leave the directory scrambled, but that has to count as a bug; think what it would look like in the spec).

The spec we gave for `SetSize` made it as atomic as `write`. The following spec for `SetSize` is unconditionally atomic; this might be appropriate because an atomic `SetSize` is easier to implement than a general atomic `Write`:

```
APROC SetSize(pn, x) = << d(pn) := (d(pn) + {i :IN x.seq | 0}).seg(0, x) >>
```

Here is another version of `NewFile`, written in a more operational style just for comparison. It is a bit shorter, but less explicit about the relation between the initial and final states.

```
FUNC NewFile(f0, x, data) -> F = VAR z0 := f0.size, data' |
    data'.size = data.dom.max =>
        data' := data' + data;
        RET    (x > z0 => f0 + {n: IN z0 .. x-1 | 0} [*] f0.sub(0, x-1))
             + data'
             + f0.sub(f.size, z0-1)
```


Our `File` spec is missing some things that are important in real file systems:

Access control: permissions or access control lists on files, ways of defaulting these when a file is created and of changing them, an identity for the requester that can be checked against the permissions, and a way to establish group identities.

Multiple directories. We will discuss this when we talk about naming.

Quotas, and what to do when the disk fills up.

Multiple volumes or file systems.

Backup. We will discuss this later when we describe the copying file system.

Cached and buffered disks

The simplest way to decouple the file system client from the slow disk is to provide code for the `Disk` abstraction that does caching and write buffering; then the file system code need not change. The idea is the same as for cached memory, although for the disk we preserve the order of writes. We didn't do this for the memory because we didn't worry about failures.

Failures add complications; in particular, *the spec must change*, since buffering writes means that some writes may be lost if there is a crash. Furthermore, the client needs a way to ensure that its writes are actually stable. We therefore need a new spec `BDisk`. To get it, we add to `Disk` a variable `oldDisks` that remembers the previous states that the disk might revert to after a crash (note that this is not necessarily all the previous states) and code to use `oldDisks` appropriately. `BDisk.write` no longer needs to test `crashed`, since it's now possible to lose writes even if the crash happens after the `write`.

```
CLASS BDisk EXPORT ..., sync  =                              % write-buffered disk

TYPE ...
CONST ...
VAR  disk        :  Dsk                                      % as in Disk
     oldDisks    :  SET Dsk := {}

...

APROC write(da, data) RAISES {notThere} = <<                 % fails if data not n * DBsize
  << VAR blocks := DToB(data) |
         check(E{da, blocks.size});
         disk := NewDisk(disk, da, blocks);
         oldDisks \/ := {n | n < blocks.size ||
                        NewDisk(disk, da, blocks.sub(0, n))};
         Forget()
     >>

FUNC NewDisk(dsk, da, bs: (Int -> DB)) -> Dsk =              % dsk overwritten with bs at da
     RET dsk + (\ da' | da' - da) * bs

PROC sync() = oldDisks := {disk}                             % make disk stable

PROC Forget() = VAR ds: SET Dsk | oldDisks := oldDisks - ds + {disk}
% Discards an arbitrary subset of the remembered disk states.

PROC Crash() = CRASH; << VAR d :IN oldDisks | disk := d; sync() >>

END BDisk
```

Forget is there so that we can write an abstraction function for code for that doesn't defer all its disk writes until they are forced by Sync. A write that actually changes the disk needs to change oldDisks, because oldDisks contains the old state of the disk block being overwritten, and there is nothing in the state of the code after the write from which to compute that old state. Later we will study a better way to handle this problem: history variables or multi-valued mappings. They complicate the code rather than the spec, which is preferable. Furthermore, they do not affect the performance of the code at all.

A weaker spec would revert to a state in which any subset of the writes has been done. For this, we change the assignment to oldDisks in write, along the lines we have seen before. We apply the changes to any old disk, not just to the current one, to allow changes to the disk from several write operations to be reordered.

```
oldDisks:= {d :IN oldDisks, w: SET N | w <= blocks.dom ||
            NewDisk(d, da, blocks.restrict(w))};
```

The module BufferedDisk below is code for BDisk. It copies newly written data into the cache and does the writes later, preserving the original order so that the state of the disk after a crash will always be the state at some time in the past. In the absence of crashes this implements Disk and is completely deterministic. We keep track of the order of writes with a queue variable, instead of keeping a dirty bit for each cache entry as we did for cached memory. If we didn't do the writes in order, there would be many more possible states after a crash, and it would be much more difficult for a client to use this module. Many real disks have this unpleasant property, and many real systems deal with it by ignoring it.

A striking feature of this code is that it uses the same abstraction that it implements, namely BDisk. The code for BDisk that it uses we call UDisk (U for 'underlying'). We think of it as a 'physical' disk, and of course it is quite different from BufferedDisk: it contains SCSI controllers, magnetic heads, etc. A module that implements the same interface that it uses is sometimes called a *filter* or a *stackable module*. A Unix filter like sed is a familiar example that uses and implements the byte stream interface. We will see many other examples of this later.

Invocations of UDisk are in bold type, so you can easily see how the module depends on the lower-level code for BDisk.

```
CLASS BufferedDisk                                  % implements BDisk
    EXPORT Byte, Data, DA, E, DBSize, read, write, size, check, sync, Crash =

TYPE % Data, DA, DB, Blocks, E as in Disk
    N           =  Int
    J           =  Int

    Queue       =  SEQ DA                           % data is in cache

CONST
    cacheSize   := 1000
    queueSize   := 50

VAR udisk       :  Disk
    cache       :  DA -> DB := {}
    queue       := Queue{}

% ABSTRACTION FUNCTION bdisk.disk = udisk.disk + cache
% ABSTRACTION FUNCTION bdisk.oldDisks =
    { q: Queue | q <= queue || udisk.disk + cache.restrict(q.rng) }
```



```
% INVARIANT queue.rng <= cache.dom                  % if queued then cached
% INVARIANT queue.size = queue.rng.size             % no duplicates in queue
% INVARIANT cache.dom.size <= cacheSize             % cache not too big
% INVARIANT queue.size <= queueSize                 % queue not too big

APROC new(size: Int) -> BDisk = <<                  % overrides StdNew
    self := StdNew(); udisk := udisk.new(size); RET self >>

PROC read(e) -> Data RAISES {notThere} =
% We could make provision for read-ahead, but do not.
    check(e);
    VAR data := Data{}, da := e.da, upTo := e.da + e.size |
        DO da < upTo =>
            IF  cache!da => data + := cache(da); da + := 1
            [*] % read as many blocks from disk as possible
                VAR n := RunNotInCache(da, upTo),
                    buffer := udisk.read(E{da, n}),
                    k := MakeCacheSpace(n) |
                    % k blocks will fit in cache; add them.
                    DO VAR j :IN k.seq | ~ cache!(da + j) =>
                          cache(da + j) := udisk.DToB(buffer)(j)
                    OD;
                    data + := buffer; da + := n
            FI
        OD; RET data

PROC write(da, data) RAISES {notThere} =
  VAR blocks := udisk.DToB(data) |
    check(E{da, blocks.size});
    DO VAR n :IN queue.dom | queue(n) IN da .. da+size-1 => FlushQueue(n) OD;
    % Do any previously buffered writes to these addresses. Why?
    VAR j := MakeCacheSpace(blocks.size), n := 0 |
        IF  j < blocks.size => udisk.write(da, data)
            % Don't cache if the write is bigger than the cache.
        [*] DO blocks!n =>
                cache(da+n) := blocks(n); queue + := {da+n}; n + := 1
            OD
        FI

PROC Sync() = FlushQueue(queue.size - 1)

PROC Crash() = CRASH; cache := {}; queue := {}

FUNC RunNotInCache(da, upTo: DA) -> N =
    RET {n | da + n <= upTo /\ (ALL j :IN n.seq | ~ cache!(da + j)}.max

PROC MakeCacheSpace(n) -> Int =
% Make room for n new blocks in the cache; returning min(n, the number of blocks now available).
% May flush queue entries.
% POST: cache.dom.size + result <= cacheSize
    . . .

PROC FlushQueue(n) = VAR q := queue.sub(0, n) |
% Write queue entries 0 .. n and remove them from queue.
% Should try to combine writes into the biggest possible writes
    DO q # {} => udisk.write(q.head, 1); q := q.tail OD;
    queue := queue.sub(n + 1, queue.size - 1)

END BufferedDisk
```

This code keeps the cache as full as possible with the most recent data, except for gigantic writes. It would be easy to change it to make non-deterministic choices about which blocks to keep in the cache, or to take advice from the client about which blocks to keep. The latter would require changing the interface to accept the advice, of course.

Note that the only state of `BDisk` that this module can actually revert to after a crash is the one in which none of the queued writes has been done. You might wonder, therefore, why the body of the abstraction function for `BDisk.oldDisks` has to involve `queue`. Why can't it just be `{udisk.disk}`? The reason is that when the internal procedure `FlushQueue` does a write, it changes the state that a crash reverts to, and there's no provision in the `BDisk` spec for adding anything to `oldDisks` except during `write`. So `oldDisks` has to include all the states that the disk can reach after a sequence of 'internal' writes, that is, writes done in `FlushQueue`. And this is just what the abstraction function says.

Building other kinds of disks

There are other interesting and practical ways to code a disk abstraction on top of a 'base' disk. Some examples that are used in practice:

Mirroring: use two base disks of the same size to code a single disk of that size, but with much greater availability and twice the read bandwidth, by doing each write to both base disks.

Striping: use *n* base disks to code a single disk *n* times as large and with *n* times the bandwidth, by reading and writing in parallel to all the base disks

RAID: use *n* base disks of the same size to code a single disk *n*-1 times as large and with *n*-1 times the bandwidth, but with much greater availability, by using the *n*th disk to store the exclusive-or of the others. Then if one disk fails, you can reconstruct its contents from the others.

Snapshots: use 'copy-on-write' to code an ordinary disk and some number of read-only 'snapshots' of its previous state.

Buffered files

We need to make changes to the `File` spec if we want the option to code it using buffered disks without doing too many `sync`s. One possibility is do a `bdisk.sync` at the end of each `write`. This spec is not what most systems implement, however, because it's too slow. Instead, they implement a version of `File` with the following additions. This version allows the data to revert to any previous state since the last `Sync`. The additions are very much like those we made to `Disk` to get `BDisk`. For simplicity, we don't change `oldDs` for operations other than `write` and `setSize` (well, except for truncation); real systems differ in how much they buffer the other operations.


```
MODULE File EXPORT ..., Sync =

TYPE ...
VAR d           := D{}
    oldDs     : SET D := {}
...

APROC Write(pn, x, byte) = << VAR f0 := d(pn) |
    d(pn) := NewFile(f0, x, data);
    oldDs \/ := {n | n < data.size ||
                  d{pn -> NewFile(f0, x, data.sub(0, n))))} >>

APROC Sync() = << oldDs := {d} >>

PROC Crash() = CRASH; << VAR d' :IN oldDs => d := d'; Sync() >>

END File
```

Henceforth we will use `File` to refer to the modified module. Since we are not giving code for `File`, we leave out `Forget` for simplicity.

Many file systems do their own caching and buffering. They usually loosen this spec so that a crash resets each file to some previous state, but does not necessarily reset the entire system to a previous state. (Actually, of course, real file systems usually don't have a spec, and it is often very difficult to find out what they can actually do after a crash.)

```
MODULE File2 EXPORT ..., Sync =

TYPE ...
    OldFiles    =  PN -> SET F

VAR d           := D{}
    oldFiles    := OldFiles{* -> {}}

...

APROC Write(pn, x, byte) = << VAR f0 := d(pn) |
    d(pn) := NewFile(f0, x, data);
    oldFiles(pn) \/ := {n | n < data.size || NewFile(f0, x, data.sub(0, n))))} >>

APROC Sync() = << oldFiles:= OldFiles{* -> {}} >>

PROC Crash() =
    CRASH;
    << VAR d' |    d'.dom = d.dom
                /\ (ALL pn :IN d.dom | d'(pn) IN oldFiles(pn) \/ {d(pn)})
       => d := d' >>

END File
```

A still weaker spec allows `d` to revert to a state in which any subset of the byte writes has been done, except that the files still have to be sequences. By analogy with unordered `BDisk`, we change the assignment to `oldFiles` in `Write`.

```
    oldFiles(pn) := {f :IN oldFiles(pn), w: SET n | w <= data.dom ||
                        NewFile(f, x, data.restrict(w))} >>
```

Coding files

The main issue is how to represent the bytes of the file on the disk so that large reads and writes will be fast, and so that the file will still be there after a crash. The former requires using contiguous disk blocks to represent the file as much as possible. The latter requires a representation for `D` that can be changed atomically. In other words, the file system state has type `PN -> SEQ Byte`, and we have to find a disk representation for the `SEQ Byte` that is efficient, and one for the function that is robust. This section addresses the first problem.

The simplest approach is to represent a file by a sequence of disk blocks, and to keep an *index* that is a sequence of the `DA`'s of these blocks. Just doing this naively, we have

```
TYPE F            =  [das: SEQ DA, size: N]          % Contents and size in bytes
```

The abstraction function to the spec says that the file is the first `f.size` bytes in the disk blocks pointed to by `c`. Writing this as though both `File` and its code `FImpl0` had the file `f` as the state, we get

```
File.f = (+ : (FImpl0.f.das * disk.disk)).seg(0, FImpl0.f.size)
```

or, using the `disk.read` method rather than the state of `disk` directly

```
File.f = (+ : {da :IN FImpl0.f.das || disk.read(E{da, 1})}).seg(0, FImpl0.f.size)
```

But actually the state of `File` is `d`, so we should have the same state for `FImpl` (with the different representation for `F`, of course), and

```
File.d = (LAMBDA (pn) -> File.F =
     VAR f := FImpl0.d(pn) |                          % fails if d is undefined at pn
          RET (+ : (f.das * disk.disk)).seg(0, f.size)
```

We need an invariant that says the blocks of each file have enough space for the data.

```
% INVARIANT ( ALL f :IN d.rng | f.das.size * DBSize >= f.size )
```

Then it's easy to see how to code `read`:

```
PROC read(pn, x, n) =
     VAR f := dir(pn),
         diskData := + : (da :IN f.das || disk.read(E{da, 1})},
         fileData := diskData.seg(0, f.size) |
          RET fileData.seg(x, n)
```

To code `write` we need a way to allocate free `DA`s; we defer this to the next section.

There are two problems with using this representation directly:

1. The index takes up quite a lot of space (with 4 byte `DA`'s and `DBSize` = 1Kbyte it takes .4% of the disk). Since RAM costs about 50 times as much as disk, keeping it all in RAM will add about 20% to the cost of the disk, which is a significant dollar cost. On the other hand, if the index is not in RAM it will take two disk accesses to read from a random file address, which is a significant performance cost.

2. The index is of variable length with no small upper bound, so representing the index on the disk is not trivial either.

To solve the first problem, store `Disk.E`'s in the index rather than `DA`'s. A single extent can represent lots of disk blocks, so the total size of the index can be much less. Following this idea, we would represent the file by a sequence of `Disk.E`'s, stored in a single disk block if it isn't too big or in a file otherwise. This recursion obviously terminates. It has the drawback that random access to the file might become slow if there are many extents, because it's necessary to search them linearly to find the extent that contains byte `x` of the file.

To solve the second problem, use some kind of tree structure to represent the index. In standard Unix file systems, for example, the index is a structure called an *inode* that contains:

> a sequence of 10 `DA`'s (enough for a 10 KB file, which is well above the median file size), followed by
>
> the `DA` of an *indirect* `DB` that holds `DBSize/4` = 250 or so `DA`'s (enough for a 250 KB file), followed by
>
> the `DA` of a second-level indirect block that holds the `DA`'s of 250 indirect blocks and hence points to 250^2 = 62500 `DA`'s (enough for a 62 MB file),
>
> and so forth. The third level can address a 16 GB file, which is enough for today's systems.

Thus the inode itself has room for 13 `DA`'s. These systems duck the first problem; their extents are always a single disk block.

We give code for that incorporates both extents and trees, representing a file by a generalized extent that is a tree of extents. The leaves of the tree are *basic* extents `Disk.E`, that is, references to contiguous sequences of disk blocks, which are the units of i/o for `disk.read` and `disk.write`. The purpose of such a general extent is simply to define a sequence of disk addresses, and the `E.das` method computes this sequence so that we can use it in invariants and abstraction functions. The tree structure is there so that the sequence can be stored and modified more efficiently.

An extent that contains a sequence of basic extents is called a *linear* extent. To do fast i/o operations, we need a linear extent which includes just the blocks to be read or written, grouped into the largest possible basic extents so that `disk.read` and `disk.write` can work efficiently. `Flatten` computes such a linear extent from a general extent; the spec for `Flatten` given below flattens the entire extent for the file and then extracts the smallest segment that contains all the blocks that need to be touched.

`Read` and `Write` just call `Flatten` to get the relevant linear extent and then call `disk.read` and `disk.write` on the basic extents; `Write` may extend the file first, and it may have to read the first and last blocks of the linear extent if the data being written does not fill them, since the disk can only write entire blocks. Extending or truncating a file is more complex, because it requires changing the extent, and also because it requires allocation. Allocation is described in the next section. Changing the extent requires changing the tree.

The tree itself must be represented in disk blocks; methods inspired by B-trees can be used to change it while keeping it balanced. Our code shows how to extract information from the tree, but not how it is represented in disk blocks or how it is changed. In standard Unix file systems, changing the tree is fairly simple because a basic extent is always a single disk block in the multi-level indirect block scheme described above.

We give the abstraction function to the simple code above. It just says that the DAs of a file are the ones you get from Flatten.

The code below makes heavy use of function composition to apply some function to each element of a sequence: s * f is {f(s(0)), ..., f(s(s.size-1))}. If f yields an integer or a sequence, the combination + : (s * f) adds up or concatenates all the f(s(i)).

```
MODULE FSImpl =                                          % implements File

TYPE N          =  Nat
     E          =  [c: (Disk.DA + SE), size: N]          % size = # of DA's in e
                   SUCHTHAT Size(e) = e.size
                   WITH {das:=EToDAs, le:=EToLE}
     BE         =  E SUCHTHAT be.c IS Disk.DA            % Basic Extent
     LE         =  E SUCHTHAT le.c IS SEQ BE             % Linear Extent: sequence of BEs
                   WITH {"+":=Cat}
     SE         =  SEQ E                                 % Sequence of Extents: may be tree

     X          =  File.X
     F          =  [e, size: X]                          % size = # of bytes

     PN         =  File.PN                               % Path Name

CONST DBSize    := 1024

VAR  d          :  File.PN -> F := {}
     disk

% ABSTRACTION FUNCTION File.d = (LAMBDA (pn) -> File.F = d!pn =>
% The file is the first f.size bytes in the disk blocks of the extent f.e
     VAR f := d(pn),
         data := + : {be :IN Flatten(f.e, 0, f.e.size).c || disk.read(be)} |
          RET data.seg(0, f.size) )

% ABSTRACTION FUNCTION FImpl0.d = (LAMBDA (pn) -> FImpl0.F =
     VAR f := d(pn) | RET {be :IN Flatten(f.e, 0, f.e.size).c || be.c}

FUNC Size(e) -> Int = RET ( e IS BE => e.size [*] + :(e.c * Size) )
% # of DA's reachable from e. Should be equal to e.size.

FUNC EToDAs(e) -> SEQ DA =                               % e.das
% The sequence of DA's defined by e. Just for specs.
     RET ( e IS BE => {n :IN e.size.seq || e.c + n} [*] + :(e.c * EToDAs) )

FUNC EToLE(e) -> LE =                                    % e.le
% The sequence of BE's defined by e.
     RET ( e IS BE => LE{SE{e}, e.size}                 [*] + :(e.c * EToLE ) )

FUNC Cat(le1, le2) -> LE =
% The "+" method of LE. Merge e1 and e2 if possible.
     IF   e1 = {} => RET le2
     []   e2 = {} => RET le1
     []   VAR e1 := le1.c.last, e2 := le2.c.head, se |
          IF   e1.c + e1.size = e2.c =>
               se := le1.c.reml + SE{E{e1.c, e1.size + e2.size}} + le2.c.tail
          [*]  se := le1.c + le2.c
          FI;
          RET LE{se, le1.size + le2.size}
     FI
```



```
FUNC Flatten(e, start: N, size: N) -> LE = VAR le0 := e.le, le1, le2, le3 |
% The result le is such that le.das = e.das.seg(start, size);
% This is fewer than size DA's if e gets used up.
% It's empty if start >= e.size.
% This is not practical code; see below.
        le0 = le1 + le2 + le3
     /\ le1.size = {start, e.size}.min
     /\ le2.size = {size, {e.size - start, 0}.max}.min
     => RET le2

...

END FSImpl
```

This version of Flatten is not very practical; in fact, it is more like a spec than code for. A practical one, given below, searches the tree of extents sequentially, taking the largest possible jumps, until it finds the extent that contains the startth DA. Then it collects extents until it has gotten size DA's. Note that because each e.size gives the total number of DA's in e, Flatten only needs time log(e.size) to find the first extent it wants, provided the tree is balanced. This is a standard trick for doing efficient operations on trees: summarize the important properties of each subtree in its root node. A further refinement (which we omit) is to store cumulative sizes in an SE so that we can find the point we want with a binary search rather than the linear search in the DO loop below; we did this in the editor buffer example of handout 3.

```
FUNC Flatten(e, start: N, size: N) -> LE =
     VAR z := {size, {e.size - start, 0}.max}.min |
          IF  z = 0   => RET E{c := SE{},         size := 0}
          [*] e IS BE => RET E{c := e.c + start, size := z}.le
          [*] VAR se := e.c AS SE, sbe : SEQ BE := {}, at := start, want := z |
               DO want > 0 =>                    % maintain at + want <= Size(se)
                    VAR e1 := se.head, e2 := Flatten(e1, at, want) |
                         sbe := sbe + e2.c; want := want - e2.size;
                         se := se.tail; at := {at - e1.size, 0}.max
               OD;
               RET E{c := sbe, size := z}
          FI
```

Allocation

We add something to the state to keep track of which disk blocks are free:

```
VAR free: DA -> Bool
```

We want to ensure that a free block is not also part of a file. In fact, to keep from losing blocks, a block should be free iff it isn't in a file or some other data structure such as an inode:

```
PROC IsReachable(da) -> Bool =
     RET ( EXISTS f :IN d.rng | da IN f.e.das \/ ...

% INVARIANT (ALL da | IsReachable(da) = ~ free(da) )
```

This can't be coded without some sort of log-like mechanism for atomicity if we want separate representations for free and f.e, that is, if we want any code for free other than the brute-force search implied by IsReachable itself. The reason is that the only atomic operation we have on the disk is to write a single block, and we can't hope to update the representations of both free and f.e with a single block write. But ~ IsReachable is not satisfactory code for free, even

though it does not require a separate data structure, because it's too expensive — it traces the entire extent structure to find out whether a block is free.

A weaker invariant allows blocks to be lost, but still ensures that the file data will be inviolate. This isn't as bad as it sounds, because blocks will only be lost if there is a crash between writing the allocation state and writing the extent. Also, it's possible to garbage-collect the lost blocks.

```
% INVARIANT (ALL da | IsReachable(da) ==> ~ free(da))
```

A weaker invariant than this would be a disaster, since it would allow blocks that are part of a file to be free and therefore to be allocated for another file.

The usual representation of `free` is a `SEQ Bool` (often called a *bit table*). It can be stored in a fixed-size file that is allocated by magic (so that the code for allocation doesn't depend on itself). To reduce the size of `free`, the physical disk blocks may be grouped into larger units (usually called 'clusters') that are allocated and deallocated together.

This is a fairly good scheme. The only problem with it is that the table size grows linearly with the size of the disk, even when there are only a few large files, and concomitantly many bits may have to be touched to allocate a single extent. This will certainly be true if the extent is large, and may be true anyway if lots of allocated blocks must be skipped to find a free one.

The alternative is a tree of free extents, usually coded as a B-tree with the extent size as the key, so that we can find an extent that exactly fits if there is one. Another possibility is to use the extent address as the key, since we also care about getting an extent close to some existing one. These goals are in conflict. Also, updating the B-tree atomically is complicated. There is no best answer.

Encoding and decoding

To store complicated values on the disk, such as the function that constitutes a directory, we need to encode them into a byte sequence, since `Disk.Data` is `SEQ Byte`. (We also need encoding to send values in messages, an important operation later in the course.) It's convenient to do this with a pair of functions for each type, called `Encode` and `Decode`, which turn a value of the type into a byte sequence and recover the value from the sequence. We package them up into an `EncDec` pair.

```
TYPE Q          =  SEQ Byte
     EncDec     =  [enc: Any -> Q, dec: Q -> Any]      % Encode/Decode pair
                   SUCHTHAT ( EXISTS T: SET Any |
                                 encDec.enc.dom = T
                              /\ (ALL t :IN T | dec(enc(t)) = t) ))
```

Other names for 'encode' are 'serialize' (used in Java), 'pickle', and 'marshal' (used for encoding arguments and results of remote procedure calls).

A particular `EncDec` works only on values of a single type (represented by the set `T` in the `SUCHTHAT`, since you can't quantify over types in Spec). This means that `enc` is defined exactly on values of that type, and `dec` is the inverse of `enc` so that the process of encoding and then decoding does not lose information. We do *not* assume that `enc` is the inverse of `dec`, since there may be many byte sequences that decode to the same value; for example, if the value is a set, it would be pointless and perhaps costly to insist on a canonical ordering of the encoding. In this

course we will generally assume that every type has methods `enc` and `dec` that form an `EncDec` pair.

A type that has other types as its components can have its `EncDec` defined in an obvious way in terms of the `EncDec`'s of the component types. For example, a `SEQ T` can be encoded as a sequence of encoded `T`'s, provided the decoding is unambiguous. A function `T -> U` can be encoded as a set or sequence of encoded `(T, U)` pairs.

A directory is one example of a situation in which we need to encode a sequence of values into a sequence of bytes. A log is another example of this, discussed below, and a stream of messages is a third. It's necessary to be able to parse the encoded byte sequence unambiguously and recover the original values. We can express this idea precisely by saying that a parse is an `EncDec` sequence, a language is a set of parses, and the language is unambiguous if for every byte sequence `q` the language has at most one parse that can completely decode `q`.

```
TYPE M          =  SEQ Q                                  % for segmenting a Q
     P          =  SEQ EncDec                             % Parse
     % A sequence of decoders that parses a Q, as defined by IsParse below
     Language   =  SET P

FUNC IsParse(p, q) -> Bool = RET ( EXISTS m |
        + :m = q                                           % m segments q
     /\ m.size = p.size                                    % m is the right size
     /\ (ALL n :IN p.dom | (p(n).dec)!m(n)] )              % each p decodes its m

FUNC IsUnambiguous(l: Language) -> Bool = RET (ALL q, p1, p2|
     p1 IN l /\ p2 IN l /\ IsParse(p1, q) /\ IsParse(p2, q) ==> p1 = p2)
```

Of course ambiguity is not decidable in general. The standard way to get an unambiguous language for encodings is to use type-length-value (TLV) encoding, in which the result `q` of `enc(x)` starts with some sort of encoding of `x`'s type, followed by an encoding of `q`'s own length, followed by a `Q` that contains the rest of the information the decoder needs to recover `x`.

```
FUNC IsTLV(ed: EncDec) -> Bool =
     RET (ALL x :IN ed.enc.dom | (EXISTS d1, d2, d3 |
             ed.enc(x) = d1 + d2 + d3 /\ EncodeType(x) = d1
          /\ (ed.enc(x).size).enc = d2 ))
```

In many applications there is a grammar that determines each type unambiguously from the preceding values, and in this case the types can be omitted. For instance, if the sequence is the encoding of a `SEQ T`, then it's known that all the types are `T`. If the length is determined from the type it can be omitted too, but this is done less often, since keeping the length means that the decoder can reliably skip over parts of the encoded sequence that it doesn't understand. If desired, the encodings of different types can make different choices about what to omit.

There is an international standard called ASN-1 (for Abstract Syntax Notation) that defines a way of writing a grammar for a language and deriving the `EncDec` pairs automatically from the grammar. Like most such standards, it is rather complicated and often yields somewhat inefficient encodings. It's not as popular as it used to be, but you might stumble across it.

Another standard way to get an unambiguous language is to encode into S-expressions, in which the encoding of each value is delimited by parentheses, and the type, unless it can be omitted, is given by the first symbol in the S-expression. A variation on this scheme which is popular for Internet Email and Web protocols, is to have a 'header' of the form

```
    attribute1: value1
    attribute2: value2
    ...
```

with various fairly ad-hoc rules for delimiting the values that are derived from early conventions for the human-readable headers of Email messages.

The trendy modern version serialization languae is called XML (eXtensible Markup Language). It generalizes S-expressions by having labeled parentheses, which you write `<foo>` and `</foo>`.

In both TLV and S-expression encodings, decoding depends on knowing exactly where the byte sequence starts. This is not a problem for `Q`'s coming from a file system, but it is a serious problem for `Q`'s coming from a wire or byte stream, since the wire produces a continuous stream of voltages, bits, bytes, or whatever. The process of delimiting a stream of symbols into `Q`'s that can be decoded is called *framing*; we will discuss it later in connection with networks.

Directories

Recall that a `D` is just a `PN -> F`. We have seen various ways to represent `F`. The simplest code relies on an `EncDec` for an entire `D`. It represents a `D` as a file containing `enc` of the `PN -> F` map as a set of ordered pairs.

There are two problems with this scheme:

- Lookup in a large `D` will be slow, since it requires decoding the whole `D`. This can be fixed by using a hash table or B-tree. Updating the `D` can still be done as in the simple scheme, but this will also be slow. Incremental update is possible, if more complex; it also has atomicity issues.
- If we can't do an atomic file write, then when updating a directory we are in danger of scrambling it if there is a crash during the write. There are various ways to solve this problem. The most general and practical way is to use the transactions explained in the next section.

It is very common to code directories with an extra level of indirection called an 'inode', so that we have

```
TYPE INo        =  Int                          % Inode Number
     D          =  PN -> INo
     INoMap     =  INo -> F

VAR  d          :  D := {}
     inodes     :  INoMap := {}
```

You can see that `inodes` is just like a directory except that the names are `INo`'s instead of `PN`'s. There are three advantages:

Because `INo`'s are integers, they are cheaper to store and manipulate. It's customary to provide an `Open` operation to turn a `PN` into an `INo` (usually through yet another level of indirection called a 'file descriptor'), and then use the `INo` as the argument of `Read` and `Write`.

Because `INo`'s are integers, if `F` is fixed-size (as in the Unix example discussed earlier, for instance) then `inodes` can be represented as an array on the disk that is just indexed by the `INo`.

The enforced level of indirection means that file names automatically get the semantics of pointers or memory addresses: two of them can point to the same file variable.

The third advantage can be extended by extending the definition of `D` so that the value of a `PN` can be another `PN`, usually called a "symbolic link".

```
TYPE D          =  PN -> (INo + PN)
```

Transactions

We have seen several examples of a general problem: to give a spec for what happens after a crash that is acceptable to the client, and code for that satisfies the spec even though it has only small atomic actions at its disposal. In writing to a file, in maintaining allocation information, and in updating a directory, we wanted to make a possibly large state change atomic in the face of crashes during its execution, even though we can only write a single disk block atomically.

The general technique for dealing with this problem is called *transactions*. General transactions make large state changes atomic in the face of arbitrary concurrency as well as crashes; we will discuss this later. For now we confine ourselves to 'sequential transactions', which only take care of crashes. The idea is to conceal the effects of a crash entirely within the transaction abstraction, so that its clients can program in a crash-free world.

The code for sequential transactions is based on the very general idea of a *deterministic state machine* that has inputs called *actions* and makes a deterministic transition for every input it sees. The essential observation is that:

> If two instances of a deterministic state machine start in the same state and see the same inputs, they will make the same transitions and end up in the same state.

This means that if we record the sequence of inputs, we can replay it after a crash and get to the same state that we reached before the crash. Of course this only works if we start in the same state, or if the state machine has an 'idempotency' property that allows us to repeat the inputs. More on this below.

Here is the spec for sequential transactions. There's a state that is queried and updated (read and written) by actions. We keep a stable version `ss` and a volatile version `vs`. Updates act on the volatile version, which is reset to the stable version after a crash. A 'commit' action atomically sets the stable state to the current volatile state.

```
MODULE SeqTr [                                  % Sequential Transaction
    V,                                          % Value of an action
    S WITH { s0: ()-> S }                       % State; s0 initially
    ] EXPORT Do, Commit, Crash =

TYPE A          =  S->(V, S)                    % Action

VAR  ss         := S.s0()                       % Stable State
     vs         := S.s0()                       % Volatile State

APROC Do(a) -> V = << VAR v | (v, vs) := a(vs); RET v >>
APROC Commit() = << ss := vs >>
APROC Crash () = << vs := ss >>                 % Abort is the same

END SeqTr
```

In other words, you can do a whole series of actions to the volatile state `vs`, followed by a `Commit`. Think of the actions as reads and writes, or queries and updates. If there's a crash before the `Commit`, the state reverts to what it was initially. If there's a crash after the `Commit`, the state

reverts to what it was at the time of the commit. An action is just a function from an initial state to a final state and a result value.

There are many coding techniques for transactions. Here is the simplest. It breaks each action down into a sequence of *updates*, each one of which can be done atomically; the most common example of an atomic update is a write of a single disk block. The updates also must have an 'idempotency' property discussed later. Given a sequence of `Do`'s, each applying an action, the code concatenates the update sequences for the actions in a volatile *log* that is a representation of the actions. `Commit` writes this log atomically to a stable log. Once the stable log is written, `Redo` applies the volatile log to the stable state and erases both logs. `Crash` resets the volatile to the stable log and then applies the log to the stable state to recover the volatile state. It then uses `Redo` to update the stable state and erase the logs. Note that we give `S` a `"+"` method `s + l` that applies a log to a state.

This scheme reduces the problem of implementing arbitrary changes atomically to the problem of atomically writing an arbitrary amount of stuff to a log. This is easier, but still not trivial to do efficiently; we discuss it at the end of the section.

We begin with code that is simple, but somewhat impractical. It uses lazy evaluation for `ss`, representing it as the result of applying a stable log (sequence of updates) `sl` to a fixed initial state. By contrast, there's an explicit `vs` variable as well as a volatile log `vl`, with an invariant relating them.

```
MODULE SimpleLogRecovery [                          % implements SeqTr
    V,                                              % Value of an action
    S0 WITH { s0: () -> S0 }                        % State
    ] EXPORT Do, Commit, Crash =

TYPE A          =  S->(V, S)                        % Action
    U           =  S -> S                           % atomic Update
    L           =  SEQ U                            % Log
    S           =  S0 WITH { "+":=DoLog }           % State; s+l applies l to s

VAR  vs         := S.s0()                           % Volatile State
    sl          := L{}                              % Stable Log
    vl          := L{}                              % Volatile Log

% ABSTRACTION to SeqTr
    SeqTr.ss = S.s0 + sl
    SeqTr.vs = vs

% INVARIANT vs = S.s0 + sl + vl

APROC Do(a) -> V = << VAR v, l | (v, vs + l) = a(vs) =>
% Find an l (a sequence of updates) that has the same effect as a on the current state. Compare SeqTr.Do
    vl := vl + l; vs := vs + l; RET v >>

PROC Commit() = << sl := vl >>

PROC Crash() =
    CRASH;
    << vl := {}; vs := S.s0() >>;                   % crash erases vs, vl
    << vs := S.s0 + sl >>;                          % recovery restores vs

% Internal procedures
```



```
FUNC DoLog(s, l) -> S =                             % s+l = DoLog(s, l)
% Apply the updates in l to the state s.
    l={} => RET s [*] RET DoLog((l.head)(s),l.tail))
```

We can write this more concisely as applying the composition of the updates in `l` to `s`:

```
FUNC DoLog(s, l) -> S = RET (* : l)(s)              % s+l = DoLog(s, l)

END SimpleLogRecovery
```

This is not so great for three reasons:

1. `sl` grows without bound
2. The time to recover `vs` likewise grows without bound.
3. The size of `vs` grows, so we will in general have to represent part of it on the disk, but we don't take any advantage of the fact that this part is stable.

To overcome these problems, we introduce more elaborate code that maintains an explicit `ss`, and uses `sl` only for the changes made since `ss` was updated.

```
MODULE LogRecovery [...] EXPORT Do,Commit,Crash =   % implements SeqTr
    % Parameters and types as in SimpleLogRecovery

VAR  ss         := S.s0()                           % Stable State
    vs          := S.s0()                           % Volatile State
    sl          := L{}                              % Stable Log
    vl          := L{}                              % Volatile Log

% ABSTRACTION to SeqTr
    SeqTr.ss = ss + sl
    SeqTr.vs = vs

% INVARIANT vs = ss + vl

APROC Do(a) -> V = << VAR v, l | (v, vs + l) = a(vs) =>
% Find an l (a sequence of updates) that has the same effect as a on the current state.
    vl := vl + l; vs := vs + l; RET v >>

PROC Commit() = << sl := vl >>; Redo()

PROC Crash() =
    CRASH;
    << vl := {}; vs := S.s0()  >>;                  % crash erases vs, vl
    << vl := sl; vs := ss + vl >>;                  % recovery restores them
    Redo()                                          % and repeats the Redo; optional

% Internal procedures

FUNC DoLog(s, l) -> S =                             % s+l = DoLog(s, l)
% Apply the updates in l to the state s.
    l={} => RET s [*] RET DoLog((l.head)(s),l.tail))

PROC Redo() =                                       % replay vl, then clear sl
    DO vl # {} => << ss := ss + vl.head; vl := vl.tail >> OD; << sl := {} >>

END LogRecovery
```

For this *redo* crash recovery to work, `l` must have the property that `ss + l' + l = ss + l` for any prefix `l' <= l`. This will certainly be true if `l' + l = l`, that is, `l` 'absorbs' any prefix of itself. and from that it follows that repeatedly applying prefixes of `l`, followed by all of `l`, has the same effect as applying `l`. For example, suppose `l = L{a;b;c;d;e}`. Then `L{a;b;c; a; a; a;b;c;d; a;b; a;b;c;d;e; a; a;b;c;d;e}` must have the same effect as `l` itself; here we have grouped the prefixes together for clarity. We need this property because a crash can happen while `Redo` is running; the crash reapplies the whole log and runs `Redo` again. Another crash can happen while the second `Redo` is running, and so forth.

This 'hiccup' property follows from 'log idempotence', `l + l = l`, because

```
l' + l = l' + l' + k = l' + k = l
```

where `l = l' + k`. That is, we can keep absorbing the last hiccup `l'` into the final complete `l`. For example, taking some liberties with the notation for sequences:

```
  abcaaabcdababcdeaabcde
= abcaaabcdababcde + (a + abcde)
= abcaaabcdababcde + abcde
= abcaaabcdab + (abcde + abcde)
= abcaaabcdab + abcde
= abcaaabcd + (ab + abcde)
= abcaaabcd + abcde
```

and so forth.

We can get log idempotence if the `U`'s commute and are idempotent (that is, `u * u = u`), or if they are all writes, since the last write to each variable wins. More generally, for arbitrary `U`'s we can attach a `UID` to each `U` and record it in `S` when the `U` is applied, so we can tell that it shouldn't be applied again. Calling the original state `SS`, and defining a `meaning` method that turns a `U` record into a function, we have

```
TYPE
    S           =  [ss, tags: SET UID]
    U           =  [uu: SS->SS, tag: UID] WITH { meaning:=Meaning }

FUNC Meaning(u, s)->S =
        u.tag IN s.tags => RET s                        % u already done
    [*] RET S{ (u.uu)(s.ss), s.tags + {u.tag} }
```

If all the `U`'s in `l` have different tags, we get log idempotence. The tags make `U`'s 'testable' in the jargon of transaction processing; after a crash we can test to find out whether a `U` has been done or not. In the standard database code each `U` works on one disk page, the tag is the 'log sequence number', the index of the update in the log, and the update writes the tag on the disk page.

Writing the log atomically

There is still an atomicity problem in this code: `Commit` atomically does `<< sl := vl >>`, and the logs can be large. A simple way to use a disk to code a log that requires this assignment of arbitrary-sized sequences is to keep the size of `sl` in a separate disk block, and to write all the data first, then do a `Sync` if necessary, and finally write the new size. Since `sl` is always empty before this assignment, in this representation it will remain empty until the single `Disk.write` that sets its size. This is rather wasteful code, since it does an extra disk write.

More efficient code writes a 'commit record' at the end of the log, and treats the log as empty unless the commit record is present. Now it's only necessary to ensure that the log can never be mis-parsed if a crash happens while it's being written. An easy way to accomplish this is to write a distinctive 'erased value into each disk block that may become part of the log, but this means

that for every disk write to a log block, there will be another write to erase it. To avoid this cost we can use a ring buffer of disk blocks for the log and a sequence number that increments each time the ring buffer wraps around; then a block is 'erased' if its sequence number is not the current one. There's still a cost to initialize the sequence numbers, but it's only paid once. With careful code, a single bit of sequence number is enough.

In some applications it's inconvenient to make room in the data stream for a sequence number every `DBsize` bytes. To get around this, use a 'displaced' representation for the log, in which the first data bit of each block is removed from its normal position to make room for the one bit sequence number. The displaced bits are written into their own disk blocks at convenient intervals.

Another approach is to compute a strong checksum for the log contents, write it at the end after all the other blocks are known to be on the disk, and treat the log as empty unless a correct checksum is present. With a good *n*-bit checksum, the probability of mis-parsing is 2^{-n}.

Redundancy

A disk has many blocks. We would like some assurance that the failure of a single block will not damage a large part of the file system. To get such assurance we must record some critical parts of the representation redundantly, so that they can be recovered even after a failure.

The simplest way to get this effect is to record *everything* redundantly. This gives us more: a single failure won't damage *any* part of the file system. Unfortunately, it is expensive. In current systems this is usually done at the disk abstraction, and is called *mirroring* or *shadowing* the disk.

The alternative is to record redundantly only the information whose loss can damage more than one file: extent, allocation, and directory information.

Another approach is to

> do all writes to a log,
>
> keep a copy of the log for a long time (by writing it to tape, usually), and
>
> checkpoint the state of the file system occasionally.

Then the current state can be recovered by restoring the checkpoint and replaying the log from the moment of the checkpoint. This method is usually used in large database systems, but not in any file systems that I know of.

We will discuss these methods in more detail near the end of the course.

Copying File Systems

The file system described in `FSImpl` above separates the process of adding `DB`'s to the representation of a file from the process of writing data into the file. A *copying* file system (CFS) combines these two processes into one. It is called a 'log-structured' file system in the literature[1], but as we shall see, the log is not the main idea. A CFS is based on three ideas:

- Use a generational copying garbage collector (called a *cleaner*) to reclaim `DB`'s that are no longer reachable and keep all the free space in a single (logically) contiguous region, so that there is no need for a bit table or free list to keep track of free space.
- Do *all* writes sequentially at one end of this region, so that existing data is never overwritten and new data is sequential.
- Log and cache updates to metadata (the index and directory) so that the metadata doesn't have to be rewritten too often.

A CFS is a very interesting example of the subtle interplay among the ideas of sequential writing, copying garbage collection, and logging. This section describes the essentials of a CFS in detail and discusses more briefly a number of refinements and practical considerations. It will repay careful study.

Here is a picture of a disk organized for a CFS:

```
abc==defgh====ijkl=m=nopqrs-----------------
```

In this picture letters denote reachable blocks, `=`'s denote unreachable blocks that are not part of the free space, and `-`'s denote free blocks (contiguous on the disk viewed as a ring buffer). After the cleaner copies blocks `a-e` the picture is

```
-------fgh====ijkl=m=nopqrsabcde------------
```

because the data `a-e` has been copied to free space and the blocks that used to hold `a-e` are free, together with the two unreachable blocks which were not copied. Then after blocks `g` and `j` are overwritten with new values `G` and `J`, the picture is

```
-------f=h====i=kl=m=nopqrsabcdeGJ----------
```

The new data `G` and `J` has been written into free space, and the blocks that used to hold `g` and `j` are now unreachable. After the cleaner runs to completion the picture is

```
--------------------nopqrsabcdeGJfhiklm----
```

Pros and cons

A CFS has two main advantages:

- All writing is done sequentially; as we know, sequential writes are much faster than random writes. We have a good technique for making disk reads faster: caching. As main memory caches get bigger, more reads hit in the cache and disks spend more of their time writing, so we need a technique to make writes faster.
- The cleaner can copy reachable blocks to anywhere, not just to the standard free space region, and can do so without interfering with normal operation of the system. In particular, it can copy reachable blocks to tape for backup, or to a different disk drive that is faster, cheaper, less full, or otherwise more suitable as a home for the data.

[1] M. Rosenblum and J. Osterhout, The design and implementation of a log-structured file system, *ACM Transactions on Computer Systems*, **10**, 1, Feb. 1992, pp 26-52.

There are some secondary advantages. Since the writes are sequential, they are not tied to disk blocks, so it's easy to write items of various different sizes without worrying about how they are packed into `DB`'s. Furthermore, it's easy to compress the sequential stream as it's being written[2], and if the disk is a RAID you never have to read any blocks to recompute the parity. Finally, there is no bit table or free list of disk blocks to maintain.

There is also one major drawback: unless large amounts of data in the same file are written sequentially, a file will tend to have lots of small extents, which can cause the problems discussed on page 13. In Unix file systems most files are written all at once, but this is certainly not true for databases. Ways of alleviating this drawback are the subject of current research. The cost of the cleaner is also a potential problem, but in practice the cost of the cleaner seems to be small compared to the time saved by sequential writes.

Updating metadata

For the CFS to work, it must update the index that points to the `DB`'s containing the file data on every write and every copy done by the cleaner, not just when the file is extended. And in order to keep the writing sequential, we must handle the new index information just like the file data, writing it into the free space instead of overwriting it. This means that the directory too must be updated, since it points to the index; we write it into free space as well. Only the *root* of the entire file system is written in a fixed location; this root says where to find the directory.

You might think that all this rewriting of the metadata is too expensive, since a single write to a file block, whether existing or new, now triggers three additional writes of metadata: for the index (if it doesn't fit in the directory), the directory, and the root. Previously none of these writes was needed for an existing block, and only the index write for a new block. However, the scheme for logging updates that we introduced to code transactions can also handle this problem. The idea is to write the *changes* to the index into a log, and cache the updated index (or just the updates) only in main memory. An example of a logged change is "block 43 of file 'alpha' now has disk address 385672". Later (with any luck, after several changes to the same piece of the index) we write the index itself and log the consequent changes to the directory; again, we cache the updated directory. Still later we write the directory and log the changes to the root. We only write a piece of metadata when:

> We run out of main memory space to cache changed metadata, or
>
> The log gets so big (because of many writes) that recovery takes too long.

To recover we replay the *active tail* of the log, starting before the oldest logged change whose metadata hasn't been rewritten. This means that we must be able to read the log sequentially from that point. It's natural to write the log to free space along with everything else. While we are at it, we can also log other changes like renames.

Note that a CFS can use exactly the same directory and index data as an ordinary file system, and in fact exactly the same code for `Read`. To do this we must give up the added flexibility we can get from sequential writing, and write each `DB` of data into a `DB` on the disk. Several codes have done this (but the simple code below does not).

[2] M. Burrows et al., On-line compression in a log-structured file system, *Proc. 5th Conference on Architectural Support for Programming Languages and Operating Systems*, Oct. 1992, pp 2-9. This does require some blocking so that the decompressor can obtain the initial state it needs.

The logged changes serve another purpose. Because a file can only be reached from a single directory entry (or inode), the cleaner need not trace the directory structure in order to find the reachable blocks. Instead, if the block at `da` was written as block `b` of file `f`, it's sufficient to look at the file index and find out whether block `b` of file `f` is still at `da`. But the triple `(b, f, da)` is exactly the logged change. To take advantage of this we must keep the logged change as long as `da` remains reachable since the cleaner needs it (it's called 'segment summary' information in the literature). We don't need to replay it on recovery once its metadata is written out, however, and hence we need the sequential structure of the log only for the active tail.

Existing CFS's use the extra level of naming called inodes that is described on page 14. Inode numbers don't change during writing or copying, so the `PN -> INo` directory doesn't change. The root points to index information for the inodes (called the 'inode map'), which points to inodes, which point to data blocks or, for large files, to indirect blocks that point to data blocks.

Segments

Running the cleaner is fairly expensive, since it has to read and write the disk. It's therefore important to get as much value out of it as possible, by cleaning lots of unreachable data instead of copying lots of data that is still reachable. To accomplish this, divide the disk into *segments*, large enough (say 1 MB or 10 MB) that the time to seek to a new segment is much smaller than the time to read or write a whole segment. Clean each segment separately. Keep track of the amount of unreachable space in each segment, and clean a segment when (unreachable space) * (age of data) exceeds a threshold. Rosenblum and Osterhout explain this rule, which is similar in spirit to what a generational garbage collector[3] does; the goal is to recover as much free space as possible, without allowing too much unreachable space to pile up in old segments.

Now the free space isn't physically contiguous, so we must somehow link the segments in the active tail together. We also need a table that keeps track for each segment of whether it is free, and if not, what its unreachable space and age are; this is cheap because segments are so large.

Backup

As we mentioned earlier, one of the major advantages of a CFS is that it is easier to back up. There are several reasons for this.

1. You can take a snapshot just by stopping the cleaner from freeing cleaned segments, and then copy the root information and the log to the backup medium, recording the logged data backward from the end of the log.
2. This backup data structure allows us to restore a single file (or a few files) in one pass.
3. It's only necessary to copy the log back to the point at which the previous backup started.
4. The disks reads done by backup are sequential and therefore fast. This is an important issue when the file system occupies many terabytes. At the 75 MB/s peak transfer rate of the disk, it takes $1.5*10^4$ seconds, or about 4 hours, to copy a terabyte. This means that a small number of disks and tapes running in parallel can do it in a fraction of a day. If the transfer rate is reduced to 1 MB/s by lots of seeks (which is what you get with random seeks if the average block size is 10 KB), the copying time becomes 10 days, which is impractical.

[3] H. Lieberman and C. Hewitt, A real-time garbage collector based on the lifetimes of objects, *Comm. ACM* **26**, 6, June 1983, pp 419-429.

5. If a large file is partially updated, only the updates will be logged and appear in the backup.
6. It's easy to merge several incremental backups to make a full backup.

To get these advantages, we have to retain the ordering of segments in the log even after recovery no longer needs it.

There have been several research implementations of CFS's, and at least one commercial one called Spiralog in Digital Equipment Corporation's (now HP's) VMS system. You can read a good deal about it at http://www.digital.com/info/DTJM00/.

A simple CFS implementation

We give code for `CopyingFS` of a CFS that contains all the essential ideas (except for segments, and the rule for choosing which segment to clean), but simplifies the data structures for the sake of clarity. `CopyingFS` treats the disk as a root `DB` plus a ring buffer of bytes. Since writing is sequential this is practical; the only cost is that we may have to pad to the end of a `DB` occasionally in order to do a `Sync`. A `DA` is therefore a byte address on the disk. We could dispense with the structure of disk blocks entirely in the representation of files, just write the `data` of each `File.Write` to the disk, and make a `FSImpl.BE` point directly to the resulting byte sequence on the disk. Instead, however, we will stick with tradition, take `BE = DA`, and represent a file as a `SEQ DA` plus its size.

So the disk consists of a root page, a busy region, and a free region (as we have seen, in a real system both busy and free regions would be divided into segments); see the figure below. The busy region is a sequence of encoded `Item`'s, where an `Item` is either a `D` or a `Change` to a `DB` in a file or to the `D`. The busy region starts at `busy` and ends just before `free`, which always points to the start of a disk block. We could write `free` into the root, but then making anything stable would require a (non-sequential) write of the root. Instead, the busy region ends with a recognizable `endDB`, put there by `Sync`, so that recovery can find the end of the busy region.

`dDA` is the address of the latest directory on the disk. The part of the busy region after `dDA` is the active tail of the log and contains the changes that need to be replayed during recovery to reconstruct the current directory; this arrangement ensures that we start the replay with a `d` to which it makes sense to apply the changes that follow.

This code does bytewise writes that are buffered in `buf` and flushed to the disk only by `Sync`. Hence after a crash the state reverts to the state at the last `Sync`. Without the replay done during recovery by `ApplyLog`, it would revert to the state the last time the root was written; be sure you understand why this is true.

We assume that a sequence of encoded `Item`'s followed by an `endDB` can be decoded unambiguously. See the earlier discussion of writing logs atomically.

Other simplifications:

1. We store the `SEQ DA` that points to the file `DB`'s right in the directory. In real life it would be a tree, along one of the lines discussed in `FSImpl`, so that it can be searched and updated efficiently even when it is large. Only the top levels of the tree would be in the directory.
2. We keep the entire directory in main memory and write it all out as a single `Item`. In real life we would cache parts of it in memory and write out only the parts that are dirty (in other words, that contain changes).

3. We write a data block as part of the log entry for the change to the block, and make the `DA`'s in the file representation point to these log entries. In real life the logged change information would be batched together (as 'segment summary information') and the data written separately, so that recovery and cleaning can read the changes efficiently without having to read the file data as well, and so that contiguous data blocks can be read with a single disk operation and no extra memory-to-memory copying.

4. We allocate space for data in `Write`, though we buffer the data in `buf` rather than writing it immediately. In real life we might cache newly written data in the hope that another adjacent write will come along so that we can allocate contiguous space for both writes, thus reducing the number of extents and making a later sequential read faster.

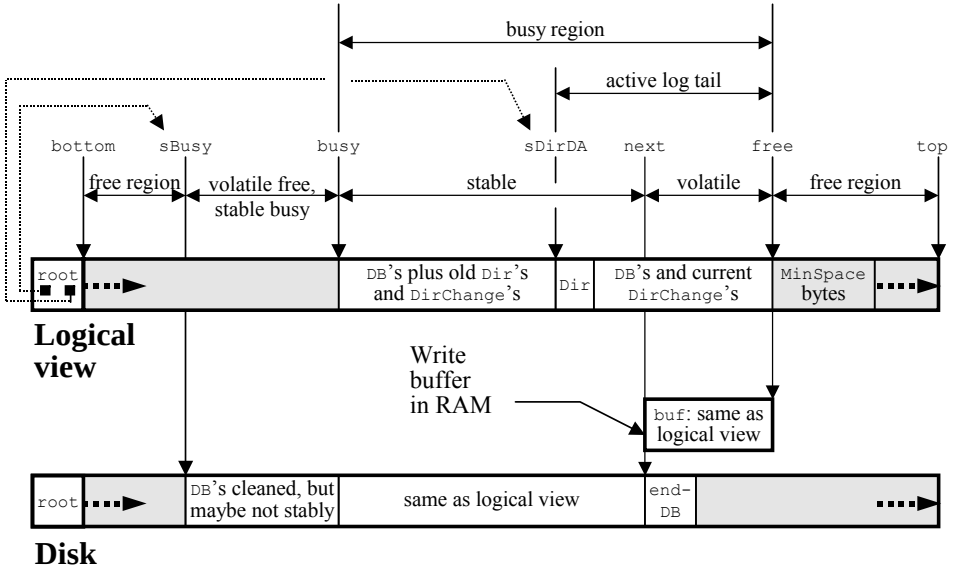
5. Because we don't have segments, the cleaner always copies items starting at `busy`. In real life it would figure out which segments are most profitable to clean.

6. We run the cleaner only when we need space. In real life, it would run in the background to take advantage of times when the disk is idle, and to maintain a healthy amount of free space so that writes don't have to wait for the cleaner to run.

7. We treat `WriteData` and `WriteRoot` as atomic. In real life we would use one of the techniques for making log writes atomic that are described on page 23.

8. We treat `Init` and `Crash` as atomic, mainly for convenience in writing invariants and abstraction functions.In real life they do several disk operations, so we have to lock out external invocations while they are running.

9. We ignore the possibility of errors.




```
MODULE CopyingFS EXPORTS PN, Sync =                  % implements File, uses Disk

TYPE DA          =  Nat                              % Disk Address in bytes
                    WITH "+":=DAAdd, "-":=DASub}
     LE          =  SEQ DA                           % Linear Extent
     Data        =  File.Data

     X           =  File.X
     F           =  [le, size: X]                    % size = # of bytes

     PN          =  String WITH [...]                % Path Name
     D           =  PN -> F

     Item        =  (DBChange + DChange + D + Pad)   % item on the disk
     DBChange    =  [pn, x, db]                      % db is data at x in file pn
     DChange     =  [pn, dOp, x]                     % x only for SetSize
     DOp         =  ENUM[create, delete, setSize]
     Pad         =  [size: X]                        % For filling up a DB;
                                                     % Pad{x}.enc.size = x.

     IDA         =  [item, da]
     SI          =  SEQ IDA                          % for parsing the busy region

     Root        =  [dDA: DA, busy: DA]              % assume encoding < DBSize

CONST
     DBSize      := Disk.DBSize
     diskSize    := 1000000
     rootDA      := 0
     bottom      := rootDA + DBSize                  % smallest DA outside root
     top         := (DBSize * diskSize) AS DA
     ringSize    := top - bottom
     endDB       := DB{...}                          % starts unlike any Item

VAR                                                  % All volatile; stable data is on disk.
     d           :  D    := {}
     sDDA        :  DA   := bottom                   % = ReadRoot().dDA
     sBusy       :  DA   := Bottom                   % = ReadRoot().busy
     busy        :  DA   := bottom
     free        :  DA   := bottom
     next        :  DA   := bottom                   % DA to write buf at
     buf         :  Data := {}                       % waiting to be written
     disk                                            % the disk

ABSTRACTION FUNCTION File.d = ( LAMBDA (pn) -> File.F =
% The file is the data pointed to by the DA's in its F.
     VAR f := d(pn), diskData := + :(f.le * ReadOneDB) |
          RET diskData.seg(0, f.size) )

ABSTRACTION FUNCTION File.oldDs = { SD(), d }

INVARIANT 1: ( ALL f :IN d.rng | f.le.size * DBSize >= f.size )
% The blocks of a file have enough space for the data. From FSImpl.
```

The reason that `oldDs` doesn't contain any intermediate states is that the stable state changes only in a `Sync`, which shrinks `oldDs` to just `d`.

During normal operation we need to have the variables that keep track of the region boundaries and the stable directory arranged in order around the disk ring, and we need to maintain this condition after a crash. Here are the relevant current and post-crash variables, in order (see below for `MinSpace`). The 'post-crash' column gives the value that the 'current' expression will have after a crash.

Current	*Post-crash*	
`busy`	`sBusy`	start of busy region
`sDDA`	`sDDA`	most recent stable `d`
`next`		end of stable busy region
`free`	`next`	end of busy region
`free + minSpace()`	`next + minSpace()`	end of cushion for writes

In addition, the stable busy region should start and end before or at the start and end of the volatile busy region, and the stable directory should be contained in both. Also, the global variables that are supposed to equal various stable variables (their names start with '`s`') should in fact do so. The analysis that leads to this invariant is somewhat tricky; I hope it's right.

```
INVARIANT 2:
       IsOrdered((SEQ DA){next + MinSpace(), sBusy, busy, sDDA, next, free,
                          free + MinSpace(), busy})
    /\ EndDA() = next /\ next//DBSize = 0 /\ Root{sDDA, sBusy} = ReadRoot()
```

Finally,

> The busy region should contain all the items pointed to from `DA`'s in `d` or in global variables.
>
> The directory on disk at `sDDA` plus the changes between there and `free` should agree with `d`.
>
> This condition should still hold after a crash.

```
INVARIANT 3:
       IsAllGood(ParseLog(busy, buf), d)
    /\ IsAllGood(ParseLog(sBusy, {}), SD())
```

The following functions are mainly for the invariants, though they are also used in crash recovery. `ParseLog` expects that the disk from `da` to the next `DB` with contents `endDB`, plus `data`, is the encoding of a sequence of `Item`'s, and it returns the sequence `SI`, each `Item` paired with its `DA`. `ApplyLog` takes an `SI` that starts with a `D` and returns the result of applying all the changes in the sequence to that `D`.

```
FUNC ParseLog(da, data) -> SI = VAR si, end: DA |
% Parse the log from da to the next endDB block, and continue with data.
       + :(si * (\ ida | ida.item.enc) = ReadData(da, end - da) + data
    /\ (ALL n :IN si.dom - {0} |
          si(n).da = si(n-1).da + si(n-1).item.enc.size)
    /\ si.head.da = da
    /\ ReadOneDB(end) = endDB => RET si

FUNC ApplyLog(si) -> D = VAR d' := si.head.item AS D |
% si must start with a D. Apply all the changes to this D.
    DO VAR item := si.head.item |
       IF   item IS DBChange  => d'(item.pn).le(item.x/DBSize) := si.head.da
       []   item IS DChange => d' := ...                % details omitted
       [*]  SKIP                                        % ignore D and Pad
       FI; si := si.tail
```



```
    OD; RET d'

FUNC IsAllGood(si, d') -> Bool = RET
% All d' entries point to DBChange's and si agrees with d'
       (ALL da, pn, item | d'!pn /\ da IN d'(pn).le /\ IDA{item, da} IN si
                          ==> item IS DBChange)
    /\ ApplyLog(si) = d'

FUNC SD() -> D = RET ApplyLog(ParseLog(sDDA), {})
% The D encoded by the Item at sDDA plus the following DChange's

FUNC EndDA() -> DA = VAR ida := ParseLog(sDDA).last |
% Return the DA of the first endDB after sDDA, assuming a parsable log.
    RET ida.da + ida.item.enc.size
```

The minimum free space we need is room for writing out `d` when we are about to overwrite the last previous copy on the disk, plus the wasted space in a disk block that might have only one byte of data, plus the `endDB`.

```
FUNC MinSpace() -> Int = RET d.enc.size + (DBSize-1) + DBsize
```

The following `Read` and `Write` procedures are much the same as they would be in `FSImpl`, where we omitted them. They are full of boring details about fitting things into disk blocks; we include them here for completeness, and because the way `Write` handles allocation is an important part of `CopyingFS`. We continue to omit the other `File` procedures like `SetSize`, as well as the handling in `ApplyLog` of the `DChange` items that they create.

```
PROC Read(pn, x, size: X) -> Data =
    VAR f     := d(pn),
        size  := {{size, f.size - x}.min, 0}.max,     % the available bytes
        n     := x/DBSize,                            % first block number
        nSize := NumDBs(x, size),                     % number of blocks
        blocks:= n .. n + nSize -1,                   % blocks we need in f.le
        data  := + :(blocks * f.le * ReadItem *       % all data in these blocks
                     (\ item | (item AS DBChange).db)) |
            RET data.seg(x//DBSize, size)             % the data requested

PROC Write(pn, x, data) = VAR f := d(pn) |
% First expand data to contain all the DB's that need to be written
    data := 0.fill(x - f.size) + data;                % add 0's to extend f to x
    x := {x, f.size}.min;                             % and adjust x to match
    IF  VAR y := x//DBSize | y # 0 =>                 % fill to a DB in front
        x := x - y; data := Read(pn, x, y) + data
    [*] SKIP FI;
    IF  VAR y := data.size//DBSize | y # 0 =>         % fill to a DB in back
        data + := Read(pn, x + data.size, DBSize - y)
    [*] SKIP FI;
    % Convert data into DB's, write it, and compute the new f.le
    VAR blocks := Disk.DToB(data), n := x/DBSize,
        % Extend f.le with 0's to the right length.
        le := f.le + LE.fill(0, x + blocks.size - le.size),
        i := 0 |
            DO blocks!i =>
                le(n + i) := WriteData(DBChange{pn, x, blocks(i)}.enc);
                x + := DBSize; i + := 1
            OD; d(pn).le := le
```

These procedures initialize the system and handle crashes. `Crash` is somewhat idealized; more realistic code would read the log and apply the changes to `d` as it reads them, but the logic would be the same.

```
PROC Init() = disk := disk.new(diskSize); WriteD()      % initially d is empty

PROC Crash() = <<                                       % atomic for simplicity
    CRASH;
    sDDA   := ReadRoot().sDDA; d := SD();
    sBusy    := ReadRoot().busy; busy := sBusy;
    free := EndDA(); next := free; buf := {} >>
```

These functions read an item, some data, or a single `DB` from the disk. They are boring. `ReadItem` is somewhat unrealistic, since it just chooses a suitable size for the `item` at `da` so that `Item.dec` works. In real life it would read a few blocks at `DA`, determine the length of the item from the header, and then go back for more blocks if necessary. It reads either from `buf` or from the disk, depending on whether `da` is in the write buffer, that is, between `next` and `free`.

```
FUNC ReadItem(da) -> Item = VAR size: X |
    RET Item.dec( (   DABetween(da, next, free) => buf.seg(da - next, size)
    [*]  ReadData(da, size) ) )

FUNC ReadData(da, size: X) -> Data =                    % 1 or 2 disk.read's
    IF   size + da <= top =>                            % Int."+", not DA."+"
        % Read the necessary disk blocks, then pick out the bytes requested.
        VAR data := disk.read(LE{da/DBSize, NumDBs(da, size)}) |
            RET data.seg(da//DBSize, size)
    [*]  RET ReadData(da, top - da) + ReadData(bottom, size - (top - da))

PROC ReadOneDB(da) = RET disk.read(LE{da/DBSize, 1}))
```

`WriteData` writes some data to the disk. It is not boring, since it includes the write buffering, the cleaning, and the space bookkeeping. The writes are buffered in `buf`, and `Sync` does the actual disk write. In this module `Sync` is only called by `WriteD`, but since it's a procedure in `File` it can also be called by the client. When `WriteData` needs space it calls `Clean`, which does the basic cleaning step of copying a single item. There should be a check for a full disk, but we omit it. This check can be done by observing that the loop in `WriteData` advances `free` all the way around the ring, or by keeping track of the available free space. The latter is fairly easy, but `Crash` would have to restore the information as part of its replay of the log.

These write procedures are the only ones that actually write into `buf`. `Sync` and `WriteRoot` below are the only procedures that write the underlying disk.

```
PROC WriteData(data) -> DA =                            % just to buf, not disk
    DO IsFull(data.size) => Clean() OD;
    buf + := data; VAR da := free | free + := data.size; RET da

PROC WriteItem(item) = VAR q := item.enc | buf + := q; free + := q.size
% No check for space because this is only called by Clean, WriteD.

PROC Sync() =
% Actually write to disk, in 1 or 2 disk.write's (2 if wrapping).
% If we will write past sBusy, we have to update the root.
    IF (sBusy - next) + (free - next) <= MinSpace() => WriteRoot()[*]   SKIP FI;
    % Pad buf to even DB's. A loop because one Pad might overflow current DB.
    DO VAR z := buf.size//DBSize | z # 0 => buf := buf + Pad{DBSize-z}.enc OD;.
    buf := buf + endDB;                                 % add the end marker DB
    <<    % atomic for simplicity
```



```
    IF   buf.size + next < top => disk.write(next/DBSize, buf)
    [*]  disk.write(next  /DBSize, buf.seg(0       , top-next  ));
         disk.write(bottom/DBSize, buf.sub(top-next, buf.size-1))
    FI;
    >>; free := next + buf.size - DBSize; next := free; buf := {}
```

The constraints on using free space are that `Clean` must not cause writes beyond the stable `sBusy` or into a disk block containing `Item`'s that haven't yet been copied. (If `sBusy` is equal to `busy` and in the middle of a disk block, the second condition might be stronger. It's necessary because a write will clobber the whole block.) Furthermore, there must be room to write an `Item` containing `d`. Invariant 2 expresses all this precisely. In real life, of course, `Clean` would be called in the background, the system would try to maintain a fairly large amount of free space, and only small parts of `d` would be dirty. `Clean` drops `DChange`'s because they are recorded in the `D` item that must appear later in the busy region.

```
FUNC IsFull(size: X) -> Bool = RET busy - free < MinSpace() + size

PROC Clean() = VAR item := ReadItem(busy) |             % copy the next item
    IF   item IS DBChange /\ d(item.pn).le(item.x/DBSize) = busy =>
         d(item.pn).le(item.x/DBSize) := free; WriteItem(item)
    []   item IS D /\ da = sDDA => WriteD()              % the latest D
    [*]  SKIP                                           % drop DChange, Pad
    FI; busy := busy + item.enc.size

PROC WriteD() =
% Called only from Clean and Init. Could call it more often to speed up recovery
%, after DO busy - free < MinSpace() => Clean() OD to get space.
    sDDA := free; WriteItem(d); Sync(); WriteRoot()
```

The remaining utility functions read and write the root, convert byte sizes to `DB` counts, and provide arithmetic on `DA`'s that wraps around from the top to the bottom of the disk. In real life we don't need the arithmetic because the disk is divided into segments and items don't cross segment boundaries; if they did the cleaner would have to do something quite special for a segment that starts with the tail of an item.

```
FUNC ReadRoot() -> Root = VAR root, pad |
    ReadOneDB(rootDA) = root.enc + pad.enc => RET root

PROC WriteRoot() = << VAR pad, db | db = Root{sDDA, busy}.enc + pad.enc =>
    disk.write(rootDA, db); sBusy := busy >>

FUNC NumDBs(da, size: X) -> Int = RET (size + da//DBSize + DBSize-1)/DBSize
% The number of DB's needed to hold size bytes starting at da.

FUNC DAAdd(da, i: Int) -> DA = RET ((da - bottom + i) // ringSize) + bottom

FUNC DASub(da, i: Int) -> DA = RET ((da - bottom - i) // ringSize) + bottom
% Arithmetic modulo the data region. abs(i) should be < ringSize.

FUNC DABetween(da, da1, da2) -> Bool = RET da = da1 \/ (da2 - da1) < (da1 - da)

FUNC IsOrdered(s: SEQ DA) -> Bool =
    RET (ALL i :IN s.dom - {0, 1} | DABetween(s(i-1), s(i-2), s(i)))

END CopyingFS
```

8. Generalizing Abstraction Functions

In this handout, we give a number of examples of specs and code for which simple abstraction functions (of the kind we studied in handout 6 on abstraction functions) don't exist, so that the abstraction function method doesn't work to show that the code satisfies the spec. We explain how to generalize the abstraction function method so that it always works.

A Spec program defines a set of possible histories, and for safety we only care about finite histories. The idea of the abstraction function is that it works on the state at a single time (transition) in a history. This means, informally, that we may need to encode information from the past or the future in the current state.

We begin with an example in which the spec maintains state that doesn't actually affect its behavior. Optimized code can simulate the spec without having enough state to generate all the state of the spec. By adding *history variables* to the code, we can extend its state enough to define an abstraction function, without changing its behavior. An equivalent way to get the same result is to define an *abstraction relation* from the code to the spec.

Next we look at code that simulates a spec without taking exactly one step for each step of the spec. As long as the *external* behavior is the same in each step of the simulation, an abstraction function (or relation) is still enough to show correctness, even when an arbitrary number of transitions in the spec correspond to a single transition in the code.

Finally, we look at an example in which the spec makes a non-deterministic choice earlier than the choice is exposed in the external behavior. Code may make this choice later, so that there is no abstraction relation that generates the premature choice in the spec's state. By adding *prophecy variables* to the code, we can extend its state enough to define an abstraction function, without changing its behavior. An equivalent way to get the same result is to use an abstraction relation and define a *backward simulation* from the code to the spec.

If we avoided extra state, too few or too many transitions, and premature choices in the spec, the simple abstraction function method would always work. You might therefore think that all these problems are not worth solving, because it sounds as though they are caused by bad choices in the way the spec is written. But this is wrong. A spec should be written to be as clear as possible to the clients, not to make it easy to prove the correctness of code for. The reason for these priorities is that we expect to have many more clients for the spec than implementers. The examples below should make it clear that there are good reasons to write specs that create these problems for abstraction functions. Fortunately, with all three of these extensions we can always find an abstraction function to show the correctness of any code that actually is correct.

A statistical database

Consider the following spec of a "statistical database" module, which maintains a collection of values and allows the size, mean, and variance of the collection to be extracted. Recall that the mean m of a sequence db of size $n > 0$ is just the average $\frac{\sum_i db(i)}{n}$, and the variance is

$\frac{\sum_i (db(i)-m)^2}{n} = \frac{\sum_i db(i)^2}{n} - m^2$. (We make the standard assumptions of commutativity, associativity, and distributivity for the arithmetic here.)

```
MODULE StatDB [ V WITH {Zero: ()->V, "+": (V,V)->V, (V,V)->V, "-": (V,V)->V,
                        "/": (V,Int)->V} ]
              EXPORT Add, Size, Mean, Variance =

VAR db          :  SEQ V := {}                          % a multiset; don't care about the order

APROC Add(v) = << db + := {v}; RET >>

APROC Size() -> Int = << RET db.size >>

APROC Mean() -> V RAISES {empty} = <<
    IF db = {} => RAISE empty [*] VAR sum := (+ : db) | RET sum/Size() FI >>

APROC Variance() -> V RAISES {empty} = <<
    IF  db = {} => RAISE empty
    [*] VAR avg := Mean(), sum := (+ : {v :IN db || (v - avg)**2}) |
        RET sum/Size()
    FI >>

END StatDB
```

This spec is a very natural one that follows directly from the definitions of mean and variance.

The following code for the `StatDB` module does not retain `db`. Instead, it keeps track of the size, sum, and sum of squares of the values in `db`. Simple algebra shows that this is enough to compute the mean and variance incrementally, as `StatDBImpl` does.

```
MODULE StatDBImpl                                       % implements StatDB
    [ V WITH {Zero: ()->V, "+": (V,V)->V, (V,V)->V, "-": (V,V)->V,
              "/": (V,Int)->V} ]
     EXPORT Add, Size, Mean, Variance =

VAR count       := 0
    sum         := V.Zero()
    sumSquare   := V.Zero()

APROC Add(v) = << count + := 1; sum + := v; sumSquare + := v**2; RET >>

APROC Size() -> Int = << RET count >>

APROC Mean() -> V RAISES {empty} =
    << IF count = 0 => RAISE empty [*] RET sum/count FI >>

APROC Variance() -> V RAISES {empty} = <<
    IF  count = 0 => RAISE empty
    [*] RET sumSquare/count - Mean()**2
    FI >>

END StatDBImpl
```

`StatDBImpl` implements `StatDB`, in the sense of trace set inclusion. However we cannot prove this using an abstraction function, because each nontrivial state of the code corresponds to many states of the spec. This happens because the spec contains more information than is needed to generate its external behavior. In this example, the states of the spec could be partitioned into equivalence classes based on the possible future behavior: two states are equivalent if they give

rise to the same future behavior. Then any two equivalent states yield the same future behavior of the module. Each of these equivalence classes corresponds to a state of the code.

To get an abstraction function we must add history variables, as explained in the next section.

History variables

The problem in the StatDB example is that the spec states contain more information than the code states. A *history variable* is a variable that is added to the state of the code *T* in order to keep track of the extra information in the spec *S* that was left out of the code. Even though the code has been optimized *not* to retain certain information, we can put it back in to prove the code correct, as long as we do it in a way that does not change the behavior of the code. What we do is to construct new code *TH* (*T* with *H*istory) that has the *same* behavior as *T*, but a bigger state. If we can show that *TH* implements *S*, it follows that *T* implements *S*, since traces of *T* = traces of *TH* ⊆ traces of *S*.

In this example, we can simply add an extra state component db (which is the entire state of StatDB) to the code StatDBImpl, and use it to keep track of the entire collection of elements, that is, of the entire state of StatDB. This gives the following module:

```
MODULE StatDBImplH ... =                          % implements StatDB

VAR  count       := 0                             % as before
     sum         := V.Zero()                      % as before
     sumSquare   := V.Zero()                      % as before
     db          :  SEQ V := {}                   % history: state of StatDB

APROC Add(v) = <<
     count + := 1; sum + := v; sumSquare + := v**2;
     db + := {v}; RET >>

% The remaining procedures are as before

END StatDBImplH
```

All we have done here is to record some additional information in the state. We have not changed the way existing state components are initialized or updated, or the way results of procedures are computed. So it should be clear that this module exhibits the *same* external behaviors as the code StatDBImpl given earlier. Thus, if we can prove that StatDBImplH implements StatDB, then it follows immediately that StatDBImpl implements StatDB.

However, we can prove that StatDBImplH implements StatDB using an abstraction function. The abstraction function, AF, simply discards all components of the state *except* db. The following invariant of StatDBImplH describes how db is related to the other state:

```
INVARIANT
        count     = db.size
     /\ sum       = (+ : db)
     /\ sumSquare = (+ : {v :IN db || v**2})
```

That is, count, sum and sumSquare contain the number of elements in db, the sum of the elements in db, and the sum of the squares of the elements in db, respectively.

With this invariant, it is easy to prove that AF is an abstraction function from StatDBImplH to StatDB. This proof shows that the abstraction function is preserved by every step, because the

only variable in StatDB, db, is changed in exactly the same way in both modules. The interesting thing to show is that the Size, Mean, and Variance operations produce the same results in both modules. But this follows from the invariant.

In general, we can augment the state of code for with additional components, called *history variables* (because they keep track of additional information about the history of execution), subject to the following constraints:

1. Every initial state has at least one value for the history variables.
2. No existing step is disabled by the addition of predicates involving history variables.
3. A value assigned to an existing state component does not depend on the value of a history variable. One important case of this is that a return value does not depend on a history variable.

These constraints guarantee that the history variables simply record additional state information and do not otherwise affect the behaviors exhibited by the module. If the module augmented with history variables is correct, the original module without the history variables is also correct, because they have the same traces.

This definition is formulated in terms of the underlying state machine model. However, most people think of history variables as syntactic constructs in their own particular programming languages; in this case, the restrictions on their use must be defined in terms of the language syntax.

In the StatDB example, we have simply added a history variable that records the entire state of the spec. This is not necessary; sometimes there might be only a small piece of the state that is missing from the code. However, the brute-force strategy of using the entire spec state as a history variable will work whenever any addition of history variables will work.

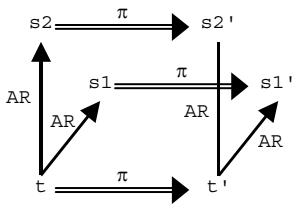
Abstraction relations

If you don't like history variables, you can define an *abstraction relation* between the code and the spec; it's the same thing in different clothing.

An abstraction relation is a simple generalization of an abstraction function, allowing several states in *S* to correspond to the same state in *T*. An abstraction relation is a subset of *states*(*T*) × *states*(*S*) that satisfies the following two conditions:

1. If *t* is any initial state of *T*, then there is an initial state *s* of *S* such that (*t*, *s*) ∈ *R*.
2. If *t* and *s* are reachable states of *T* and *S* respectively, with (*t*, *s*) ∈ *R*, and (*t*, π, *t*') is a step of *T*, then there is a step of *S* from *s* to some *s*', having the same trace, and with (*t*', *s*') ∈ *R*.

The picture illustrates the idea; it is an elaboration of the picture for an abstraction function in handout 6. It shows t related to s1 and s2, and an action π taking each of them into a state related to t'.



It turns out that the same theorem holds as for abstraction functions:

Theorem 1: If there is an abstraction relation from *T* to *S*, then *T* implements *S*, that is, every trace of *T* is a trace of *S*.

The reason is that for *T* to simulate *S* it isn't necessary to have a function from *T* states to *S* states; it's sufficient to have a relation. A way to think of this is that the two modules, *T* and *S*, are running in parallel. The execution is driven by module *T*, which executes in any arbitrary way. *S* follows along, producing the same externally visible behavior. The two conditions above guarantee that there is always some way for *S* to do this. Namely, if *T* begins in any initial state *t*, we just allow *S* to begin in some related initial state *s*, as given by (1). Then as *T* performs each of its transitions, we mimic the transition with a corresponding transition of *S* having the same externally visible behavior; (2) says we can do so. In this way, we can mimic the entire execution of *T* with an execution of *S*.

An abstraction relation for `StatDB`

Recall that in the `StatDB` example we couldn't use an abstraction function to prove that the code satisfies the spec, because each nontrivial state of the code corresponds to many states of the spec. We can capture this connection with an abstraction relation. The relation that works is described in Spec[1] as:

```
TYPE T = [count: Int, sum: V, sumSquare: V]          % state of StatDBImpl
     S = [db: SEQ V]                                 % state of StatDB

FUNC AR(t, s) -> Bool =
     RET    db.size = count
         /\ (+ : db)) = sum
         /\ (+ : {v :IN db || v**2}) = sumSquare
```

The proof that `AR` is an abstraction relation is straightforward. We must show that the two properties in the definition of an abstraction relation are satisfied. In this proof, the abstraction relation is used to show that every response to a size, mean or variance query that can be given by `StatDBImpl` can also be given by `StatDB`. The new state of `StatDB` is uniquely determined by the code of `StatDB`. Then the abstraction relation in the prior states together with the code performed by both modules shows that the abstraction relation still holds for the new states.

[1] This is one of several ways to represent a relation, but it is the standard one in Spec. Earlier we described the abstraction relation as a set of pairs (*t*, *s*). In terms of `AR`, this set is `{t, s | AR(t, s) | (t, s)}` or simply `AR.set`, using one of Spec's built-in methods on predicates. Yet another way to write it is as a function `T -> SET S`. In terms of `AR`, this function is `(\ t | {s | AR(t, s)}` or simply `AR.setF`, using another built-in method. These different representations can be confusing, but different aspects of the relation are most easily described using different representations.

An abstraction relation for `MajReg`

Consider the abstraction function given for `MajReg` in handout 5. We can easily write it as an abstraction relation from `MajReg` to `Register`, not depending on the invariant to make it a function. Recall the types:

```
TYPE P           =  [V, N]                    % Pair of value and sequence number
     M           =  C -> P                    % Memory: a pair at each copy

FUNC AR(m, v) -> Bool = VAR n := m.rng.max.n | RET (P{v, n} IN m.rng)
```

For (1), suppose that *t* is any initial state of `MajReg`. Then there is some default value `v` such that all copies have value `v` and `n = 0` in *t*. Let *s* be the state of `Register` with value `v`; then *s* is an initial state of `Register` and (*t*, *s*) ∈ *AR*, as needed.

For (2), suppose that *t* and *s* are reachable states of `MajReg` and `Register`, respectively, with (*t*, *s*) ∈ *AR*, and (*t*, π, *t'*) a step of `MajReg`. Because *t* is a reachable state, it must satisfy the invariants given for `MajReg`. We consider cases, based on π. Again, the interesting cases are the procedure bodies.

Abstraction relations vs. history variables

Notice that the *invariant* for the history variable `db` above bears an uncanny resemblance to the *abstraction relation* `AR`. This is not an accident—the same ideas are used in both proofs, only they appear in slightly different places. The following table makes the correspondence explicit.

Abstraction relation to history variable	*History variable to abstraction relation*
Given an abstraction relation `AR`, define `TH` by adding the abstract state `s` as a state variable to `T`. `AR` defines an invariant on the state of `TH`: `AR(t, s)`.	Given `TH`, `T` extended with a history variable `h`, there's an invariant `I(t, h)` relating `h` to the state of `T`, and an abstraction function `AF(t, h) -> S` such that `TH` simulates `S`.
Define `AF((t, s)) = s`	Define `AR(t, s) =` `(EXISTS h | I(t, h) /\ AF(t, h) = s)` That is, `t` is related to `s` if there's a value for `h` in state `t` that `AF` maps to `s`.
For each step `(t, π, t')` of `T`, and `s` such that `AR(t, s)` holds, the abstraction relation gives us `s'` such that `(t, π, t')` simulates `(s, π, s')`. Add `((t, s), p, (t', s'))` as a transition of `TH`. This maintains the invariant.	For each step `(t, π, t')` of `T`, and `h` such that the invariant `I(t, h)` holds, `TH` has a step `((t, h), π, (t', h'))` that simulates `(s, π, s')` where `s = AF(t, h)` and `s' = AF(t', h')`. So `AR(t', s')` as required.

This correspondence makes it clear that any code that can be proved correct using history variables can also be proved correct using an abstraction relation, and vice-versa. Some people prefer using history variables because it allows them to use an abstraction function, which may be simpler (especially in terms of notation) to work with than an abstraction relation. Others prefer using an abstraction relation because it allows them to avoid introducing extra state components and explaining how and when those components are updated. Which you use is just a matter of taste.

Taking several steps in the spec

A simple generalization of the definition of an abstraction relation (or function) allows for the possibility that a particular step of *T* may correspond to more or less than one step of *S*. This is fine, as long as the externally-visible actions are the same in both cases. Thus this distinction is only interesting when there are internal actions.

Formally, a (generalized) abstraction relation *R* satisfies the following two conditions:

1. If *t* is any initial state of *T*, then there is an initial state *s* of *S* such that $(t, s) \in R$.
2. If *t* and *s* are reachable states of *T* and *S* respectively, with $(t, s) \in R$, and (t, π, t') is a step of *T*, then there is an *execution fragment* of *S* from *s* to some *s*', having the same trace, and with $(t', s') \in R$.

Only the second condition has changed, and the only difference is that an execution fragment (of any number of steps, including zero) is allowed instead of just one step, as long as it has the same trace, that is, as long as it looks the same from the outside. We generalize the definition of an abstraction function in the same way. The same theorem still holds:

Theorem 2: If there is a generalized abstraction function or relation from *T* to *S*, then *T* implements *S*, that is, every trace of *T* is a trace of *S*.

From now on in the course, when we say "abstraction function" or "abstraction relation", we will mean the generalized versions.

Some examples of the use of these generalized definitions appear in handout 7 on file systems, where there are internal transitions of code that have no counterpart in the corresponding specs. We will see examples later in the course in which single steps of code correspond to several steps of the specs.

Here, we give a simple example involving a large write to a memory, which is done in one step in the spec but in individual steps in the code. The spec is:

```
MODULE RWMem [A, V] EXPORT BigRead, BigWrite =

TYPE M           =  A -> V
VAR  memory      :  M

FUNC BigRead() -> M = RET memory

APROC BigWrite(m: M) = << memory := m; RET >>

END RWMem
```

The code is:

```
MODULE RWMemImpl [A, V] EXPORT BigRead, BigWrite =

TYPE M           =  A -> V
VAR  memory      :  M
     pending     :  SET A := {}

FUNC BigRead() -> M = pending = {} => RET memory

PROC BigWrite(m) =
     << pending := memory.dom >>;
     DO << VAR a | a IN pending => memory(a) := m(a); pending - := {a} >> OD;
     RET

END RWMemImpl
```


We can prove that `RWMemImpl` implements `RWMem` using an abstraction function. The state of `RWMemImpl` includes program counter values to indicate intermediate positions in the code, as well as the values of the ordinary state components. The abstraction function cannot yield partial changes to memory; therefore, we define the function as if an entire abstract `BigWrite` occurred at the point where the *first* change occurs to the memory occurs in `RWMemImpl`. (Alternative definitions are possible; for instance, we could have chosen the *last* change.) The abstraction function is defined by:

> `RWMem.memory = RWMemImpl.memory` unless `pending` is nonempty. In this case `RWMem.memory = m`, where `BigWrite(m)` is the active `BigWrite` that made `pending` nonempty. `RWMem`'s pc for an active `BigRead` is the same as that for `RWMemImpl`. `RWMem`'s pc for an active `BigWrite` is before the body if the pc in `RWMemImpl` is at the beginning of the body; otherwise it is after the body.

In the proof that this is an abstraction function, all the atomic steps in a `BigWrite` of `RWMemImpl` except for the step that writes to memory correspond to no steps of `RWMem`. This is typical: code for usually has many more transitions than a spec, because the code is limited to the atomic actions of the machine it runs on, but the spec has the biggest atomic actions possible because that is the simplest to understand.

Note that the guard in `RWEMemImpl.BigRead` prevents a `BigRead` from returning an intermediate state of `memory`, which would be a transition not allowed by the spec. Of course this can't happen unless there is concurrency.

In this example, it is also possible to interchange the code and the spec, and show that `RWMem` implements `RWMemImpl`. This can be done using an abstraction function. In the proof that this is an abstraction function, the body of a `BigWrite` in `RWMem` corresponds to the entire sequence of steps comprising the body of the `BigWrite` in `RWMemImpl`.

Exercise: Add crashes to this example. The spec should contain a component `OldStates` that keeps track of the results of partial changes that could result from a crash during the current `BigWrite`. A `Crash` during a `BigWrite` in the spec can set the memory nondeterministically to any of the states in `OldStates`. A `Crash` in the code simply discards any active procedure. Prove the correctness of your code using an abstraction function. Compare this to the specs for file system crashes in handout 7.

Premature choice

In all the examples we have done so far, whenever we have wanted to prove that one module implements another (in the sense of trace inclusion), we have been able to do this using either an abstraction function or else its slightly generalized version, an abstraction relation. Will this always work? That is, do there exist modules *T* and *S* such that the traces of *T* are all included among the traces of *S*, yet there is no abstraction function or relation from *T* to *S*? It turns out that there do—abstraction functions and relations aren't quite enough.

To illustrate the problem, we give a very simple example. It is trivial, since its only point is to illustrate the limitations of the previous proof methods.

Example: Let NonDet be a state machine that makes a nondeterministic choice of 2 or 3. Then it outputs 1, and subsequently it outputs whatever it chose.

```
MODULE NonDet EXPORT Out =

VAR i := 0

APROC Out() -> Int = <<
     IF   i = 0 => BEGIN i := 2 [] i := 3 END; RET 1
     [*]  RET i FI >>

END NonDet
```

Let LateNonDet be a state machine that outputs 1 and then nondeterministically chooses whether to output 2 or 3 thereafter.

```
MODULE LateNonDet EXPORT Out =

VAR i := 0

APROC Out() -> Int = <<
     IF i = 0 => i := 1 [*] i = 1 => BEGIN i := 2 [] i := 3 END [*] SKIP FI;
     RET i >>

END LateNonDet
```

Clearly NonDet and LateNonDet have the same traces: `Out() = 1; Out() = 2; ...` and `Out() = 1; Out() = 3; ...`. Can we show the implements relationships in both directions using abstraction relations?

Well, we can show that NonDet implements LateNonDet with an abstraction function that is just the identity. However, no abstraction relation can be used to show that LateNonDet implements NonDet. The problem is that the nondeterministic choice in NonDet occurs before the output of 1, whereas the choice in LateNonDet occurs later, after the output of 1. It is impossible to use an abstraction relation to simulate an early choice with a later choice. If you think of constructing an abstract execution to correspond to a concrete execution, this would mean that the abstract execution would have to make a choice before it knows what the code is going to choose.

You might think that this example is unrealistic, and that this kind of thing never happens in real life. The following three examples show that this is wrong; we will study code for all of these examples later in the course. We go into a lot of detail here because most people find these situations very unfamiliar and hard to understand.

Premature choice: Reliable messages

Here is a realistic example (somewhat simplified) that illustrates the same problem: two specs for reliable channels, which we will study in detail later, in handout 26 on reliable messages. A reliable channel accepts messages and delivers them in FIFO order, except that if there is a crash, it may lose some messages. The straightforward spec drops some queued messages during the crash.

```
MODULE ReliableMsg [M] EXPORT Put, Get, Crash =

VAR q  : SEQ M := {}

APROC Put(m)      = << q + := {m} >>
APROC Get() -> M = << VAR m := q.head | q := q.tail; RET m >>
```



```
APROC Crash()     = << VAR q' | q' <<= q => q := q' >>
```
% Drop any of the queued messages (<<= is non-contiguous subsequence)

```
END ReliableMsg
```

Most practical code (for instance, the Internet's TCP protocol) has cases in which it isn't known whether a message will be lost until long after the crash. This is because they ensure FIFO delivery, and get rid of retransmitted duplicates, by numbering messages sequentially and discarding any received message with an earlier sequence number than the largest one already received. If the underlying message transport is not FIFO (like the Internet) and there are two undelivered messages outstanding (which can happen after a crash), the earlier one will be lost if and only if the later one overtakes it. You don't know until the overtaking happens whether the first message will be lost. By this time the crash and subsequent recovery may be long since over.

The following spec models this situation by 'marking' the messages that are queued at the time of a crash, and optionally dropping any marked messages in Get.

```
MODULE LateReliableMsg [M] EXPORT Put, Get, Crash =

VAR q  : SEQ [m, mark: Bool] := {}

APROC Put(m)      = << q + := {m} >>
APROC Get() -> M =
     << DO VAR x := q.head | q := q.tail; IF x.mark => SKIP [] RET x.m FI OD >>

APROC Crash()     = << q := {x :IN q || x{mark := true}} >>
```
% Mark all the queued messages. This is a sequence, not a set constructor, so it doesn't reorder the messages.

```
END LateReliableMsg
```

Like the simple NonDet example, these two specs are equivalent, but we cannot prove that LateReliableMsg implements ReliableMsg with an abstraction relation, because ReliableMsg makes the decision about what messages to drop sooner, in Crash. LateReliableMsg makes this decision later, in Get, and so does the standard code.

Premature choice: Consensus

For another examples, consider the *consensus* problem of getting a set of process to agree on a single value chosen from some set of allowed values; we will study this problem in detail later, in handout 18 on consensus. The spec doesn't mention the processes at all:

```
MODULE Consensus [V] EXPORT Allow, Outcome =

VAR outcome     :  (V + Null) := nil                    % Data value to agree on

APROC Allow(v) = << outcome = nil => outcome := v [] SKIP >>
FUNC  Outcome() -> (V + Null) = RET outcome [] RET nil

END Consensus
```

This spec chooses the value to agree on as soon as the value is allowed. Outcome may return nil even after the choice is made because in distributed code it's possible that not all the participants have heard what the outcome is. Code for almost certainly saves up the allowed values and does a lot of communication among the processes to come to an agreement. The following spec has that form. It is more complicated than the first one (more state and more operations), and closer

to code, using an internal `Agree` action to model what the processes do in order to choose a value.

```
MODULE LateConsensus [V] EXPORT Allow, Outcome =

VAR  outcome     :  (V + Null) := nil                    % Data value to agree on
     allowed     :  SET V := {}

APROC Allow(v) = << allowed \/ := {v} >>

FUNC  Outcome() -> (V + Null) = RET outcome [] RET nil

APROC Agree() = << VAR v | v IN allowed /\ outcome = nil => outcome := v >>

END LateConsensus
```

It should be clear that these two modules have the same traces: a sequence of `Allow(x)` and `Outcome() = y` actions in which every `y` is either `nil` or the same value, and that value is an argument of some preceding `Allow`. But there is no abstraction relation from `LateConsensus` to `Consensus`, because there is no way for `LateConsensus` to come up with the outcome before it does its internal `Agree` action.

Note that if `Outcome` didn't have the option to return `nil` even after `outcome # nil`, these modules would not be equivalent, because `LateConsensus` would allow the behavior

```
Allow(1); Outcome()=nil, Allow(2), Outcome()=1
```

and `Consensus` would not.

Premature choice: Multi-word clock

Here is a third example of premature choice in a spec: reading a clock. The spec is simple:

```
MODULE Clock EXPORT Read =

VAR  t           :  Int                                  % the current time

THREAD Tick() = DO << t := t + 1 >> OD                   % demon thread advances t

PROC Read() -> Int = << RET t >>

END Clock
```

This is in a concurrent world, in which several threads can invoke `Read` concurrently, and `Tick` is a demon thread that is entirely internal. In that world there are three transitions associated with each invocation of `Read`: entry, body, and exit. The entry and exit transitions are external because `Read` is exported.

We may want code that allows the clock to have more precision than can be carried in a single memory location that can be read and written atomically. We could easily achieve this by locking the clock representation, but then a slow process holding the lock (for instance, one that gets pre-empted) could block other processes for a long time. A clever 'wait-free' code for `Read` (which appears in handout 17 on formal concurrency) reads the various parts of the clock representation one at a time and puts them together deftly to come up with a result which is guaranteed to be one of the values that `t` took on during this process. The following spec abstracts this strategy; it breaks `Read` down into two atomic actions and returns some value, non-deterministically chosen, between the values of `t` at these two actions.


```
MODULE LateClock EXPORT Read =

VAR  t           :  Int                                  % the current time

THREAD Tick() = DO << t := t + 1 >> OD                   % demon thread advances t

PROC Read() -> Int = VAR t1: Int |
     << t1 := t >>; << VAR t2 | t1 <= t2 /\ t2 <= t => RET t2 >>

END LateClock
```

Again both specs have the same traces: a sequence of invocations and responses from `Read`, such that for any two `Read`s that don't overlap, the earlier one returns a smaller value `tr`. In `Clock` the choice of `tr` depends on when the body of `Read` runs relative to the various `Tick`s. In `LateClock` the `VAR t2` makes the choice of `tr`, and it may choose a value of `t` some time ago. Any abstraction relation from `LateClock` to `Clock` has to preserve `t`, because a thread that does a complete `Read` exposes the value of `t`, and this can happen between any two other transitions. But `LateClock` doesn't decide its return value until its last atomic command, and when it does, it may choose an earlier value than the current `t`; no abstraction relation can explain this.

Prophecy variables

One way to cope with these examples and others like them is to use ad hoc reasoning to show that `LateSpec` implements `Spec`; we did this informally in each example above. This strategy is much easier if we make the transition from premature choice to late choice at the highest level possible, as we did in these examples. It's usually too hard to show directly that a complicated module that makes a late choice implements a spec that makes a premature choice.

But it isn't necessary to resort to ad hoc reasoning. Our trusty method of abstraction functions can also do the job. However, we have to use a different sort of auxiliary variable, one that can look into the future just as a history variable looks into the past. Just as we did with history variables, we will show that a module *TP* (*T* with *P*rophecy) augmented with a *prophecy variable* has the same traces as the original module *T*. Actually, we can show that it has the same *finite* traces, which is enough to take care of safety properties. It also has the same infinite traces provided certain technical conditions are satisfied, but we won't worry about this because we are not interested in liveness. To show that the traces are the same, however, we have to work *backward* from the end of the trace instead of forward from the beginning.

A prophecy variable guesses in advance some non-deterministic choice that *T* is going to make later. The guess gives enough information to construct an abstraction function to the spec that is making a premature choice. When execution reaches the choice that *T* makes non-deterministically, *TP* makes it deterministically according to the guess in the prophecy variable. *TP* has to choose enough different values for the prophecy variable to keep from ruling out any executions of *T*.

The conditions for an added variable to be a prophecy variable are closely related to the ones for a history variable, as the following table shows.

History variable	*Prophecy variable*
1. Every initial state has at least one value for the history variable.	1. *Every state* has at least one value for the prophecy variable.
2. No existing step is disabled by new guards involving a history variable.	2. No existing step is disabled *in the backward direction* by new guards involving a prophecy variable. More precisely, for each step (t, π, t') and state (t', p') there must be a p such that there is a step ((t, p), π, (t', p')).
3. A value assigned to an existing state component must not depend on the value of a history variable. One important case of this is that a return value must not depend on a history variable.	3. Same condition. A prophecy variable *can* affect what actions are enabled, subject to condition (2), but it can't affect how an action changes an existing state component.
	4. If t is an initial state of T and (t, p) is a state of TP, it must be an initial state.

If these conditions are satisfied, the state machine *TP* with the prophecy variable will have the same traces as the state machine *T* without it. You can see this intuitively by considering any finite execution of *T* and constructing a corresponding execution of *TP*, starting from the end. Condition (1) ensures that we can find a last state for *TP*. Condition (2) says that for each backward step of *T* there is a corresponding backward step of *TP*, and condition (3) says that in this step *p* doesn't affect what happens to *t*. Finally, condition (4) ensures that we end up in an initial state of *TP*.

Condition (3) is somewhat subtle. Unlike a history variable, a prophecy variable can appear in a guard and thus affect the control flow; condition (2) rules this out for history variables. That is, a particular choice made in setting a prophecy variable can decide what later actions are enabled. Condition (2) ensures that there is *some* choice for the prophecy variables that allows every sequence of actions that was possible in the unadorned program.

Let's review our examples and see how to add prophecy variables (that all start with `p`), marking the additions with boxes. For `LateNonDetP` we add `pI` that guesses the choice between 2 and 3. The abstraction function is just `NonDet.i = LateNonDetP.pI`.

```
VAR  i  := 0
     pI := 0

APROC Out() -> Int = <<
     IF   i = 0 => i := 1; BEGIN pI := 2 [] pI := 3 END
     [*]  i = 1 => BEGIN pI = 2 => i := 2 [] pI = 3 => i := 3 END [*] SKIP FI;
     RET i >>
```

For `LateReliableMsgP` we add a `pDead` flag to each marked message that forces `Get` to discard it. `Crash` chooses which `dead` flags to set. The abstraction function just discards the marks and the dead messages.

```
VAR  q  : SEQ [m, mark: Bool, pDead: Bool] := {}

% ABSTRACTION FUNCTION ReliableMsg.q = {x :IN LateReliableMsg.q | ~x.dead || x.m}
```



```
% INVARIANT (ALL i :IN q.dom | q(i).dead ==> q(i).mark)

APROC Get() -> M =
     << DO VAR x := q.head |
          q := q.tail; IF x.mark => SKIP [] ~ x.pDead => RET x.m FI OD >>

APROC Crash()    = << VAR pDeads: SEQ Bool | pDeads.size = q.size =>
     q := {x :IN q, pD :IN pDeads || x{mark := true, pDead := pD}
```

Alternatively, we can prophesy the entire state of `ReliableMsg` as we did with `db` in `StatDB`, which is a little less natural in this case:

```
VAR  pQ  : SEQ M := {}

% INVARIANT {x :IN q | ~ x.mark || x.m} <<= pQ /\ pQ <<= {x :IN q || x.m}

APROC Get() -> M =
     << DO VAR x := q.head |
          q := q.tail;
          IF x.mark /\ (pQ = {} \/ x.m # pQ.head) => SKIP
          [] pQ := pQ.tail; RET x.m
          FI OD >>

APROC Crash() =
     << VAR q' | q' <<= q => pQ := q'; q := {x :IN q || x{mark := true}} >>
```

For `LateConsensusP` we follow the example of `NonDet` and just prophesy the outcome in `Allow`. The abstraction function is `Consensus.outcome = LateConsensusP.pOutcome`

```
VAR  outcome     :  (V + Null) := nil                 % Data value to agree on
     pOutcome    :  (V + Null) := nil
     allowed     :  SET V := {}

% ABSTRACTION FUNCTION LateClock.t = pt

APROC Allow(v) =
     << allowed \/ := {v}; IF pOutcome = nil => pOutcome := v [] SKIP FI >>

APROC Agree() =
     << VAR v | v IN allowed /\ outcome = nil /\ v = pOutcome => outcome := v >>
```

For `LateClockP` we choose the result at the beginning of `Read`. The second command of `Read` has to choose this value, which means it has to wait until `Tick` has advanced `t` far enough. The transition of `LateClockP` that corresponds to the body of `Clock.Read` is the `Tick` that gives `t` the pre-chosen value. This seems odd, but since all these transitions are internal, they all have empty external traces, so it is perfectly OK.

```
VAR  t           :  Int                              % the current time
     pT          :  Int

PROC Read() -> Int = VAR t1: Int |
     << t1 := t; VAR t': Int | pT := t' >>;
     << VAR t2 | t1 <= t2 /\ t2 <= t /\ t2 = pT => RET t2 >>
```

Most people find it much harder to think about prophecy variables than to think about history variables, because thinking about backward execution does not come naturally. It's easy to see that it's harmless to carry along extra information in the history variables that isn't allowed to affect the main computation. A prophecy variable, however, *is* allowed to affect the main computation, by forcing a choice that was non-deterministic to be taken in a particular way. Condi-

tion (2) ensures that in spite of this, no traces of *T* are ruled out in *TP*. It requires us to use a prophecy variable in such a way that for any possible choice that *T* could make later, there's some choice that *TP* can make for the prophecy variable's value that allows *TP* to later do what *T* does.

Here is another way of looking at this. Condition (2) requires enough different values for the prophecy variables p_i to be carried forward from the points where they are set to the points where they are used to ensure that as they are used, any set of choices that *T* could have made is possible for some execution of *TP*. So for each command that uses a p_i to make a choice, we can calculate the set of different values of the p_i that are needed to allow all the possible choices. Then we can propagate this set back through earlier commands until we get to the one that chooses p_i, and check that it makes enough different choices.

Because prophecy variables are confusing, it's important to use them only at the highest possible level. If you write a spec *SE* that makes an early choice, and implement it with a module *T*, don't try to show that *T* satisfies *SE*; that will be too confusing. Instead, write another spec *SL* that makes the choice later, and use prophecy variables to show that *SL* implements *SE*. Then show that *T* implements *SL*; this shouldn't require prophecy. We have given three examples of such *SE* and *SL* specs; the implementations are given in later handouts.

Backward simulation

Just as we could use abstraction relations instead of adding history variables, we can use a different kind of relation, satisfying different start and step conditions, instead of prophecy variables. This new sort of relation also guarantees trace inclusion. Like an ordinary abstraction relation, it allows construction of an execution of the spec, working from an execution of the code. Not surprisingly, however, the construction works *backwards* in the execution of the code instead of forwards. (Recall the inductive proof for abstraction relations.) Therefore, it is called a *backward simulation*.

The following table gives the conditions for a backward simulation using relation *R* to show that *T* implements *S*, aligning each condition with the corresponding one for an ordinary abstraction relation. To highlight the relationship between the two kinds of abstraction mappings, an ordinary abstraction relation is also called a *forward simulation*.

Forward simulation	*Backward simulation*
1. If t is any initial state of *T*, then there is an initial state s of *S* such that $(t, s) \in R$.	1. If t is any reachable state of *T*, then there a state s of *S* such that $(t, s) \in R$.
2. If t and s are reachable states of *T* and *S* respectively, with $(t, s) \in R$, and (t, π, t') is a step of *T*, then there is an execution fragment of *S* from s to some s', having the same trace, and with $(t', s') \in R$.	2. If t' and s' are states of *T* and *S* respectively, with $(t', s') \in R$, (t, π, t') is a step of *T*, and t is reachable, then there is an execution fragment of *S* from some s to s', having the same trace, and with $(t, s) \in R$.
	3. If t is an initial state of *T* and $(t, s) \in R$ then s is an initial state of *S*.

(1) applies to any reachable state *t* rather than any initial state, since running backwards we can start in any reachable state, while running forwards we start in an initial state. (2) requires that

every backward (instead of forward) step of *T* be a simulation of a step of *S*. (3) is a new condition ensuring that a backward run of *T* ending in an initial state simulates a backward run of *S* ending in an initial state; since a forward simulation never ends, it has no analogous condition.

Theorem 3: If there exists a backward simulation from *T* to *S* then every *finite* trace of *T* is also a trace of *S*.

Proof: Start at the end of a finite execution and work backward, exactly as we did for forward simulations.

Notice that Theorem 3 only yields finite trace inclusion. That's different from the forward case, where we get infinite trace inclusion as well. Can we use backward simulations to help us prove general trace inclusion? It turns out that this doesn't always work, for technical reasons, but it works in two situations that cover all the cases you are likely to encounter:

- The infinite traces are exactly the limits of finite traces. Formally, we have the condition that for every sequence of successively extended finite traces of *S*, the limit is also a trace of *S*.
- The correspondence relation relates only finitely many states of *S* to each state of *T*.

In the `NonDet` example above, a backward simulation can be used to show that `LateNonDet` implements `NonDet`. In fact, the inverse of the relation used to show that `NonDet` implements `LateNonDet` will work. You should check that the three conditions are satisfied.

Backward simulations vs. prophecy variables

The same equivalence that holds between abstraction relations and history variables also holds between backward simulations and prophecy variables. The invariant on the prophecy variable becomes the abstraction relation for the backward simulation.

Completeness

Earlier we asked whether forward simulations always work to show trace inclusion. Now we can ask whether it is always possible to use either a forward or a backward simulation to show trace inclusion. The satisfying answer is that a *combination* of a forward and a backward simulation, one after the other, will always work, at least to show finite trace inclusion. (Technicalities again arise in the infinite case.) For proofs of this result and discussion of the technicalities, see the papers by Abadi and Lamport and by Lynch and Vondrager cited below.

History and further reading

The idea of abstraction functions has been around since the early 1970's. Tony Hoare introduced it in a classic paper (C.A.R. Hoare, Proof of correctness of data representations. *Acta Informatica* **1** (1972), pp 271-281). It was not until the early 1980's that Lamport (L. Lamport, Specifying concurrent program modules. *ACM Transactions on Programming Languages and Systems* **5**, 2 (Apr. 1983), pp 190-222) and Lam and Shankar (S. Lam and A. Shankar, Protocol verification via projections. *IEEE Transactions on Software Engineering* **SE-10**, 4 (July 1984), pp 325-342) pointed out that abstraction functions can also be used for concurrent systems.

People call abstraction functions and relations by various names. 'Refinement mapping' is popular, especially among European writers. Some people say 'abstraction mapping'.

History variables are an old idea. They were first formalized (as far as I know), in Abadi and Lamport, The existence of refinement mappings. *Theoretical Computer Science* **2**, 82 (1991), pp 253-284. The same paper introduced prophecy variables and proved the first completeness result. For more on backward and forward simulations see N. Lynch and F. Vondrager, Forward and backward simulations—Part I: Untimed systems. *Information and Computation* **121**, 2 (Sep. 1995), pp 214-233.

9. Atomic Semantics of Spec

This handout defines the semantics of the atomic part of the Spec language fairly carefully. It tries to be precise about all difficult points, but is sloppy about some things that seem obvious in order to keep the description short and readable. For the syntax and an informal account of the semantics, see the Spec reference manual, handout 4.

There are three reasons for giving a careful semantics of Spec:

1. To give a clear and unambiguous meaning for Spec programs.
2. To make it clear that there is no magic in Spec; its meaning can be given fairly easily and without any exotic methods.
3. To show the versatility of Spec by using it to define itself, which is quite different from the way we use it in the rest of the course.

This handout is divided into two parts. In the first half we describe semi-formally the essential ideas and most of the important details. Then in the second half we present the complete atomic semantics precisely, with a small amount of accompanying explanation.

Semi-formal atomic semantics of Spec[1]

Our purpose is to make it clear that there is no arm waving in the Spec notation that we have given you. A translation of this into fancy words is that we are going to study a formal semantics of the Spec language.

Now that is a formidable sounding term, and if you take a course on the semantics of programming languages (6.821—Gifford, 6.830J—Meyer) you will learn all kinds of fancy stuff about bottom and stack domains and fixed points and things like that. You are not going to see any of that here. We are going to do a very simple minded, garden-variety semantics. We are just going to explain, very carefully and clearly, how it is that every Spec construct can be understood, as a transition of a state machine. So if you understand state machines you should be able to understand all this without any trouble.

One reason for doing this is to make sure that we really do know what we are talking about. In general, descriptions of programming languages are not in that state of grace. If you read the Pascal manual or the C manual carefully you will come away with a number of questions about exactly what happens if I do this and this, questions which the manual will not answer adequately. Two reasonably intelligent people who have studied it carefully can come to different conclusions, argue for a long time, and not be able to decide what is the right answer by reading the manual.

There is one class of mechanisms for saying what the computer should do that often does answer your questions precisely, and that is the instruction sets of computers (or, in more modern language, the architecture). These specs are usually written as state machines with fairly simple

[1] These semi-formal notes take the form of a dialogue between the lecturer and the class. They were originally written by Mitchell Charity for the 1992 edition of this course, and have been edited for this handout.

transitions, which are not beyond the power of the guy who is writing the manual to describe properly. A programming language, on the other hand, is not like that. It has much more power, generality, and wonderfulness, and also much more room for confusion.

Another reason for doing this is to show you that our methods can be applied to a different kind of system than the ones we usually study, that is, to a programming language, a notation for writing programs or a notation for writing specs. We are going to learn how to write a spec for that particular class of computer systems. This is a very different application of Spec from the last one we looked at, which was file systems. For describing a programming language, Spec is not the ideal descriptive notation. If you were in the business of giving the semantics of programming languages, you wouldn't use Spec. There are many other notations, some of them better than Spec (although most are far worse). But Spec is good enough; it will do the job. And there is a lot to be said for just having one notation you can use over and over again, as opposed to picking up a new one each time. There are many pitfalls in devising a new notation.

Those are the two themes of this lecture. We are going to get down to the foundations of Spec, and we are going to see another, very different application of Spec, a programming language rather than a file system.

For this lecture, we will only talk about the sequential or atomic semantics of Spec, not about concurrent semantics. Consider the program:

```
                    x, y = 0
thread 1:                                   thread 2:
<<  x := 3 >>                               << z := x + y >>
<<  y := 4 >>
```

In the concurrent world, it is possible to get any of the values 0, 3, or 7 for `z`. In the sequential world, which we are in today, the only possible values are 0 and 7. It is a simpler world. We will be talking later (in handout 17 on formal concurrency) about the semantics of concurrency, which is unavoidably more complicated.

In a sequential Spec program, there are three basic constructs (corresponding to sections 5, 6, and 7 of the reference manual):

Expressions
Commands
Routines

For each of these we will give a *meaning function*, `ME`, `MC`, and `MR`, that takes a fragment of Spec and yields its meaning as some sort of Spec value.[2] We shall see shortly exactly what type of values these are.

In order to describe what each of these things means, we first of all need some notion of what kind of thing the meaning of an expression or command might be. Then we have to explain in detail the exact meaning of each possible kind of expression. The basic technique we use is the standard one for a situation where you have things that are made up out of smaller things: *structural induction*.

[2] It's common in the literature to denote the meaning of a syntactic construct `S` by [[`S`]],

The idea of structural induction is this. If you have something which is made up of an `A` and a `B`, and you know the meaning of each, and have a way to put them together, you know how to get the meaning of the bigger thing.

Some ways to put things together in Spec:

```
A , B
A ; B
a + b
A [] B
```

State

What are the meanings going to be? Our basic notion is that what we are doing when writing a Spec program is describing a state machine. The central properties of a state machine are that it has states and it has transitions.

A state is a function from names to values: `State: Name -> Value`. For example:

```
VAR  x: Int
     y: Int
```

If there are no other variables, the state simply consists of the mapping of the names `"x"` and `"y"` to their corresponding values. Initially, we don't know what their values are. Somehow the meaning we give to this whole construct has to express that.

Next, if we write `x := 1`, after that the value of `x` is `1`. So the meaning of this had better look something like a transition that changes the state, so that no matter what the `x` was before, it is `1` afterwards. That's what we want this assignment to mean.

Spec is much simpler than C. In particular, it does not have "references" or "pointers". When you are doing problems, if you feel the urge to call `malloc`, the correct thing to do is to make a function whose range is whatever sort of thing you want to allocate, and then choose a new element of the domain that isn't being used already. You can use the integers or any convenient sort of name for the domain, that is, to name the values. If you define a `CLASS`, Spec will do this for you automatically.

So the state is just these name-to-value mappings.

Names

Spec has a module structure, so that names have two parts, the module name and the simple name. When referring to a variable in another module, you need both parts.

```
MODULE M                    MODULE N

VAR  x                      M.x := 3
     x := 3
     ...
     M.x := 3
```

To simplify the semantics, we will use `M.x` as the name everywhere. In other words, to apply the semantics you first must go through the program and replace every `x` declared in the current module `M` with `M.x`. This converts all references to global variables into these two part names, so that each name refers to exactly one thing. This transformation makes things simpler to describe

and understand, but uglier to read. It doesn't change the meaning of the program, which could have been written with two part names in the first place.

All the global variables have these two part names. However, local variables are not prefixed by the module name:

```
PROC
    VAR i | ... i
```

This is how we tell the global state apart from the local state. Global state names have dots, local state names do not.

Question: Can modules be nested?

No. Spec is meant to be suitable for the kinds of specs and code that we do in this course, which are no more than moderately complex. Features not really needed to write our specs are left out to keep it simpler.

Expressions

What should the meaning of an expression be? Note that expressions do *not* affect the state.

The type for the meaning of an expression is `S -> V`: an expression is a function from state to value (we ignore for now the possibility that an expression might raise an exception). It can be a partial function, since Spec does not require that all expressions be defined. But it has to be a function—we require that expressions are deterministic. We want determinism so something like `f(x) = f(x)` always comes out true. Reasoning is just too hard if this isn't true. If a function were to be nondeterministic then obviously this needn't come out true, since the two occurrences could yield different values. (The classic example of a nondeterministic function is a random number generator.)

So, **expressions are deterministic and do not affect state.**

Question: What about assignments?

Assignments are not expressions. If you have been programming in C, you have the weird idea that assignments are expressions. Spec, however, takes a hard line that expressions must be deterministic or functional; that is, their values depend only on the state. This means that functions, which are the abstraction of expressions, are not allowed to affect the state. The whole point of an assignment is to change the state, so an assignment cannot be an expression.

There are three types of expressions:

Type	*Example*	*Meaning*
constant	`1`	`(\ s | 1)`
variable	`x`	`(\ s | s("x"))`
function invocation	`f(x)`	next sub-section

(The type of these lambda's is not quite right, as we will see later).

Note that we have to keep the Spec in which we are writing the semantics separate from the Spec of the semantics we are describing. Therefore, we had to write `s("x")` instead of just `x`, because it is the `x` of the target Spec we are talking about, not the `x` of the describing Spec.

The third type of expression is function invocation. We will only talk about functions with a single argument. If you want a function with two arguments, you can make one by combining the two arguments into a tuple or record, or by currying: defining a function of the first argument that returns a function of the second argument. This is a minor complication that we ignore.

What about `x + y`? This is just shorthand for `T."+"(x, y)`, where `T` is the type of `x`. Everything that is not a constant or a variable is an invocation. This should be a familiar concept for those of you who know Scheme.

Semantics of function invocation

What are the semantics of function invocation? Given a function `T -> U`, the correct type of its meaning is `(T, S) -> U`, since the function can read the state but not modify it. Next, how are we going to attach a meaning to an invocation `f(x)`? Remember the rule of structural induction. In order to explain the meaning of a complicated thing, you are supposed to build it out of the meaning of simpler things. We know the meaning of `x` and of `f`. We need to come up with a map from states to values that is the meaning of `f(x)`. That is, we get our hands on the meaning of `f` and the meaning of `x`, and then put them together appropriately. What is the meaning of `f`? It is `s("f")`. So,

`f(x)` means `... s("f") ... s("x") ...`

How are we going to put it together, remembering the type we want for `f(x)`, which is `S -> U`?

`f(x)` means `(\ s | s("f") (s("x"), s)) )`

Now this could be complete nonsense, for instance if `s("f")` evaluates to an integer. If `s("f")` isn't a function then this doesn't typecheck. But there is no doubt about what this means if it is legal. It means invoke the function.

That takes care of expressions, because there are no other expressions besides these. Structural induction says you work your way through all the different ways to put little things together to make big things, and when you have done them all, you are finished.

Question: What about undefined functions?

Then the `(T, S) -> U` mapping is partial.

Question: Is `f(x) = f(x)` if `f(x)` is undefined?

No, it's undefined. "Undefined" is not a value; instead, when an expression is undefined the command that contains it fails, as we are about to see.

Commands

What is the type of the meaning of a command? Well, we have states and values to play with, and we have used up `S -> V` on expressions. What sort of thing is a command? It's a transition from one state to another.

Expressions: `S -> V`

Commands: `S -> S ?`

This is good for a subset of commands. But what about this one?

```
x := 1 [] x := 2
```

Is its meaning a function from states to states? No, from states to *sets* of states. It can't just be a function. It has to be a relation. Of course, there are lots of ways to code relations as functions. The way we use is:

Commands: `(S, S) -> Bool`

There is a small complication because Spec has exceptions, which are useful for writing many kinds of specs, not to mention programs. So we have to deal with the possibility that the result of a command is not a garden-variety state, but involves an exception.

To handle this we make a slight extension and invent a thing called an *outcome*, which is very much like a state except that it has some way of coding that an exception has happened. Again, there are many ways to code that. The way we use is that an outcome has the same type as a state: it's a function from names to values. However, there are a couple of funny names that you can't actually write in the program. One of them is `$x`, and we adopt the convention that if `o("$x") = ""` (empty string), then `o` is a garden-variety state. If `o("$x") = "exception-name"`, then there is that exception in outcome `o`. Some Spec commands, in particular `";"` and `EXCEPT`, do something special if one of their pieces produces an exception. This convention defines the meaning of the outcome component `$x`.

How do we say that `o` is related to `s`? The function returns `true`. We are encoding a relation between states and outcomes as a function from a state and outcome to a `Bool`. The function is supposed to give back `true` exactly when the relation holds.

So the meaning of a command has the relation type `(S, O) -> Bool`. We call this type `ATr`, for Atomic TRansition. Why isn't it `(O, O) -> Bool`? The reason is that a command never *starts* in an outcome that isn't a state; instead, when a command yields an outcome that isn't a state that affects the outcome of the containing command; see the discussion of `";"` and `EXCEPT` below for examples of this.

Now we just work our way through the command constructs (with an occasional digression).

Commands — assignment

```
x := 1
```

or in general

```
variable := expression
```

What we have to come up with for the meaning is an expression of the form

```
(\ s, o | ...)
```

So when does the relation hold for `x := exp`? Well, perhaps when `o(x) = exp`? (`ME` is the meaning function for expressions.)

```
o("x") = ME(e)(s)
```

. This is a start, since the valid transition


```
x=0                 x=1
          ->
y=0                 y=0
```

would certainly be allowed. But what others would be allowed? What about:

```
x=0                 x=1
          ->
y=0                 y=94
```

It would also be allowed, so this can't be quite right. Half right, but missing something important. You have to say that you don't mess around with the rest of the state. The way you do that is to say that the outcome is equal to the state except at the variable.

```
o = s{"x" -> ME(e)(s)}
```

This is just a Spec function constructor, of the form `f{arg -> value}`. Note that we are using the semantics of expressions that we defined in the previous section.

Aside—an alternate encoding for commands

As we said before, there are many ways to code the command relation. Another possibility is:

Commands: `S -> SET O`

This encoding seems to make the meanings of commands clumsier to write, though it is entirely equivalent to the one we have chosen.

There is a third approach, which has a lot of advantages: write predicates on the state values. If `x` and `y` are the state variables in the pre-state, and `x'` and `y'` the state variables in the post-state, then

```
(x' = 1 /\ y' = y)
```

is another way of writing

```
o = s{"x" -> 1}
```

In fact, this approach is another way of writing programs. You could write everything just as predicates. (Of course, you could also write everything in the ugly `o = s{...}` form, but that would look pretty awful. The predicates don't look so bad.)

Sometimes it's actually nice to do this. Say you want to write the predicate that says you can have any value at all for `x`. The Spec

```
VAR z | x := z
```

is just

```
(y' = y)
```

(in the simple world where the only state variables are `x` and `y`). This is much simpler than the previous, rather inscrutable, piece of program. So sometimes this predicate way of doing things can be a lot nicer, but in general it seems to be not as satisfactory, mainly because the `y'=y` stuff clutters things up a lot.

That was just an aside, to let you know that sometimes it's convenient to describe the things that can go on in a spec using predicates rather than functions from state pairs to `Bool`. There is more discussion of alternative ways to represent relations in the section on relations in handout 3.

Commands — routine invocation `p(x)`

What are the semantics of routine invocation? Well, it has to do something with `s`. The idea is that `p` is an abstraction of a state transition, so its meaning will be a relation of type `ATr`. What about the argument `x`? There are many ways to deal with it. Our way is to use another pseudo-variable `$a` to pass the argument and get back the result.

The meaning of `p(e)` is going to be

```
(\ s, o |          (s                                    Take the state,
                     {"$a" -> ME(e)(s)}                  append the argument,
          ME(p)(s)                                       get p's meaning
                                        , o))            and invoke it
```

or, writing the whole thing on one line in the normal way,

```
(\ s, o | ME(p)(s)(s{"$a" -> ME(e)(s)}, o))
```

What does this say? This invocation relates a state to an outcome if, when you take that state, and modify its `$a` component to be equal to the value of the argument, the meaning of the routine relates that state to the outcome. Another way of writing this, which isn't so nested and might be clearer, would be to introduce an intermediate state `s'`. Now we have to use `LAMBDA`:

```
(LAMBDA (s, o)->Bool = VAR s' = s{"$a" -> ME(e)(s)} | RET ME(p)(s)(s', o))
```

These two are exactly the same thing. The invocation relates `s` to `o` iff the routine relates `s'` to `o`, where `s'` is just `s` with the argument passing component modified. `$a` is just a way of communicating the argument value to the routine.

Question: Why use `ME(p)(s)` rather than `MR`?

`MR` is the meaning function for routines, that is, it turns the *syntax* of a routine declaration into a function on states and arguments that is the meaning of that syntax. We would use `MR` if `p` were a `FUNC`. But `p` is just a variable (of course it had better be bound to a routine value, or this won't typecheck), that is, an expression, so the proper meaning is the one for expressions, which is `ME`.

Aside—an alternate encoding for invocation

Here is a different way of communicating the argument value to the function; you can skip this section if you like. We could take the view that the routine definition

```
PROC P(i: Int) = ...
```

is defining a whole flock of different commands, one for every possible argument value. Then we need to pick out the right one based on the argument value we have. If we coded it this way (and it is merely a coding thing) we would get:

```
ME(p)(s)(ME(e)(s)) (s,o)
```

This says, first get `ME(p)`, the meaning of `p`. This is no longer a transition but a function from argument values to transitions, because the idea is that for every possible argument value, we are going to get a different meaning for the routine, namely what that routine does when given that

particular argument value. So we pass it the argument value `ME(e)(s)`, and invoke the resulting transition.

These two alternatives are based on different choices about how to code the meaning of routines. If you code the meaning of a routine simply as a transition, then Spec picks up the argument value out of the magic `$a` variable. But there is nothing mystical going on here. Setting `$a` corresponds exactly to what we would do if we were designing a calling sequence. We would say "I am going to pass the argument in register 1". Here, register 1 is `$a`.

The second approach is a little bit more mystical. We are taking more advantage of the wonderful abstract power and generality that we have. If someone writes a factorial function, we will treat it as an infinite supply of different functions with no arguments; one computes the factorial of 1, another the factorial of 2, another the factorial of 3, and so forth. In `ME(p)(s)(ME(e)(s))(s, o)`, `ME(p)(s)` is the infinite supply, `ME(e)(s)` is the argument that picks out a particular function, to which we finally pass `(s, o)`.

However, there are lots of other ways to do this. One of the things which makes the semantics game hard is that there are many choices you can make. They don't really make that much difference, but they can create a lot of confusion, because

- a bad choice can leave you in a briar patch of notation,
- you can get confused about what choice was made, and
- every author uses a slightly different scheme.

So, while this

```
RET ME(p) (S) (S("$a" -> ME(e) (s)),o)
```

and this

```
VAR s' := s{"$a" -> ME(e)(s)} | RET ME(p)(s) (s',o)
```

are two ways of writing exactly the same thing, this

```
RET ME(p)(s)(ME(e)(s)) (s,o)
```

is different, and only makes sense with a different choice about what the meaning of a function is. The latter is more elegant, but we use the former because it is less confusing.

Stepping back from these technical details, what the meaning function is doing is taking an expression and producing its meaning. The expression is a piece of syntax, and there are a lot of possible ways of coding the syntax. Which exact way we choose isn't that important.

Now we return to the meanings of Spec commands.

Commands — `SKIP`

```
(\ s, o | s = o)
```

In other words, the outcome after `SKIP` is the same as the pre-state. Later on, in the formal half of the handout, we give a table for the commands which takes advantage of the fact that there is a lot of boilerplate—the `(\ s, o | ...)` stuff is always the same, and so is the treatment of exceptions. So the table just shows, for each syntactic form, what goes after the `|`.

Commands — HAVOC

```
(\ s, o | true)
```

In other words, after HAVOC you can have any outcome. Actually this isn't quite good enough, since we want to be able to have any *sequence* of outcomes. We deal with this by introducing another magic state component $havoc with a Bool value. Once $havoc is true, any transition can happen, including one that leaves it true and therefore allows havoc to continue. We express this by adding to the command boilerplate the disjunct s("$havoc"), so that if $havoc is true in s, any command relates s to any o.

Now for the compound commands.

Commands — c1 [] c2

```
MC(c1)          MC(c2)

      (s, o)          (s, o)

              \/
```

or on one line,

```
MC(c1)(s, o) \/ MC(c2)(s, o)
```

Non-deterministic choice is the 'or' of the relations.

Commands — c1 [*] c2

It is clear we should begin with

```
MC(c1)(s, o) \/ …
```

But what next? One possibility is

```
                ~ MC(c1)(s, o) /\ ...
```

This is in the right direction, but not correct. Else means that if there is no *possible* outcome of c1, then you get to try c2. So there are two possible ways for an else to relate a state to an outcome. One is for c1 to relate the state to the outcome, the other is that there is no possible way to make progress with c1 in the state, and c2 to relates the state to the outcome.

The correct encoding is

```
MC(cl)(s,o) \/ (ALL o' | ~ MC(cl) (s, o')) /\ MC(c2)(s,o) )
```

Commands — c1 ; c2

Although the meaning of semicolon may seem intuitively obvious, it is more complex than one might first suspect—more complicated than "or", for instance, even though "or" is less familiar. We interpreted the command c1 [] c2 as MC(cl) \/ MC(c2). Because semicolon is a sequential composition, it requires that our semantics move through an intermediate state.

If these were functions (if we could describe the commands as functions) then we could simply describe a sequential composition as (F2 (Fl s)). However, because Spec is not a functional language, we need to compose relations, in other words, to establish an intermediate state as a precursor to the final output state. As a first attempt, we might try:


```
(LAMBDA (s, o) -> Bool = RET
    (EXISTS o' | MC(cl)(s, o') /\ MC(c2)(o', o)))
```

In words, this says that you can get from s to o via c1 ; c2 if there exists an intermediate state o' such that c1 takes you from s to o' and c2 takes you from o' to o. This is indeed the composition of the relations, which we can write more concisely as MC(c1) * MC(c2). But is this always the meaning of ";"? In particular, what if c1 produces an exception?

When c1 produces an exception, we should *not* execute c2. Our first try does not capture that possibility. To correct for this, we need to verify that o' is a normal state. If it is an exceptional state, then it is the result of the composition and we ignore c2.

```
    (EXISTS o' | MC(cl)(s, o') /\ (   ~IsX(o') /\ MC(c2)(o', o)
                                   \/  IsX(o') /\ o' = o))
```

Commands — c1 EXCEPT xs => c2

Now, what if we have a handler for the exception? If we assume (for simplicity) that all exceptions are handled, we simply have the complement of the semicolon case. If there's an exception, then do c2. If there's no exception, do *not* do c2. We also need to include an additional check to insure that the exception considered is an element of the exception set—that is to say, that it is a handled exception.

```
    (EXISTS o' | MC(cl)(s, o') /\
              (   ((~IsX(o') \/ ~o'("$x") IN xs) /\ o' = o)
               \/    IsX(o') /\  o'("$x") IN xs) /\ MC(c2)(o'{"$x" -> ""}, o)
    )
```

So, with this semantics for handling exceptions, the meaning of:

```
(c1 EXCEPT xs => c2); c3
```

is

if normal	do c1, no c2, do c3
if exception, handled	do c1, do c2, do c3
if exception and not handled	do c1, no c2, no c3

Commands — VAR id: T | c0

The idea is "there exists a value for id such that c0 succeeds". This intuition suggests something like

```
    (EXISTS v :IN T |      MC(c0)(s{"id" -> v}, o))
```

However, if we look carefully, we see that id is left defined in the output state o. (Why is this bad?) To correct this omission we need to introduce an intermediate state o' from which we may arrive at the final output state o where id is undefined.

```
(EXISTS v :IN T, o' | MC(c0)(s("id" -> v}, o') /\ o = o'(id -> })
```

Routines

In Spec, routines include functions, atomic procedures, and procedures. For simplicity, we focus on atomic procedures. How do we think about APROCs?

We know that the body of an APROC describes transitions from its input state to its output state. Given this transition, how do we handle the results? We previously introduced a pseudo name $a to which a procedure's argument value is bound. The caller also collects the value from $a after the procedure body's transition. Refer to the definition of MR below for a more complete discussion.

In reality, Spec is more complex because it attempts to make RET more convenient by allowing it to occur anywhere in a routine. To accommodate this, the meaning of RET e is to set $a to the value of e and then raise the special exception $RET, which is handled as part of the invocation.

Formal atomic semantics of Spec

In the rest of the handout, we describe the meaning of atomic Spec commands in complete detail, except that we do not give precise meanings for the various expression forms other than lambda expressions; for the most part these are drawn from mathematics, and their meanings should be clear. We also omit the detailed semantics of modules, which is complicated and uninteresting.

Overview

The semantics of Spec are defined in three stages: expressions, atomic commands, and non-atomic commands (treated in handout 17 on formal concurrency). For the first two there is no concurrency: expressions and atomic commands are atomic. This makes it possible to give their meanings quite simply:

Expressions as *functions* from states to results, that is, values or exceptions.

Atomic commands as *relations* between states and outcomes: a command relates an initial state to every possible outcome of executing the command in the initial state.

An outcome maps names (treated as strings) to values. It also maps three special strings that are not program names (we call them pseudo-names):
$a, which is used to pass argument and result values in an invocation;
$x, which records an exceptional outcome;
$havoc, which is true if any sequence of later outcomes is possible.

A state is a normal outcome, that is, an outcome which is not exceptional; it has $x=noX. The looping outcome of a command is encoded as the exception $loop; since this is not an identifier, you can't write it in a handler.

The state is divided into a *global* state that maps variables of the form m.id (for which id is declared at the top level in module m) and a *local* state that maps variables of the form id (those whose scope is a VAR command or a routine). Routines share only the global state; the ones defined by LAMBDA also have an initial local state, while the ones declared in a routineDecl start with an empty local state. We leave as an exercise for the reader the explicit construction of the global state from the collection of modules that makes up the program.

We give the meaning of a Spec program using Spec itself, by defining functions ME, MC, and MR that return the meaning of an expression, command, and routine. However, we use only the functional part of Spec. Spec is not ideally suited for this job, but it is serviceable and by using it we avoid introducing a new notation. Also, it is instructive to see how the task is writing this particular kind of spec can be handled in Spec.

You might wonder how this spec is related to code for Spec, that is, to a compiler or interpreter. It does look a lot like an interpreter. As with other specs written in Spec, however, this one is not practical code because it uses existential quantifiers and other forms of non-determinism too freely. Most of these quantifiers are just there for clarity and could be replaced by explicit computations of the needed values without much difficulty. Unfortunately, the quantifier in the definition of VAR does not have this property; it actually requires a search of all the values of the specified type. Since you have already seen that we don't know how to give practical code for Spec, it shouldn't be surprising that this handout doesn't contain one.

Note that before applying these rules to a Spec program, you must apply the syntactic rewriting rules for constructs like VAR id := e and CLASS that are given in the reference manual. You must also replace all global names with their fully qualified forms, which include the defining module, or Global for names declared globally (see section 8 of the reference manual).

Terminology

We begin by giving the types and special values used to represent the Spec program whose meaning is being defined. We use two methods of functions, + (overlay) and restrict, that are defined in section 9 of the reference manual.

```
TYPE V          =  (Routine + ...)                    % Value
     Routine    =  aTr                                % defined as the last type below

     Id         =  String                             % Identifer
                   SUCHTHAT (EXISTS c: Char, s1: String, s2: String |
                         id = {c} + s1 + s2 /\ c IN letter + digit
                      /\ s1.rng <= letter\/digit\/{'_'} /\ s2.rng <= {'''} )
     Name       =  String
                   SUCHTHAT name IN ids \/ globals \/ {"$a", "$x", "$havoc"}
     X          =  String                             % eXception
                   SUCHTHAT x IN ids \/ {noX, retX, loopX, typeX}
     XS         =  SET X                              % eXception Set

     O          =  Name -> V WITH {isX:=OIsX}         % Outcome
     S          =  O SUCHTHAT ~ o.isX                 % State
     ATr        =  (S, O) -> Bool                     % Atomic Transition

CONST
     letter     := "ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz".rng
     digit      := "0123456789".rng
     ids        := {id | true}
     globals    := {id1, id2 | id1 + "." + id2}
     noX        := ""
     retX       := "$ret"
     loopX      := "$loop"
     typeX      := "$type error"
     trueV      :  V                                  % the value true

FUNC OIsX(o) -> Bool = RET o("$x") # noX              % o.isX
```

To write the meaning functions we need types for the representations of the main non-terminals of the language: id, name, exceptionSet, type, exp, cmd, routineDecl, module, and

program. Rather than giving the detailed representation of these types or a complete set of operations for making and analyzing their values, we write C« c1 [] c2 » for a command composed from subcommands c1 and c2 with [], and so forth for the rest of the command forms. Similarly we write E« e1 + e2 » and R« FUNC Succ(x: INT)->INT = RET x+1 » for the indicated expression and function, and so forth for the rest of the expression and routine forms. This notation makes the spec much more readable. Id, Name, and XS are declared above.

```
TYPE T          =  SET V                              % Type
     E          =  [...]                              % Expression
     C          =  [...]                              % Command
     R          =  [id, ...]                          % RoutineDecl
     Mod        =  [id, tops: SET TopLevel]           % Module
     TopLevel   =  (R + ...)                          % module toplevel decl
     Prog       =  [ms: SET Mod, ts: SET TopLevel]    % Program
```

The meaning of an id or var is just the string, of an exceptionSet the set of strings that are the exceptions in the set, of a type the set of values of the type. For the other constructs there are meaning functions defined below: ME for expressions and MC and MR for atomic commands and routines. The meaning functions for module, toplevel, and program are left as exercises.

Expressions

An expression maps a state to a value or exception. Evaluating an expression does not change the state. Thus the meaning of expressions is given by a partial function ME with type E->S->(V + X); that is, given an expression, ME returns a function from states S to results (values V or exceptions X). ME is defined informally for all of the expression forms in section 5 of the reference manual. The possible expression forms are literal, variable, and invocation. We give formal definitions only for invocations and LAMBDA literals; they are written in terms of the meaning of commands, so we postpone them to the next section

Type checking

For type checking to work we need to ensure that the value of an expression always has the type of the expression (that is, is a member of the set of values that is the meaning of the type). We do this by structural induction, considering each kind of expression. The type checking of return values ensures that the result of an invocation will have its declared type. Literals are trivial, and the only other expression form is a variable. A variable declared with VAR is initialized to a value of its type. A formal parameter of a routine is initialized to an actual by an invocation, and the type checking of arguments (see MR below) ensures that this is a value of the variable's type. The value of a variable can only be changed by assignment.

An assignment var := e requires that the value of e have the type of var. If the type of e is not equal to the type of var because it involves a union or a SUCHTHAT, this check can't be done statically. To take account of this and to ensure that the meaning of expressions is independent of the static type checking, we assume that in the context var := e the expression e is replaced by e AS t, where t is the declared type of var. The meaning of e AS t in state s is ME(e)(s) if that is in t (the set of values of type t), and the exception typeX otherwise; this exception can't be handled because it is not named by an identifier and is therefore a fatal error.

We do not give practical code for the type check itself, that is, the check that a value actually is a member of the set of values of a given type. Such code would require too many details about how values are represented. Note that what many people mean by "type checking" is a proof that

every expression in a program always has a result of the correct type. This kind of completely static type checking is not possible for Spec; the presence of unions and SUCHTHAT makes it undecidable. Sections 4 and 5 of the reference manual define what it means for one type to fit another and for a type to be suitable. These definitions are a sketch of how to code as much static type checking as Spec easily admits.

Command	*Predicate*
`SKIP`	`o = s`
`HAVOC`	`true`
`RET e`	`o = s{"$x" -> retX, $a -> ME(e)(s)}`
`RET`	`o = s{"$x" -> retX}`
`RAISE id`	`o = s{"$x" -> "id"}`
`e1(e2)`	`( EXISTS r: Routine | r = ME(e1)(s) /\ r(s{"$a" -> ME(e2)(s)}, o) )`
`var := e [1]`	`o = s{var -> ME(e)(s)}`
`var := e1(e2) [1]`	`MC(C« e1(e2); var := $a »)(s, o)`
`e => c0`	`ME(e)(s) = trueV /\ MC(c0)(s, o)`
`c1 [] c2`	`MC(c1)(s, o) \/ MC(c2)(s, o)`
`c1 [*] c2`	`MC(c1)(s, o) \/ ( MC(c2)(s, o) /\ ~(EXISTS o' | MC(c1)(s, o')) )`
`c1 ; c2`	`MC(c1)(s, o) /\ o .isX \/ ( EXISTS o' | MC(c1)(s, o') /\ ~ o'.isX /\ MC(c2)(o',o ) )`
`c1 EXCEPT xs => c2`	`MC(c1)(s, o) /\ ~ o ("$x") IN xs \/ ( EXISTS o' \| MC(c1)(s, o') /\ o'("$x") IN xs /\ MC(c2)(o'{"$x" -> noX}, o) )`
`VAR id: T | c0`	`( EXISTS v, o' | v IN T /\ MC(c0)(s {id -> v}, o') /\ o = o'{id -> } )`
`VAR id: T := e | c0`	`MC(C«VAR id: T | id = e => c0»)(s, o)`
`<< c0 >>`	`MC(c0)(s, o)`
`IF c0 FI`	`MC(c0)(s, o)`
`BEGIN c0 END`	`MC(c0)(s, o)`

`DO c0 OD` is the fixed point of the equation `c = c0; c [*] SKIP`

[1] The first case for `assignment` applies only if the right side is not an `invocation` of an `APROC`. Because an invocation of an `APROC` can have side effects, it needs different treatment.

Table 1: The predicates that define `MC(command)(s, o)` when there are no exceptions raised by expressions at the top level in `command`, and `$havoc` is false.

Atomic commands

An atomic command relates a state to an outcome; in other words, it is defined by an `ATr` (atomic transition) relation. Thus the meaning of commands is given by a function `MC` with type `C->ATr`, where `ATr = (S, O) -> Bool`. We can define the `ATr` relation for each command by a predicate: a command relates state `s` to outcome `o` iff the predicate on `s` and `o` is true. We give the predicates in table 1 and explain them informally below; the predicates apply provided there are no exceptions.

Here are the details of how to handle exceptions and how to actually define the `MC` function. You might want to look at the predicates first, since the meat of the semantics is there.

The table of predicates has been simplified by omitting the boilerplate needed to take account of `$havoc` and of the possibility that an expression is undefined or yields an exception. If a command containing expressions `e1` and `e2` has predicate `P` in the table, the full predicate for the command is

```
    s("$havoc")                                          % anything if $havoc
\/    ME(e1)!s /\ ME(e2)!s                               % no outcome if undefined
   /\ (   ME(e1)(s) IS V /\ ME(e2)(s) IS V /\ P
       \/ ME(e1)(s) IS X /\ o = s{ "$x" -> ME(e1)(s) }
       \/ ME(e2)(s) IS X /\ o = s{ "$x" -> ME(e2)(s) } )
```

If the command contains only one expression `e1`, drop the terms containing `e2`. If it contains no expressions, the full predicate is just the predicate in the table.

Once we have the full predicates, it is simple to give the definition of the function `MC`. It has the form

```
FUNC MC(c) -> ATr =
    IF
    ...
    []  VAR var, e | c = «var := e» =>
        RET (\ o, s | full predicate for this case )
    ...
    []  VAR c1, c2 | c = «c1  ; c2» =>
        RET (\ o, s | full predicate for this case )
    ...
    FI
```

Now to explain the predicates. First we do the simple commands, which don't have subcommands. All of these that don't involve an invocation of an `APROC` are deterministic; in other words, the relation is a function. Furthermore, they are all total unless they involve an invocation that is partial.

A `RET` produces the exception `retX` and leaves the returned value in `$a`.

A `RAISE` yields an exceptional outcome which records the exception `id` in `$x`.

An `invocation` relates `s` to `o` iff the routine which is the value of `e1` (produced by `ME(e1)(s)`) does so after `s` is modified to bind `"$a"` to the actual argument; thus `$a` is used to communicate the value of the actual to the routine.

An `assignment` leaves the state unchanged except for the variable denoted by the left side, which gets the value denoted by the right side. Recall that assignment to a compo-

nent of a function, sequence, or record variable is shorthand for assignment of a suitable constructor to the entire variable, as described in the reference manual. If the right side is an `invocation` of a procedure, the value assigned is the value of `$a` in the outcome of the invocation; thus `$a` also communicates the result of the invocation back to the invoker.

Now for the compound commands; their meaning is defined in terms of the meaning of their subcommands.

A guarded command `e => c` has the same meaning as `c` except that `e` must be true.

A choice relates `s` to `o` if either part does.

An else `c1 [*] c2` relates `s` to `o` if `c1` does or if `c1` has no outcome and `c2` does.

A sequential composition `c1 ; c2` relates `s` to `o` if there is a suitable intermediate state, or if `o` is an exceptional outcome of `c1`.

`c1 EXCEPT xs=>c2` is the same as `c1` for a normal outcome or an exceptional outcome not in the exception set `xs`. For an exceptional outcome `o'` in `xs`, `c2` must relate `o'` as a normal state to `o`. This is the dual of the meaning of `c1 ; c2` if `xs` includes all exceptions.

`VAR id: t | c` relates `s` to `o` if there is a value `v` of type `t` such that `c` relates (`s` with `id` bound to `v`) to an `o'` which is the same as `o` except that `id` is undefined in `o`. It is this existential quantifier that makes the spec useless as an interpreter for Spec.

`<< ... >>`, `IF ... FI` or `BEGIN ... END` brackets don't affect `MC`.

The meaning of `DO c OD` can't be given so easily. It is the fixed point of the sequence of longer and longer repetitions of `c`.[3] It is possible for `DO c OD` to loop indefinitely; in this case it relates `s` to `s` with `"$x"->loopX`. This is not the same as relating `s` to no outcome, as `false => SKIP` does.

The multiple occurrences of `declInit` and `var` in `VAR declInit*` and `(varList):=exp` are left as boring exercises, along with routines that have several formals.

Routines

Now for the meaning of a routine. We define a meaning function `MR` for a `routineDecl` that relates the meaning of the routine to the meaning of the routine's body; since the body is a command, we can get its meaning from `MC`. The idea is that the meaning of the routine should be a relation of states to outcomes just like the meaning of a command. In this relation, the pseudo-name `$a` holds the argument in the initial state and the result in the outcome. For technical reasons, however, we define `MR` to yield not an `ATr`, but an `S->ATr`; a local state (`static` below) must be supplied to get the transition relation for the routine. For a `LAMBDA` this local state is the current state of its containing command. For a routine declared at top level in a module this state is empty.

The `MR` function works in the obvious way:

[3] For the details of this construction see G. Nelson, A generalization of Dijkstra's calculus, *ACM Trans. Programming Languages and Systems* **11**, 4, Oct. 1989, pp 517-562.

1. Check that the argument value in `$a` has the type of the formal.
2. Remove local names from the state, since a routine shares only global state with its invoker.
3. Bind the value to the formal.
4. Find out using `MC` how the routine body relates the resulting state to an outcome.
5. Make the invoker's outcome from the invoker's local state and the routine's final global state.
6. Deal with the various exceptions in that outcome.

   A `retX` outcome results in a normal outcome for the invocation if the result has the result type of the routine, and a `typeX` outcome otherwise.

   A normal outcome is converted to `typeX`, a type error, since the routine didn't supply a result of the correct type.

   An exception raised in the body is passed on.

```
FUNC MR(r) -> (S->ATr) = VAR id1, id2, t1, t2, xs, c0 |
       r = R« APROC id1(id2: t1)->t2 RAISES xs = << c0 >> »
    \/ r = R« FUNC  id1(id2: t1)->t2 RAISES xs =    c0    » =>
    RET (\ static: S | (\ s, o |
         s("$a") IN t1                                  % if argument typechecks
       /\ ( EXISTS g: S, s', o' |
              g = s.restrict(globals)                   % g is the current globals
            /\ s' = (static + g){id2 -> s("$a")}        % s' is initial state for c0
            /\ MC(c0)(s', o' )                          % apply c0
            /\ o = (s + o'.restrict(globals))           % restore old locals from s
                 {"$x" ->                               % adjust $x in the outcome
                   (    o'("$x") = retX =>
                          (    o'("$a") IN t2 => noX    % retX means normal outcome
                           [*] typeX )                  % if result typechecks;
                    [*] o'("$x") = noX => typeX         % normal outcome means typeX;
                    [*] o'("$x)                         % pass on exceptions
                  )
                 }
    \/ ~ s("$a") IN t1 /\ o = s{"$x" -> typeX}          % argument doesn't typecheck
    ) )                                                 % end of the two lambdas
```

We leave the meaning of a routine with no result as an exercise.

Invocation and `LAMBDA` expressions

We have already given in `MC` the meaning of invocations in commands, so we can use `MC` to deal with invocations in expressions. Here is the fragment of the definition of `ME` that deals with an `E` that is an invocation `e1(e2)` of a function. It is written in terms of the meaning `MC(C«e1(e2)»)` of the invocation as a command, which is defined above. The meaning of the command is an atomic transition `aTr`, a predicate on an initial state and an outcome of the routine. In the outcome the value of the pseudo-name `$a` is the value returned by the function. The definition given here discards any side-effects of the function; in fact, in a legal Spec program there can be no side-effects, since functions are not allowed to assign to non-local variables or call procedures.

```
FUNC ME(e) -> (S -> (V + X)) =
     IF
     ...
     [] VAR e1, e2 | e = E« e1(e2) » =>
     % if E is an invocation its meaning is this function from states to values
          VAR aTr := MC(C« e1(e2) ») |
            RET ( LAMBDA (s) -> V =
                % the command must have a unique outcome, that is, aTr must be a
                % function at s. See Relation in section 9 of the reference manual
                VAR o := aTr.func(s) | RET (~o.isX => o("$a") [*] o("$x")) )
     ...
     FI
```

The result of the expression is the value of `$a` in the outcome if it is normal, the value of `$x` if it is exceptional. If the invocation has no outcome or more than one outcome, `ME(e)(s)` is undefined.

The fragment of `ME` for `LAMBDA` uses `MR` to get the meaning of a `FUNC` with the same signature and body. As we explained earlier, this meaning is a function from a state to a transition function, and it is the value of `ME((LAMBDA ...))`. The value of `(LAMBDA ...)`, like the value of any expression, is the result of evaluating `ME((LAMBDA ...))` on the current state. This yields a transition function as we expect, and that function captures the local state of the `LAMBDA` expression; this is standard static scoping. .

```
     IF
     ...
     [] VAR signature, c0 | e = E« (LAMBDA signature = c0) » =>
          RET MR(R« FUNC id1 signature = c0 »)
     ...
     FI
```


10. Performance

Overview

This is not a course about performance analysis or about writing efficient programs, although it often touches on these topics. Both are much too large to be covered, even superficially, in a single lecture devoted to performance. There are many books on performance analysis[1] and a few on efficient programs[2].

Our goal in this handout is more modest: to explain how to take a system apart and understand its performance well enough for most practical purposes. The analysis is necessarily rather rough and ready, but nearly always a rough analysis is adequate, often it's the best you can do, and certainly it's much better than what you usually see, which is no analysis at all. Note that performance *analysis* is not the same as performance *measurement*, which is more common.

What is performance? The critical measures are *bandwidth* and *latency*. We neglect other aspects that are sometimes important: availability (discussed later when we deal with replication), connectivity (discussed later when we deal with switched networks), and storage capacity

When should you work on performance? When it's needed. Time spent speeding up parts of a program that are fast enough is time wasted, at least from any practical point of view. Also, the march of technology, also known as Moore's law, means that in 18 months from March 2006 a computer will cost the same but be twice as fast[3] and have twice as much RAM and four times as much disk storage; in five years it will be ten times as fast and have 100 times as much disk storage. So it doesn't help to make your system twice as fast if it takes two years to do it; it's better to just wait. Of course it still might pay if you get the improvement on new machines as well, or if a 4 x speedup is needed.

How can you get performance? There are techniques for making things faster:
- better algorithms,
- fast paths for common cases, and
- concurrency.

And there is methodology for figuring out where the time is going:
- analyze and measure the system to find the bottlenecks and the critical parameters that determine its performance, and
- keep doing so both as you improve it and when it's in service.

As a rule, a rough back-of-the-envelope analysis is all you need. Putting in a lot of detail will be a lot of work, take a lot of time, and obscure the important points.

[1] Try R. Jain, *The Art of Computer Systems Performance Analysis*, Wiley, 1991, 720 pp.
[2] The best one I know is J. Bentley, *Writing Efficient Programs*, Prentice-Hall, 1982, 170 pp.
[3] A new phenomenon as of 2006 is that the extra speed is likely to come mostly in the form of concurrency, that is, several processors on the chip, rather than a single processor that is twice as fast. This is because the improvements in internal processor architecture that have made it possible to use internal concurrency to speed up a processor that still behaves as though it is executing instructions sequentially are nearly played out.

What is performance: bandwidth and latency

Bandwidth and latency are usually the important metrics. Bandwidth tells you how much work gets done per second (or per year), and latency tells you how long something takes from start to finish: to send a message, process a transaction, or referee a paper. In some contexts it's customary to call these things by different names: throughput and response time, or capacity and delay. The ideas are exactly the same.

Here are some examples of communication bandwidth and latency on a single link. Note that all the numbers are in bytes/sec; it's traditional to quote bandwidths for some interconnects in bits/sec, so be wary of numbers you read.

Medium	*Link*	*Bandwidth*		*Latency*		*Width*
Pentium 4 chip	on-chip bus	30	GB/s	.4	ns	64
PC board	Rambus bus	1.6	GB/s	75	ns	16
	PCI I/O bus	533	MB/s	200	ns	32
Wires	Serial ATA (SATA)	300	MB/s	200	ns	1
	SCSI	40	MB/s	500	ns	32
LAN	Gigabit Ethernet	125	MB/s	100 +	µs	1
	Fast Ethernet	12.5	MB/s	100 +	µs	1
	Ethernet	1.25	MB/s	100 +	µs	1

Here are examples of communication bandwidth and latency through a switch that interconnects multiple links.

Medium	*Switch*	*Bandwidth*		*Latency*		*Links*
Pentium 4 chip	register file	180	GB/s	.4	ns	6
Wires	Cray T3E	122	GB/s	1	µs	2K
LAN	Ethernet switch	4	GB/s	4–100	µs	32
Copper pair	Central office	80	MB/s	125	µs	50K

Finally, here are some examples of other kinds of work, different from simple communication.

Medium	*Bandwidth*		*Latency*	
Disk	40	MB/s	10	ms
RPC on Giganet with VIA	30	calls/ms	30	µs
RPC	3	calls/ms	1	ms
Airline reservation transactions	10000	trans/s	1	sec
Published papers	20	papers/yr	2	years

Specs for performance

How can we put performance into our specs? In other words, how can we specify the amount of real time or other resources that an operation consumes? For resources like disk space that are controlled by the system, it's quite easy. Add a variable `spaceInUse` that records the amount of disk space in use, and to specify that an operation consumes no more than `max` space, write

```
<< VAR used: Space | used <= max => spaceInUse := spaceInUse + used >>
```


This is usually what you want, rather than saying exactly how much space is consumed, which would restrict the code too much.

Doing the same thing for real time is a bit trickier, since we don't usually think of the advance of real time as being under the control of the system. The spec, however, has to put a limit on how much time can pass before an operation is complete. Suppose we have a procedure `P`. We can specify `TimedP` that takes no more than `maxPLatency` to complete as follows. The variable `now` records the current time, and `deadlines` records a set of latest completion times for operations in progress. The thread `Clock` advances `now`, but not past a deadline. An operation like `TimedP` sets a deadline before it starts to run and clears it when it is done.

```
VAR  now       : Time
     deadlines: SET Time

THREAD Clock() = DO now < deadlines.min => now + := 1 [] SKIP OD

PROC TimedP() = VAR t : Time
     << now < t /\ t < now + maxPLatency /\ ~ t IN deadlines =>
          deadlines := deadlines + {t} >>;
     P();
     << deadlines := deadlines - {t}; RET >>
```

This may seem like an odd way of doing things, but it does allow exactly the sequences of transitions that we want. The alternative is to construct `P` so that it completes within `maxPLatency`, but there's no straightforward way to do this.

Often we would like to write a probabilistic performance spec; for example, service time is drawn from a normal distribution with given mean and variance. There's no way to do this directly in Spec, because the underlying model of non-deterministic state machines has no notion of probability. What we can do is to keep track of actual service times and declare a failure if they get too far from the desired form. Then you can interpret the spec to say: either the observed performance is a reasonably likely consequence of the desired distribution, or the system is malfunctioning.

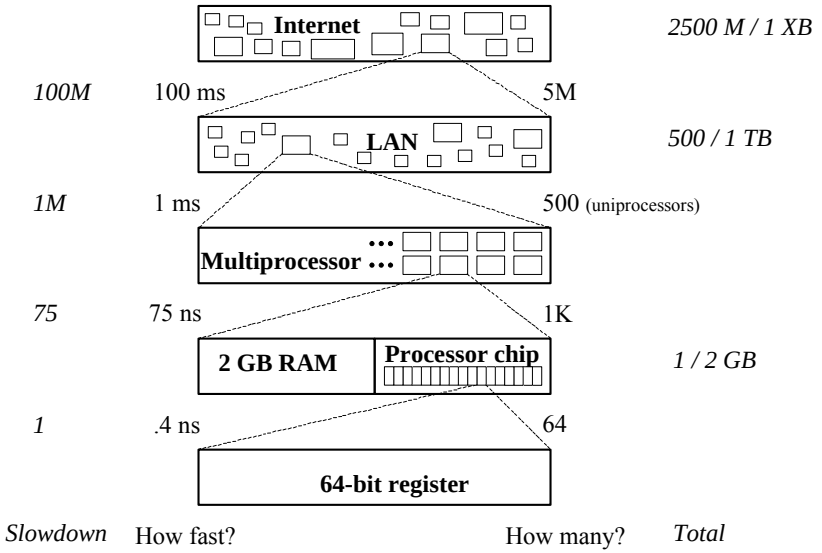
How to get performance: Methodology

First you have to choose the right scale for looking at the system. Then you have to model or analyze the system, breaking it down into a few parts that add up to the whole, and measure the performance of the parts.

Choosing the scale

The first step in understanding the performance of a system is to find the right scale on which to analyze it. The figure shows the scales from the processor clock to an Internet access; there is a range of at least 50 million in speed and 50 million in quantity. Usually there is a scale that is the right one for understanding what's going on. For the performance of an inner loop it might be the system clock, for a simple transaction system the number of disk references, and for a Web browser the number of IP packets.

In practice, systems are not deterministic. Even if there isn't inherent non-determinism caused by unsynchronized clocks, the system is usually too complex to analyze in complete detail. The way to simplify it is to approximate. First find the right scale and the right primitives to count, ignoring all the fine detail. Then find the critical parameters that govern performance at that scale: number of RPC's per transaction, cache miss rate, clock ticks per instruction, or whatever. In this



Scales of interconnection. Relative speed and size are in italics.

way you should be able to find a simple formula that comes within 20% of the observed performance, and usually this is plenty good enough.

For example, in the 1994 election DEC ran a Web server that provided data on the California election. It got about 80k hits/hour, or 20/sec, and it ran on a 200 MIPS machine. The data was probably all in memory, so there were no disk references. A hit typically returns about 2 KB of data. So the cost was about 10M instructions/hit, or 5K instructions/byte returned. Clearly this was not an optimized system.

By comparison, a simple debit-credit transaction (the TPC-A benchmark) when carefully coded does slightly more than two disk i/o's per transaction (these are to read and write per-account data that won't fit in memory). If carefully coded it takes about 100K instructions. So on a 2000 MIPS machine it will consume 50 μs of compute time. Since two disk i/o's is 20 ms, it takes 400 disks to keep up with this CPU for this application. Since this is not too reasonable, engineers have responded by coding transactions less carefully, taking advantage of the fact that instructions are so cheap.

As a third example, consider sorting 10 million 64 bit numbers; the numbers start on disk and must end up there, but you have room for the whole 80 MB in memory. So there's 160 MB of disk transfer plus the in-memory sort time, which is $n \log n$ comparisons and about half that many swaps. A single comparison and half swap might take 10 instructions with a good code for Quicksort, so this is a total of 10 * 10 M * 24 = 2.4 G instructions. Suppose the disk system can transfer 80 MB/sec and the processor runs at 200 MIPS. Then the total time is 2 sec for the disk plus 1.2 sec for the computing, or 3.2 sec, less any overlap you can get between the two phases.

With considerable care this performance can be achieved. On a parallel machine you can do perhaps 30 times better.[4]

Here are some examples of parameters that might determine the performance of a system to first order: cache hit rate, fragmentation, block size, message overhead, message latency, peak message bandwidth, working set size, ratio of disk reference time to message time.

Modeling

Once you have chosen the right scale, you have to break down the work at that scale into its component parts. The reason this is useful is the following principle:

> If a task *x* has parts *a* and *b*, the cost of *x* is the cost of *a* plus the cost of *b*, plus a system effect (caused by contention for resources) which is usually small.

Most people who have been to school in the last 20 years seem not to believe this. They think the 'system effect' is so large that knowing the cost of *a* and *b* doesn't help at all in understanding the cost of *x*. But they are wrong. Your goal should be to break down the work into a small number of parts, between two and ten. Adding up the cost of the parts should give a result within 10% of the measured cost for the whole.

If it doesn't then either you got the parts wrong (very likely), or there actually *is* an important system effect. This is not common, but it does happen. Such effects are always caused by *contention* for resources, but this takes two rather different forms:

- *Thrashing* in a cache, because the sum of the working sets of the parts exceeds the size of the cache. The important parameter is the cache miss rate. If this is large, then the cache miss time and the working set are the things to look at. For example, SQL server on Windows NT running on a DEC Alpha 21164 in 1997 executes .25 instructions/cycle, even though the processor chip is capable of 2 instructions/cycle. The reason turns out to be that the instruction working set is much larger than the instruction cache, so that essentially every block of 4 instructions (16 bytes or one cache line) causes a cache miss, and the miss takes 64 ns, which is 16 4 ns cycles, or 4 cycles/instruction.
- *Clashing* or queuing for a resource that serves one customer at a time (unlike a cache, which can take away the resource before the customer is done). The important parameter is the queue length. It's important to realize that a resource need not be a physical object like a CPU, a memory block, a disk drive, or a printer. Any lock in the system is a resource on which queuing can occur. Typically the physical resources are instrumented so that it's fairly easy to find the contention, but this is often not true for locks. In the Alta Vista web search engine, for example, CPU and disk utilization were fairly low but the system was saturated. It turned out that queries were acquiring a lock and then page faulting; during the page fault time lots of other queries would pile up waiting for the lock and unable to make progress.

In the section on techniques we discuss how to analyze both of these situations.

[4] Andrea Arpaci-Dusseau et al., High-performance sorting on networks of workstations. SigMod 97, Tucson, Arizona, May, 1999, http://now.cs.berkeley.edu/NowSort/nowSort.ps .

Measuring

The basic strategy for measuring is to count the number of times things happen and observe how long they take. This can be done by sampling (what most profiling tools do) or by logging significant events such as procedure entries and exits. Once you have collected the data, you can use statistics or graphs to present it, or you can formulate a model of how it should be (for example, time in this procedure is a linear function of the first parameter) and look for disagreements between the model and reality.[5] The latter technique is especially valuable for continuous monitoring of a running system. Without it, when a system starts performing badly in service it's very difficult to find out why.

Measurement is usually not useful without a model, because you don't know what to do with the data. Sometimes an appropriate model just jumps out at you when you look at raw profile data, but usually you have to think about it and try a few things. This is just like any branch of science: without a theory you can't make sense of the data.

How to get performance: Techniques

There are three main ways to make your program run faster: use a better algorithm, find a common case that can be made to run fast, or use concurrency to work on several things at once.

Algorithms

There are two interesting things about an algorithm: the 'complexity' and the 'constant factor'. An algorithm that works on n inputs can take roughly k (constant) time, or $k \log n$ (logarithmic), or $k\,n$ (linear), or $k\,n^2$ (quadratic), or $k\,2^n$ (exponential). The k is the constant factor, and the function of n is the complexity. Usually these are 'asymptotic' results, which means that their percentage error gets smaller as n gets bigger. Often a mathematical analysis gives a worst-case complexity; if what you care about is the average case, beware. Sometimes a 'randomized' algorithm that flips coins internally can make the average case overwhelmingly likely.

For practical purposes the difference between $k \log n$ time and constant time is not too important, since the range over which n varies is likely to be 10 to 1M, so that $\log n$ varies only from 3 to 20. This factor of 6 may be much less than the change in k when you change algorithms. Similarly, the difference between $k\,n$ and $k\,n \log n$ is usually not important. But the differences between constant and linear, between linear and quadratic, and between quadratic and exponential are very important. To sort a million numbers, for example, a quadratic insertion sort takes a trillion operations, while the $n \log n$ Quicksort takes only 20 million in the average case (unfortunately the worst case for Quicksort is also quadratic). On the other hand, if n is only 100, then the difference among the various complexities (except exponential) may be less important than the values of k.

Another striking example of the value of a better algorithm is 'multi-grid' methods for solving the n-body problem: lots of particles (atoms, molecules or asteroids) interacting according to some force law (electrostatics or gravity). By aggregating distant particles into a single virtual particle, these methods reduce the complexity from n^2 to $n \log n$, so that it is feasible to solve systems with millions of particles. This makes it practical to compute the behavior of complex chemical reactions, of currents flowing in an integrated circuit package, or of the solar system.

[5] See Perl and Weihl, Performance assertion checking. *Proc. 14th ACM Symposium on Operating Systems Principles*, Dec. 1993, pp 134-145.

Fast path

If you can find a common case, you can try to do it fast. Here are some examples.

Caching is the most important: memory, disk (virtual memory, database buffer pool), web cache, memo functions (also called 'dynamic programming'), ...

Receiving a message that is an expected ack or the next message in sequence.

Acquiring a lock when no one else holds it.

Normal arithmetic vs. overflow.

Inserting a node in a tree at a leaf, vs. splitting a node or rebalancing the tree.

Here is the basic analysis for a fast path.

1 = fast time, $1 \ll 1 + s$ = slow time, m = miss rate = probability of taking the slow path.

```
t = time = 1 + m * s
```

There are two ways to look at it:

The slowdown from the fast case (time 1). If $m = 1/s$ then $t = 2$, a 2 x slowdown.

The speedup from the slow case (time s). If $m = 50\%$ then $t = s/2 + 1$, nearly a 2 x speedup,

You can see that it makes a big difference. For $s = 100$, a miss rate of 1% yields a 2 x slowdown, but a miss rate of 50% yields a 2 x speedup.

The analysis of fast paths is most highly developed in the study of computer architecture.[6]

Batching has the same structure:

1 = unit cost, s = startup (per-batch) cost, b = batch size.

```
t = time = (b + s) / b = 1 + s/b
```

So b is like $1/m$. Amdahl's law for concurrency (discussed below) also has the same structure.

Concurrency with lots of small jobs

Usually concurrency is used to increase bandwidth. It is easiest when there are lots and lots 'independent' requests, such as web queries or airline reservation transactions. Some examples: customers buying books at Amazon, web searches on Google, or DNS name lookups. In this kind of problem there is no trouble getting enough concurrent work to do, and the challenge is getting it done efficiently. The main obstacle is bottlenecks.

Getting more and more concurrency to work is called *scaling* a system. Scaling is increasingly important because the internet allows demand for a popular service to grow almost without bound, and because commodity components are so much cheaper per unit of raw performance than any other kind of component. For a system to scale:

- It must not have any algorithms with complexity worse than $\log n$, where n is the number of components.

[6] Hennessy and Patterson, *Computer Architecture: A Quantitative Approach*, 2nd ed., Morgan Kaufmann, 1995. The second edition has a great deal of new material.

- It must not have bottlenecks.

Both of these are easy when there is no shared data, and get harder as you increase the amount of shared data, the rate at which it changes, and the requirement for consistency.

For example, in the Domain Name System (DNS) that maps a name like `lcs.mit.edu` into an IP address, the root is potentially a bottleneck, since in principle every lookup must start at the root. If the root were a single machine, DNS would be in big trouble. Two techniques relax this bottleneck in the real DNS:

> Caching the result of lookups in the root, on the theory that the IP address for `mit.edu` changes very seldom. The price, of course, is that when it does change there is a delay before everyone notices.
>
> Many copies of the root that are loosely consistent, on the theory that when a name is added or changed, it's not essential for all the copies to find out about it atomically.

Returning to concurrency for bandwidth, there can be multiple identical resources or several distinct resources. In the former case the main issue is load balancing (there is also the cost of switching a task to a particular resource). The most common example is multiple disks. If the load if perfectly balanced, the i/o rate from n disks is n times the rate from one disk. The debit-credit example above showed how this can be important. Getting the load perfectly balanced is hard; in practice it usually requires measuring the system and moving work among the resources. It's best to do this automatically, since the human cost of doing it manually is likely to dwarf the savings in hardware.

When the resources are distinct, we have a 'queuing network' in which jobs 'visit' different resources in some sequence, or in various sequences with different probabilities. Things get complicated very quickly, but the most important case is quite simple. Suppose there is a single resource and tasks arrive independently of each other ('Poisson arrivals'). If the resource can handle a single request in a service time s, and its utilization (the fraction of time it is busy) is u, then the average request gets handled in a response time

```
r = s/(1 - u)
```

The reason is that a new request needs to get s amount of service before it's done, but the resource is only free for 1 - u of the time. For example, if u = .9, only 10% of the time is free, so it takes 10 seconds of real time to accumulate 1 second of free time.

Look at the slowdowns for different utilizations.

> 2 x at 50%
> 10 x at 90%
> Infinite at 100% ('saturation')

Note that this rule applies only for Poisson (memoryless or 'random' arrivals). At the opposite extreme, if you have periodic arrivals and the period is synchronized with the service time, then you can do pipelining, drop each request into a service slot that arrives soon after the request, and get $r = s$ with u = 1. One name for this is "systolic computing".

A high u has two costs: increased r, as we saw above, and increased sensitivity to changing load. Doubling the load when u = .2 only slows things down by 30%; doubling from u = .8 is a catastrophe. High u is OK if you can tolerate increased r and you know the load. The latter could be because of predictability, for example, a perfectly scheduled pipeline. It could also be because of aggregation and statistics: there are enough random requests that the total load varies very little.

Unfortunately, many loads are "bursty", which means that requests are more likely to follow other requests; this makes aggregation less effective.

When there are multiple requests, usually one is the bottleneck, the most heavily loaded component, and you only have to look at that one (of course, if you make it better then something else might become the bottleneck).

Servers with finite load

Many papers on queuing theory analyze a different situation, in which there is a fixed number of customers that alternate between thinking (for time z) and waiting for service (for the response time z). Suppose the system in steady state (also called 'equilibrium' or 'flow balance'), that is, the number of customers that demand service equals the number served, so that customers don't pile up in the server or drain out of it. You can find out a lot about the system by just counting the number of customers that pass by various points in the system

A customer is in the server if it has entered the server (requested service) and not yet come out (received all its service). If there are n customers in the server on the average and the throughput (customers served per second) is x, then the average time to serve a customer (the response time) must be $r = n/x$. This is "Little's law", usually written $n = rx$. It is obvious if the customers come out in the same order they come in, but true in any case. Here n is called the "queue length", though it includes the time the server is actually working as well.

If there are N customers altogether and each one is in a loop, thinking for z seconds before it enters the server, and the throughput is x as before, then we can use the same argument to compute the total time around the loop $r + z = N/x$. Solving for r we get $r = N/x - z$. This formula doesn't say anything about the service time s or the utilization u, but we also know that the throughput $x = u/s$ ($1/s$ degraded by the utilization). Plugging this into the equation for r we get $r = Ns/u - z$, which is quite different from the equation $r = s/(1 - u)$ that we had for the case of uniform arrivals. The reason for the difference is that the population is finite and hence the maximum number of customers that can be in the server is N.

Concurrency in a single job

In using concurrency on a single job, the goal is to reduce latency—the time to get the job done. This requires a parallel algorithm, and runs into Amdahl's law, which is another kind of fast path analysis. In this case the fast path is the part of the program that can run in parallel, and the slow path is the part that runs serially. The conclusion is the same: if you have 100 processors, then your program can run 100 times faster if it all runs in parallel, but if 1% of it runs serially then it can only run 50 times faster, and if half runs serially then it can only run twice as fast. Usually we take the slowdown view, because the ideal is that we are paying for all the processors and so every one should be fully utilized. Then a 99% parallel / 1% serial program, which achieves a speedup of 50, is only half as fast as our ideal. You can see that it will be difficult to make efficient use of 100 processors on a single job.

Another way of looking at concurrency in a single job is the following law (actually a form of Little's law, discussed above from a different point of view):

$$concurrency = latency \times bandwidth$$

As with Ohm's law, the way you look at this equation depends on what you think are the independent variables. In a CPU/memory system, for example, the latency of a cache miss is fixed at

about 100 ns. Suppose the CPU has a floating-point unit that can do 3 multiply-add operations per ns (typical for 2006). In a large job each such operation will require one operand from main memory because all the data won't fit into the cache; multiplying two large matrices is a simple example. So the required memory bandwidth to keep the floating point unit busy is 300 reads/100 ns. With latency and bandwidth fixed, the *required* concurrency is 300.

On the other hand, the *available* concurrency is determined by the program and its execution engine. For this example, you must have 300 outstanding memory reads (nearly) all the time. A read is outstanding if it has been issued by the CPU, but the data has not yet been consumed. This could be accomplished by having 300 threads, each with one outstanding read, or 30 threads each with 10 reads, or 1 thread with 300 reads.

How can you have a read outstanding? In a modern CPU with out-of-order execution and register renaming, this means that the FETCH operation has been issued and a register assigned to hold the result. Before the result returns, the CPU can keep going, issue an ADD that uses the memory result, assign the result register for the ADD, and continue issuing instructions. If there's a branch on the result, however, the CPU must guess the branch result, since proceeding down both paths quickly becomes too expensive; the polite name for this is 'branch prediction'. It can continue down the predicted path 'speculatively', which means that it has to back up if the guess turns out to be wrong. On typical jobs of this kind prediction errors are < 1%, so speculative execution can continue for quite a while. More registers are needed, however, to hold the data needed for backing up. Eventually either the CPU runs out of registers, or the chances of a bad prediction are large enough that it doesn't pay to keep speculating.

You can see that it's hard to keep all these balls in the air long enough to get 300 reads outstanding nearly all the time, and no existing CPU does so. Some CPUs get hundreds of outstanding reads by using 'vector' instructions that call for many reads or adds in a single instruction, for example, "read 50 values from addresses *a*, *a*+100, *a*+200, ... into vector register 7" or "add vector register 7 to vector register 9 and put the result in vector register 13". Such instructions are much less flexible than scalar reads, but they can use many fewer CPU resources to make a read outstanding. Somewhat more flexible operations like "read from the addresses in vector register 7 and put the results into vector register 9" are possible; they help with sparse matrices.

Multiple threads reduce the coupling among outstanding operations. In fact, with 300 threads, each one would need only one outstanding operation, so there would be no need for speculative execution and backup. Scheduling must be done by the hardware, however, since you have to switch threads after every memory operation. Keeping so many threads running requires lots of hardware resources. In fact, it requires many of the same hardware resources required for a single thread with lots of outstanding operations. High-volume CPUs currently can run 2 threads at a time, so we are some distance from the goal.

This formulation of Little's law is useful for understanding not just CPU/memory systems, but any system in which you are trying to get high bandwidth with a fixed latency. It tells you how much concurrency you need. Then you must ask whether there's a source for that much concurrency, and whether there are enough resources to maintain the internal state that it requires. In the 'embarrassingly parallel' applications of the previous section there are plenty of requests, and a job that's waiting just consumes some memory, which is usually in ample supply. This means that you only have to worry about the costs of the data structures for scheduling.

Summary

Here are the most important points about performance.

- Moore's law: The performance of computer systems at constant cost doubles every 18 months, or increases by ten times every five years.
- To understand what a system is doing, first do a back-of-the-envelope calculation that takes account only of the most important one or two things, and then measure the system. The hard part is figuring out what the most important things are.
- If a task *x* has parts *a* and *b*, the cost of *x* is the cost of *a* plus the cost of *b*, plus a system effect (caused by contention for resources) which is usually small.
- For a system to scale, its algorithms must have complexity no worse than log *n*, and it must have no bottlenecks. More shared data makes this harder, unless it's read-only.
- The time for a task which has a fast path and a slow path is $1 + m * s$, where the fast path takes time 1, the slow path takes time $1 + s$, and the probability of taking the slow path is m (the miss rate). This formula works for batching as well, where the batch size is $1/m$.
- If a shared resource has service time s to serve one request and utilization u, and requests arrive independently of each other, then the response time is $s/(1 - u)$. It tends to infinity as u approaches 1.
- concurrency = latency × bandwidth

.

11. Paper: Performance of Firefly RPC

Michael D. Schroeder and Michael Burrows[1]

Abstract

In this paper, we report on the performance of the remote procedure call code for the Firefly multiprocessor and analyze the code to account precisely for all measured latency. From the analysis and measurements, we estimate how much faster RPC could be if certain improvements were made.

[1] Authors' address: Digital Equipment Corporation, Systems Research Center, 130 Lytton Avenue, Palo Alto, California 94301. A slightly different version of this paper appeared in *ACM Transactions on Computer Systems*, **8**, 1, February 1990.

.

12. Naming

Any problem in computing can be solved by another level of indirection.
David Wheeler

Introduction

This handout is about orderly ways of naming complicated collections of objects in a computer system. A basic technique for understanding a big system is to describe it as a collection of simple parts. Being able to name these parts is a necessary aspect of such a description, and often the most important aspect.

The basic idea can be expressed in two ways that are more or less equivalent:

> Identify values by variable length names called *path names* that are sequences of simple names that are strings. Think of all the names with the same prefix (for instance, `/udir/lampson` and `/udir/lynch`) as being grouped together. This grouping induces a tree structure on the names. Non-leaf nodes in the tree are *directories*.
>
> Make a tree of nodes with simple names on the arcs. The leaf nodes are values and the internal nodes are *directories*. A node is named by a path through the tree from the root; such a name is called a *path name*.

Thus `/udir/lampson/pocs/handouts/12` is a path name for a value (perhaps the text of this handout), and `/udir/lampson/pocs/handouts` is a path name for a directory (other words for directory are folder, context, closure, environment, binding, and dictionary). The collection of all the path names that make sense in some situation is called a *name space*. Viewing a name space as a tree gives us the standard terminology of parents, children, ancestors, and descendants.

Using path names to name values (or objects, if you prefer) is often called ‘hierarchical naming’ or ‘tree-structured naming’. There are a lot of other names for it that are used in special situations: mounting, search paths, multiplexing, device addressing, network references. An important reason for studying naming in general is that you don’t have to start from scratch in understanding all those other things.

Path names are good because:

- The name space can grow indefinitely, and the growth can be managed in a decentralized way. That is, the authority to create names in one part of the space can be delegated, and thereafter there is no need for synchronization. Names that start `/udir/lampson` are independent of names that start `/udir/rinard`.
- Many kinds of data can be encapsulated under this interface, with a common set of operations. Arbitrary operations can be encoded as reads and writes of suitably chosen names.

As we have seen, a path name is a sequence of simple names. We use the types `N = String` for a simple name and `PN = SEQ N` for a path name. It is often convenient to write a path name as a string. The syntax of these strings is not important; it is just a convention for encoding the path names. Here are some examples:

`/udir/lampson/pocs/handouts/12`	Unix path name
`lampson@comcast.net`	Internet mail address. The path name is `{"net"; "comcast"; "lampson"}`[1]
`16.23.5.193`	IP network address (fixed length)

We will normally write path names as Unix file names, rather than as the sequence constructors that would be correct Spec. Thus `a/b/c/1026` instead of `PN{"a";"b";"c";"1026"}`.

People often try to distinguish a name (what something is) from an address (where it is) or a route (how to find it). This is a matter of levels of abstraction and must not be taken as absolute. At a given level of abstraction we tend to identify objects at that level by names, the lower-level objects that code them by addresses, and paths at lower levels by routes. Examples:

```
microsoft.com -> 207.46.130.149 -> {router output port, LAN address}
a/b/c/1026 -> INode/1026 -> DA/2 -> {cylinder, head, sector, byte 2}
```

Sometimes people talk about "descriptive names", which are queries in a database. We will see that these are readily encompassed within the framework of path names. That is a formal relationship, however. There is an important practical difference between a *designator* for a single entity, such as `lampson@comcast.net`, and a *description* or query such as "everyone at MIT's CSAIL whose research involves parallel computing". The difference is illuminated by the comparison between the name `eecsfaculty@eecs.mit.edu` and the query "the faculty members in MIT's EECS department". The name is probably maintained with some care; it's anyone's guess how reliable the answer to the query is. When using a name, it is wise to consider whether it is a designator or a description.

This is not to say that descriptions or queries are bad. On the contrary, they are very valuable, as any one knows who has ever used a web search engine. However, they usually work well only when a person examines the results carefully.

In the remainder of this handout we examine the specs for the two ways of describing a name space that we introduced earlier: as a memory addressed by path names, and as a tree (or more generally a graph) of directories. The two ways are closely related, but they give rise to somewhat different specs. Then we study the recursive structure of name spaces and various ways of inducing a name space on a collection of values. This leads to a more abstract analysis of how the spec for a name space can vary, depending on the properties of the underlying values. We conclude our general treatment by examining how to name a name space. Finally, we give a large number of examples of name spaces; you might want to look at these first to get some more context.

Name space as memory

We can view a name space as an example of the memory abstraction we studied earlier. Recall that a memory is a partial map `M = A -> V`. Here we take `A = PN` and replace `M` with `D` (for directory). This kind of memory differs from the byte-addressable physical memory of a computer in several ways[2]:

- The map is partial.
- The current value of the domain (that is, which names are defined) is interesting.
- The domain is changing.
- `PN`'s with the same prefix are related (though not as much as in the second view of name spaces).

Here are some examples of name spaces that can naturally be viewed as memories:

> The Simple Network Management Protocol (SNMP) is used to manage components of the Internet. It uses path names (rooted in IP addresses) to name values, and the basic operations are to read and write a single named value.
>
> Several file systems use a single large table to map the path name of a file to the extents that represent it.

```
MODULE MemNames0[V] EXPORT Read, Write, Remove, Enum, Next, Rename =

TYPE N            =  String                                  % Name
     PN           =  SEQ N WITH {"<<=":=PNLE}                % Path Name
     D            =  PN -> V                                 % Directory

VAR  d            := D{}                                     % the state

FUNC PNLE(pn1, pn2) -> Bool = pn1.LexLE(pn2, N."<=")   % pn1 <<= pn2
```

Note that `PN` has a `"<<="` operator defined that overrides the standard one for sequences; its meaning is lexical comparison, defined four lines later.

Here are the familiar `Read` and `Write` procedures; `Read` raises `error` if `d` is undefined at `pn`, for consistency with later specs. In this basic spec none of the other procedures raises `error`; this innocence will not persist when things get more complicated. It's common to also have a `Remove` procedure for making a `PN` undefined; note that unlike a file system, this `Remove` does not erase the values of longer names that start with `PN`. This is because, unlike a file system, this spec does not ensure that every prefix of a defined `PN` is defined.

```
FUNC Read(pn) -> V RAISES {error} = RET d(pn) [*] RAISE error

APROC Write(pn, v) = << d := d{pn -> v} >>

APROC Remove(pn)   = << d := d{pn ->  } >>
```

The body of `Write` is usually written `d(pn) := v`.

It's important that the map is partial, and that the domain changes. This means that we need operations to find out what the domain is. Simply returning the entire domain is not practical, since it may be too big, and usually only part of it is of interest. There are two schools of thought about

[1] Actually this is an oversimplification, since `lampson.comcast.net` is a perfectly good DNS name, and both it and `lampson@comcast.net` might be defined. We need some convention for distinguishing them. For example, we could say that the path name for `lampson@comcast.net` is `{"net"; "comcast"; "@"; "lampson"}`.

[2] It differs much less from the virtual memory, in which the map may be partial and the domain may change as new virtual memory is assigned or files are mapped. Actually these things can happen to physical memory as well, especially in the part of it implemented by I/O devices.

what form these operations should take, represented by the functions `Enum` and `Next`; only one of these is needed.

> `Enum` returns all the simple names that can lead to a value starting from `pn`; another way of saying this is that it returns all the names bound in the directory named `pn`. By recursively applying `Enum` to `pn + n` for each simple name `n` that `Enum` returns, you can explore the entire tree.
>
> On the other hand, if you keep feeding `Next` its own output, starting with `{}`, it walks the tree of defined names depth-first, returning in turn each `PN` that is bound to a `V`. It finishes with `{}`.

Note that what `Next` does is not the same as returning the results of `Enum` one at a time, since `Next` explores the entire tree, not just one directory. Thus `Enum` takes the organization of the name space into directories more seriously than does `Next`.

```
FUNC Enum(pn) -> SET N = RET {pn1 | d!(pn + pn1) || pn1.head}

FUNC Next(pn) -> PN    = VAR later := {pn' | d!pn' /\ pn <= pn'} |
     RET later.fmin(PN."<<=") [*] RET {}              % {} if later is empty
```

You might be tempted to write `{n | d!(pn + {n})}` for the result of `Enum`, but as we saw earlier, there's no guarantee in this spec that every prefix of a defined path name is defined.

A separate issue is arranging to get a reasonable number of results from one of these procedures. If the directory is large, `Enum` as defined here may return an inconveniently large set, and we may have to call `Next` inconveniently many times. In real life we would make either routine return a sequence of `N`'s or `PN`'s, usually called a 'buffer'. This is a standard use of batching to reduce the overhead of invoking an operation, without allowing the batches to get too large. We won't add this complication to our specs.

Finally, there is a `Rename` procedure that takes directories quite seriously. It reflects the idea that all the names which start the same way are related, by changing all the names that start with `from` so that they start with `to`. Because directories are not very real in the representation, this procedure has to do a lot of work. It erases everything that starts with either argument, and then copies everything in the original `d` that starts with `from` to the corresponding path name that starts with `to`. Read `x <= y` as "`x` is a prefix of `y`".

```
APROC Rename(from: PN, to: PN) RAISES {error} = << VAR d0 := d |
     IF   from <= to => RAISE error                    % can't rename to a descendant
     [*]  DO VAR pn :IN d.dom | (to <= pn \/ from <= pn) => d := d{pn -> } OD;
          DO VAR pn | d(to + pn ) # d0(from + pn) => d(to + pn):=d0(from + pn) OD
     FI >>

END MemNames0
```

Here is a different version of `Rename` that makes explicit the relation between the initial state `d` and the final state `d'`. Read `x >= y` as "`x` is a suffix of `y`".

```
APROC Rename(from: PN, to: PN) RAISES {error} = <<
     IF VAR d' |
          (ALL x: PN, y: PN |
               (    x >= from                  => ~ d'!x
                [*] x = to + y /\ d!(from + y) => d'(x) = d(from + y)
                [*] ~ x >= to /\ d!x           => d'(x) = d(x)
                [*] ~ d'!x )
          => d := d'
     [*]  RAISE error FI >>
```


There is often a rule that a name can be bound to a directory or to a value, but not both. For this we need a slightly different spec that marks a name as bound to a directory by giving it the special value `isD`, with a separate procedure for making an empty directory. To enforce the new rule every routine can now raise `error`, and `Remove` erases the whole sub-tree. As usual, boxes mark the changes from `MemNames0`.

```
MODULE MemNames[V] EXPORT Read, Write, MakeD, Remove, Enum, Rename =

TYPE Dir          =  ENUM[isDir]
     D            =  PN -> (V + Dir) SUCHTHAT this({}) IS Dir)      % root a Dir

VAR  d            := D{{} -> isDir}

% INVARIANT (ALL pn, pn' | d!pn' /\ pn' > pn => d(pn) = isDir

FUNC Read(pn) -> V RAISES {error} = d(pn) IS V => RET d(pn) [*] RAISE error

FUNC Enum(pn) -> SET N RAISES {error} =
      d(pn) IS Dir => RET {n | d!(pn + {n})}   [*] RAISE error

APROC Write(pn, v) RAISES {error}  = << Set(pn, v) >>
APROC MakeDir(pn)  RAISES {error}  = << Set(pn, isDir) >>

APROC Remove(pn) =                                  % Erase everything with pn prefix.
     << DO VAR pn' :IN d.dom | (pn <= pn') => d := d{pn' -> } OD >>

APROC Rename(from: PN, to: PN) RAISES {error} = << VAR d0 := d |
     IF   from <= to => RAISE error                 % can't rename to a descendant
     [*]  DO VAR pn :IN d.dom | (to <= pn \/ from <= pn) => d := d{pn -> } OD;
          DO VAR pn | d(to + pn ) # d0(from + pn) =>
               d(to + pn) := d0(from + pn) OD
     FI >>

APROC Set(pn, y: (V + D) RAISES {error} =
     << pn # {} /\ d(pn.reml) IS D => d(pn) := y [*] RAISE error >>

END MemNames
```

There are more dnj comments for the rest of this chapter.

A file system usually forbids overwriting a file with a directory (for no obvious reason) or overwriting a non-empty directory with anything (because a directory is precious and should not be clobbered wantonly), but these rules are rather arbitrary, and we omit them here.

Exercise: write a version of `Rename` that makes explicit the relation between the initial state `d` and the final state `d'`, in the style of the second `Rename` of `MemNames0`.

The `MemNames` spec is basically the same as the simple `Memory` spec. Complications arise because the domain can change, and because of the distinction between directories and values. The specs in the next section take this distinction much more seriously.

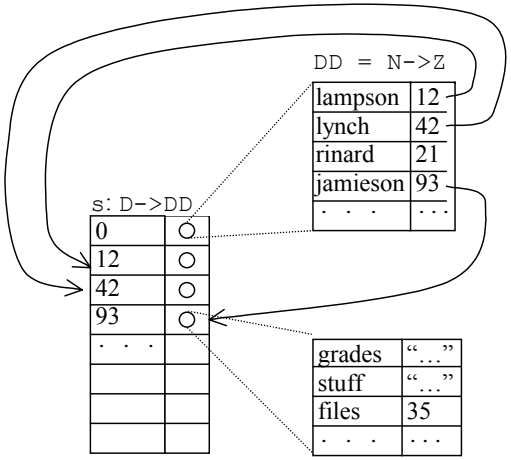
Name space as graph of directory objects

The `MemNames` specs are reasonably simple, but they are clumsy for operations on directories such as `Rename`. More fundamentally, they don't handle aliasing, where the same object has

more than one name. The other (and more usual) way to look at a hierarchical name space is to think of each directory as a function that maps a simple name (not a path name) to a value or another directory, rather than thinking of the entire tree as a single `PN -> V` map. This tree (or general graph) structure maps a `PN` by mapping each `N` in turn, traversing a path through the graph of directories; hence the term 'path name'. We continue to use the type `D` for a directory.

Our eventual goal is a spec for a name space as graph that is 'object-oriented' in the sense that you can supply different code for each directory in the name space. We will begin, however, with a simpler spec that is equivalent to `MemNames`, evolve this to a more general spec that allows aliases, and finally add the object orientation.

The obvious thing to do is to make a `D` be a function `N -> Z`, where `Z = (D + V)` as before, and have a state variable `d` which is the root of the tree. Unfortunately this completely functional structure doesn't work smoothly, because there's no way to change the value of `a/b/c/d` without changing the value of `a/b/c` so that it contains the new value of `a/b/c/d`, and similarly for `a/b` and `a` as well.[3]



We solve this problem in the usual way with another level of indirection, so that the value of a directory name is not a `N -> Z` but some kind of reference or pointer to a `N -> Z`, as shown in the figure. This reference is an 'internal name' for a directory. We use the name `DD` for the actual function `N -> Z` and introduce a state variable `s` that holds all the `DD` values; its type is `D->DD`. A `D` is just the internal name of a directory, that is, an index into `s`. We take `D = Int` for simplicity, but any type with enough values would do; in Unix `D = INo`. You may find it helpful to think of `D` as a pointer and `s` as a memory, or of `D` as an inode number and `s` as the inodes. Later sections explore the meaning of a `D` in more detail, and in particular the meaning of `root`.

Once we have introduced this extra indirection the name space does not have to be a tree, since two `PN`'s can have the same `D` value and hence refer to the same directory. In a Unix file system,

[3] The method of explicitly changing all the functions up to the root has some advantages. In particular, we can make several changes to different parts of the name space appear atomically by waiting to rewrite the root until all the changes are made. It is not very practical for a file system, though at least one has been built this way: H.E. Sturgis, *A Post-Mortem for a Time-sharing System*, PhD thesis, University of California, Berkeley, and Report CSL 74-1, Xerox Research Center, Palo Alto, Jan 1974. It has also been used in database systems to atomically change the entire database state; in this context it is called 'shadowing'. See Gray and Reuter, pp 728-732.

for example, every directory with the path name `pn` also has the path names `pn/.`, `pn/./.`, etc., and if `pn/a` is a subdirectory, then the parent also has the names `pn/a/..`, `pn/a/../a/..`, etc. Thus the name space is not a tree or even a DAG, but a graph with cycles, though the cycles are constrained to certain stylized forms involving '`.`' and '`..`'. This means, of course, that there are defined `PN`'s of unbounded length; in real life there is usually an arbitrary upper bound on the length of a defined `PN`.

The spec below does not expose `D`'s to the client, but deals entirely in `PN`'s. Real systems often do expose the `D` pointers, usually as some kind of capability (for instance in a file system that allows you to open a directory and obtain a file descriptor for it), but sometimes just as a naked pointer (for instance in many distributed name servers). The spec uses an internal function `Get`, defined near the end, that looks up a `PN` in a directory; `GetD` is a variation that raises `error` if it can't return a `D`.

`MODULE ObjNames0[V]` `EXPORT Read, Write, MakeD, Remove, Enum, Rename =`

```
TYPE D          =  Int                                  % just an internal name
     Z          =  (V + D)                              % the value of a name
     DD         =  N -> Z                               % a Directory

CONST root      :  D := 0
VAR  s          := (D -> DD){}{root -> DD{}}            % initially empty root

FUNC Read(pn) -> V RAISES {error} = VAR z := Get(root, pn) |
     IF z IS V => RET z [*] RAISE error FI

FUNC Enum(pn) -> SET PN RAISES {error} = RET s(GetD(root, pn)).dom
% Raises error if pn isn't a directory, like MemNames.
```

A write operation on the name `a/b/c` has to change the `d` component of the directory `a/b`; it does this through the procedure `SetPN`, which gets its hands on that directory by invoking `GetD(root, pn.reml)`.

```
APROC Write(pn, v) RAISES {error} = << SetPN(pn, v) >>
APROC MakeD(pn)  RAISES {error} = << VAR d := NewD() | SetPN(pn, d)  >>

APROC Remove(pn)   RAISES {error} =
     << VAR d := GetD(root, pn.reml) | >>

APROC Rename(from: PN, to: PN) RAISES {error} = <<
     IF   (to = {}) \/ (from <= to) => RAISE error      % can't rename to a descendant
     [*]  VAR fd := GetD(root, from.reml),              % know from, to # {}
               td := GetD(root, to  .reml) |
               s(fd)!(from.last) =>
                    s(td) := s(td)(to  .last -> s(fd)(from.last));
                    s(fd) := s(fd){from.last -> }
     [*]  RAISE error
     FI >>
```

The remaining routines are internal. The main one is `Get(d, pn)`, which returns the result of starting at `d` and following the path `pn`. `GetD` raises `error` if it doesn't get a directory. `NewD` creates a new, empty directory.

```
FUNC Get(d, pn) -> Z RAISES {error} =
% Return the value of pn looked up starting at z.
     IF   pn = {} => RET d
     [*]  VAR z :=s(d)(pn.head) | z IS D => RET Get(z, pn.tail)
     [*]  RAISE error
```

```
    FI

FUNC GetD(d, pn) -> D RAISES {error} = VAR z := Get(d, pn) |
    IF z IS D => RET z [*] RAISE error FI

APROC SetPN(pn, z) RAISES {error} =
    << VAR d := GetD(root, pn.reml) | s(d)(pn.last) := z >>

APROC NewD() -> D = << VAR d | ~ s!d => s(d) := DD{}; RET d >>

END ObjNames0
```

As we did with the second version of `MemNames0.Rename`, we can give a definition of `Get` in terms of a predicate. It says that there's a sequence `p` of directories starting at `d` and ending at the result of `Get`, such that the components of `pn` select the corresponding components of `p`; if there's no such sequence, raise `error`.

```
FUNC Child(z1, z2) -> Bool = z1 IS D /\ s!z1 /\ z2 IN s(z1).rng

FUNC Get(d, pn) -> Z RAISES {error} = <<
    IF VAR p :IN Child.paths |
        p.head = d /\ (ALL i :IN pn.dom | p(i+1) = s(p(i)(pn(i))) => RET p.last
    [*] RAISE error
    FI >>
```

`ObjNames0` is equivalent to `MemNames`. The abstraction function from `ObjNames0` to `MemNames` is

```
    MemNames.d = (\ pn | G(pn) IS V => G(pn) [*] G(pn) IS D => isD)
```

where we define a function `G` which is like `Get` on `root` except that it is undefined where `Get` raises `error`:

```
FUNC G(pn) -> Z = RET Get(root, pn) EXCEPT error => IF false => SKIP FI
```

The `EXCEPT` turns the `error` exception from `Get` into an undefined result for `G`.

Exercise: What is the abstraction function from `MemNames` to `ObjNames0`.

Objects, aliases, and atomicity

This spec makes clear the basic idea of interpreting a path name as a path through a graph of directories, but it is unrealistic in several ways:

The operations for changing the value of the `DD` functions in `s` may be very different from the `Write` and `MakeD` operations of `ObjNames0`. This happens when we impose the naming abstraction on a data structure that changes according to its own rules. SNMP is a good example; the values of names changes because of the operation of the network. Later in this handout we will explore a number of these variations.

There is often an 'alias' or 'symbolic link' mechanism which allows the value of a name `n` in context `d` to be a *link* `(d', pn)`. The meaning is that `d(n)` is a synonym for `Get(d', pn)`.

The operations are specified as atomic, but this is often too strong.

Our next spec, `ObjNames`, reflects all these considerations. It is rather complicated, but the complexity is the result of the many demands placed on it; ideas for simplifying it would be gratefully received. `ObjNames` is a fairly realistic spec for a naming system that allows for both symbolic links and extensible code for directories.

A `ObjNames.D` has `get` and `set` methods to allow for different code, though for now we don't take any advantage of this, but use the fixed code `GetFromS` and `SetInS`. In the section on object-oriented directories below, we will see how to plug in other versions of `D` with different `get` and `set` methods. The section on coherence below explains why `get` is a procedure rather than a function. These methods map undefined values to `nil` because it's tricky to program with undefined in this general setting; this means that `Z` needs `Null` as an extra case.

`Link` is another case of `Z` (the internal value of a name), and there is code in `Get` to follow links; the rules for doing this are somewhat arbitrary, but follow the Unix conventions. Because of the complications introduced by links, we usually use `GetDN` instead of `Get` to follow paths; this procedure converts a `PN` relative to `root` into a directory `d` and a name `n` in that directory. Then the external procedures read or write the value of that name.

Because `Get` is no longer atomic, it's no longer possible to define it in terms of a path through the directories that exists at a single instant. The section on atomicity below discusses this point in more detail.

`MODULE ObjNames[V]` `EXPORT ... =`

```
TYPE D          =  Int                                  % Just an internal name
                   WITH {get:=GetFromS, set:=SetInS}    % get returns nil if undefined
    Link        =  [d: (D + Null), pn]                  % d=nil for 'relative': the containing D
    Z           =  (V + D + Link + Null)                % nil means undefined
    DD          =  N -> Z

CONST root      :  D := 0
VAR  s          := (D -> DD){}{root -> DD{}}           % initially empty root

APROC GetFromS(d, n) -> Z =                             % d.get(n)
    << RET s(d)(n) [*] RET nil >>

APROC SetInS  (d, n, z)   =                             % d.set(n, z)
% If z = nil, SetInS leaves n undefined in s(d).
    << IF z # nil => s(d)(n) := z [*] s(d) := s(d){n -> } FI >>

PROC Read   (pn)     -> V     RAISES {error} = VAR z := Get(root, pn) |
    IF z IS V => RET z [*] RAISE error FI

PROC Enum   (pn)     -> SET N RAISES {error} =
% Can't just write RET GetD(root, pn).get.dom as in ObjNames0, because get isn't a function.
% The lack of atomicity is on purpose.
    VAR d := GetD(root, pn), ns: SET N := {}, z |
        DO VAR n | << z := d.get(n); ~ n IN ns /\ z # nil => ns + := {n} >> OD;
        RET ns

PROC Write  (pn, v)          RAISES {error} =                SetPN(pn,v, true )

PROC MakeD(pn)               RAISES {error} = VAR d:=NewD() | SetPN(pn,d, false)

PROC Rename(from: PN, to: PN) RAISES {error} = VAR d, n, d', n' |
    IF   (to = {}) \/ (from <= to) => RAISE error       % can't rename to a descendant
    [*]  (d, n) := GetDN(from, false); (d', n') := GetDN(to, false);
         << d.get!n => d'.set(n', d.get(n)); d.set(n, nil) >>
    [*]  RAISE error
    FI
```

This version of `Rename` imposes a different restriction on renaming to a descendant than real file systems, which usually have a notion of a distinguished parent for each directory and disallow

`ParentPN(d) <= ParentPN(d')`. They also usually require `d` and `d'` to be in the same 'file system', a notion which we don't have. Note that `Rename` does its two writes atomically, like many real file systems.

The remaining routines are internal. `Get` follows every link it sees; a link can appear at any point, not just at the end of the path. `GetDN` would be just

```
IF pn = {} => RAISE error [*] RET (GetD(root, pn.reml), pn.last) FI
```

except for the question of what to do when the value of this `(d, n)` is a link. The `followLastLink` parameter says whether to follow such a link or not. Because this can happen more than once, the body of `GetDN` needs to be a loop.

```
PROC Get(d, pn) -> Z RAISES {error} = VAR z := d |
% Return the value of pn looked up starting at d.
    DO << pn # {} => VAR n := pn.head, z' |
        IF   z IS D =>                                  % must have a value for n.
             z' := z.get(n);
             IF   z' # nil =>
                  % If there's a link, follow it. Otherwise just look up n.
                  IF   (z, pn') := FollowLink(z, n); pn := pn' + pn.tail
                  [*]  z        := z'                ; pn :=       pn.tail
                  FI
             [*]  RAISE error
             FI
        [*]  RAISE error
        FI
    >> OD; RET z

PROC GetD(d, pn) -> D RAISES {error} = VAR z := Get(d, pn) |
    IF z IS D => RET z AS D [*] RAISE error FI

PROC GetDN(pn, followLastLink: Bool) -> (D, N) RAISES {error} = VAR d := root |
% Convert pn into (d, n) such that d.get(n) is the item that pn refers to.
    DO   IF  pn = {} => RAISE error
         [*] VAR n := pn.last, z |
             d := Get(d, pn.reml);
             % If there's a link, follow it and loop. Otherwise return.
             << followLastLink => (d, pn) := FollowLink(d, n) [*] RET (d, n) >>
         FI
    OD

APROC FollowLink(d, n) -> (D, PN) = <<
% Fail if d.get(n) not Link. Use d as the context if the link lacks one.
    VAR l := d.get(n) | l IS Link => RET ((l.d IS D => l.d [*] d), l.pn) >>

PROC SetPN(pn, z, followLastLink: Bool) RAISES {error} =
    VAR d, n | (d, n) := GetDN(pn, followLastLink); d.set(n, z)

APROC NewD() -> D = << VAR d | ~ s!d => s(d) := D{}; RET d >>

END ObjNames
```

Object-oriented directories

Although `D` in `ObjNames` has `get` and `set` methods, they are the same for all `D`'s. To encompass the full range of applications of path names, we need to make a `D` into a full-fledged 'object', in which different instances can have different `get` and `set` operations (yet another level of indirection). This is the essential meaning of 'object-oriented': the type of an object is a record of routine types which defines a single interface to all objects of that type, but every object has its own values for the routines, and hence its own code.

To do this, we change the type to:

```
TYPE D          =  [get: APROC (n) -> Z, set: PROC (n, z) RAISES {error}]
    DR          =  Int                                  % what D used to be; R for reference
```

keeping the other types from `ObjNames` unchanged:

```
    Z           =  (V + D + Link + Null)                % nil means undefined
    DD          =  N -> Z
```

We also need to change the state to:

```
CONST root      := NewD()
VAR  s          := (DR -> DD){root -> DD{}}             % initially empty root
```

and to provide a new version of the `NewD` procedure for creating a new standard directory. The routines that `NewD` assigns to `get` and `set` have the same bodies as the `GetFromS` and `SetInS` routines.

A technical point: The reason for not writing `get:=s(dr)` in `NewD` below is that this would capture the value of `s(dr)` at the time `NewD` is invoked; we want the value at the time `get` is invoked, and this is what we get because of the fact that Spec functions are functions on the global state, rather than pure functions.

```
APROC NewD() -> D = << VAR dr | ~ s!dr =>
    s(dr) := DD{};
    RET D{ get := (\ n | s(dr)(n)),
           set := (PROC (n, z) = IF  z # nil => s(dr)(n) := z
                                 [*] s(dr) := s(dr){n -> } FI) }


PROC SetErr(n, z) RAISES {error} = RAISE error
% For later use as a set proc if the directory is read-only
```

We don't need to change anything else in `ObjNames`.

We will see many other examples of `get` and `set` routines. Note that it's easy to define a `D` that disallows updates, by making `set` be `SetErr`.

Views and recursive structure

In this section we examine ways of constructing name spaces, and in particular ways of building up directories out of existing directories. We already have a basic recursive scheme that makes a set of existing directories the children of a parent directory. The generalization of this idea is to define a function on some state that returns a `D`, that is, a pair of `get` and `set` procedures. There are various terms for this:

'encapsulating' the state,

'embedding' the state in a name space,

'making the state compatible' with a name space interface,

defining a 'view' on the state.

We will usually call it a view. The spec for a view defines how the result of `get` depends on the state and how `set` affects the state.

All of these terms express the same idea: make the state behave like a `D`, that is, abstract it as a pair of `get` and `set` procedures. Once packaged in this way, it can be used wherever a `D` can be used. In particular, it can be an argument to one of the recursive views that make a `D` out of other `D`'s: a parent directory, a link, or the others discussed below. It can also be the argument of tools like the Unix commands that list, search, and manipulate directories.

The read operations are much the same for all views, but updates vary a great deal. The two simplest cases are the one we have already seen, where you can set the value of a name just as you write into a memory location, and the even simpler one that disallows updates entirely; the latter is only interesting if `get` looks at global state that can change in other ways, as it does in the `Union` and `Filter` operations below. Each time we introduce a view, we will discuss the spec for updating it.

In the rest of this section we describe views that are based on directories: links, mounting, unions, and filters. The final section of the handout gives many examples of views based on other kinds of data.

Links and mounting

The idea behind links (called 'symbolic links' in Unix, 'shortcuts' in Windows, and 'aliases' in the Macintosh) is that of an alias (another level of indirection): define the value of a name in a directory by saying that it is the same as the value of some other name in some other directory. If the value is a directory, another way of saying this is that we can represent a directory `d` by the link `(d', pn')`, with `d(pn) = d'(pn')(pn)`, or more graphically `d/pn = d'/pn'/pn`. When put in this form it is usually called *mounting* the directory `d'(pn')` on `pn0`, if `pn0` is the name of `d`. In this language, `pn0` is called a 'mount point'. Another name for it is 'junction'.

We have already seen code in `ObjNames` to handle links. You might wonder why this code was needed. Why isn't our wonderful object-oriented interface enough? The reason is that people expect more from aliases than this interface can deliver: there can be an alias for a value, not only for a directory, and there are complicated rules for when the alias should be followed silently and when it should be an object in its own right that can be enumerated or changed

Links and mounting make it possible to give objects the names you want them to have, rather than the ones they got because of defects in the system or other people's bad taste. A very down-to-earth example is the problems caused by the restriction in standard Unix that a file system must fit on a single disk. This means that in an installation with 4 disks and 12 users, the name space contains `/disk1/john` and `/disk2/mary` rather than the `/udir/john` and `/udir/mary` that we want. By making `/udir/john` be a link to `/disk1/john`, and similarly for the other users, we can hide this annoyance.

Since a link is not just a `D`, we need extra interface procedures to read the value of a link (without following it automatically, as `Read` does), and to install a link. We call the install procedure `Mount` to emphasize that a mount point and a symbolic link are essentially the same thing. The `Mount` procedure is just like `Write` except for the second argument's type and the fact that it doesn't follow a final link in `pn`.

```
PROC ReadLink(pn) -> Link RAISES {error} = VAR d, n |
     (d, n) := GetDN(pn, false);
     VAR z | z := d.get(n); IF z IS Link => RET z [*] RAISE error FI

PROC Mount(pn, link) -> DD = SetPN(pn, link, false)
```


The section on roots below discusses where we might get the `D` in the `link` argument of `Mount`. In the common case of a link to someplace in the same name space, we have:

```
PROC MakeLink(pn, pn', local: Bool) =
     Mount(pn, Link{d := (local => nil [*] root), pn := pn'})
```

Updating (with `Write`, for instance) makes sense when there are links, but there are two possibilities. If every link is followed then a link never gets updated, since `GetDN` never returns a reference to a link. If a final link is not followed then it can be replaced by something else.

What is the relation between these links and what Unix calls 'hard links'? A Unix hard link is an inode number, which you can think of as a direct pointer to a file; it corresponds to a `D` in `ObjNames`. Several directory entries can have the same inode number. Another way to look at this is that the inodes are just another kind of name of the form `inodeRoot/2387754`, so that a hard link is just a link that happens to be an inode number rather than an ordinary path name. There is no provision for making the value of an inode number be a link (or indeed anything except a file), so that's the end of the line.

Unions

Since a directory is a function `N -> Z`, it is natural to combine two directories with the `"+"` overlay operator on functions[4]. If we do this repeatedly, writing `d1 + d2 + d3`, we get the effect of a 'search path' that looks at `d3` first, then `d2`, and finally `d1` (in that order because `"+"` gives preference to its second argument, unlike a search path which gives preference to its first argument). The difference is that this rule is part of the name space, while a search path must be coded separately in each program that cares. It's unclear whether an update of a union should change the first argument, change the second argument, do something more complicated, or raise an error. We take the last view for simplicity.

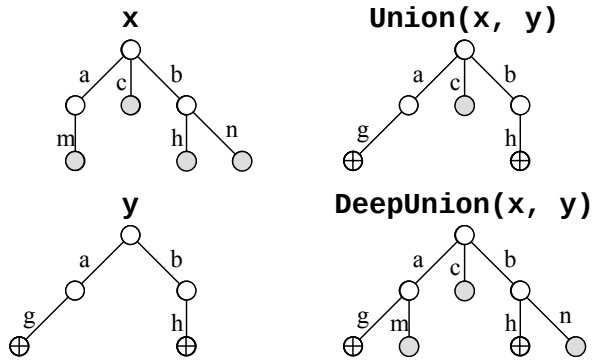
```
FUNC Union(d1, d2) -> D = RET D{get := d1.get + d2.get, set := SetErr}
```
[5]

Another kind of union combines the name spaces at every level, not just at the top level, by merging directories recursively. This is the most general way to combine two trees that have evolved independently.

```
FUNC DeepUnion(d1, d2) -> D = RET D{
     get := (\ n |
          (    d1.get(n) IS D /\ d2.get(n) IS D => DeepUnion(d1.get(n),d2.get(n))
           [*] (d1.get + d2.get)(n) )),
     set := SetErr}
```

This is a spec, of course, not efficient code.

[4] See section 9 of the Spec reference manual.
[5] This is a bit oversimplified, since `get` is an `APROC` and hence doesn't have `"+"`defined. But the idea should be clear. Plan 9 (see the examples at the end) implements unions.



Filters and queries

Given a directory `d`, we can make a smaller one by selecting some of `d`'s children. We can use any predicate for this purpose, so we get:

```
FUNC Filter(d, p: (D, N) -> Bool) -> D =
    RET D{get := ( \ n | (p(d, n) => d.get(n)) [*] nil ), set := SetErr}
```

Examples:

Pattern match in a directory: `a/b/*.ps`. The predicate is true if `n` matches `*.ps`.

Querying a table: `payroll/salary>25000/name`. The predicate is true if `Get(d, n/salary) > 25000`. See the example of viewing a table in the final section of examples.

Full text indexing: `bwl/papers/word:naming`. The predicate is true if `d.get(n)` is a text file that contains the word `naming`. The code could just search all the text files, but a practical one will probably involve an auxiliary index structure that maps words to the files that contain them, and will probably not be perfectly coherent.

See the 'semantic file system' example below for more details and a reference.

Variations

It is useful to summarize the ways in which a spec for a name space might vary. The variations almost all have to do with the exact semantics of updates:

What operations are updates, that is, can change the results of `Read`?

Are there *aliases*, so that an update to one object can affect the value of others?

Are the updates *atomic*, or it is possible for reads to see intermediate states? Can an update be lost, or partly lost, if there is a crash?

Viewed as a memory, is the name space *coherent*? That is, does every read that follows an update see the update, or is it possible for the old state to hang around for a while?

How much can the set of defined `PN`'s change? In other words, is it useful to think about a *schema* for the name space that is separate from the current state?

Updates

If the directories are 'real', then there will be non-trivial `Write`, `MakeD`, and `Rename` operations. If they are not, these operations will always raise `error`, there will be operations to update the underlying data, and the view function will determine the effects of these updates on `Read` and `Enum`. In many systems, `Read` and `Write` cannot be modeled as operations on memory because `Write(a, r)` does not just change the value returned by `Read(a)`. Instead they must be understood as methods of (or messages sent to) some object.

The earliest example of this kind of system is the DEC Unibus, the prototype for modern I/O systems. Devices on such an I/O bus have 'device registers' that are named as locations in memory. You can read and write them with ordinary load and store instructions. Each device, however, is free to interpret these reads and writes as it sees fit. For example, a disk controller may have a set of registers into which you can write a command which is interpreted as "read `n` disk blocks starting at address `da` into memory starting at address `a`". This might take three writes, for the parameters `n`, `da`, and `a`, and the third write has the side effect of starting execution of the command.

The most recent well-known incarnation of this idea is the World Wide Web, in which read and write actions (called `Get` and `Post` in the protocol) are treated as messages to servers that can search databases, accept orders for pizza, or whatever.

Aliases

We have already discussed this topic at some length. Links and unions both introduce aliases. There can also be 'hard links', which are several occurrences of the same `D`. In a Unix file system, for example, it is possible to have several directory entries that point to the same file. A hard link differs from a soft link because the connection it establishes between a name and a file cannot be broken by changing the binding of some other name. And of course a view can introduce arbitrarily complicated aliasing. For example, it's fairly common for an I/O device that has internal memory to make that memory addressable with two control registers `a` and `v`, and the rule that a read or write of `v` refers to the internal memory location addressed by the current contents of `a`.

Atomicity

The `MemNames` and `ObjNames0` specs made all the update operations atomic. For code to satisfy these specs, it must hold some kind of lock on every directory touched by `GetDN`, or at least on the name looked up in each such directory. This can involve a lot of directories, and since the name space is a graph it also introduces the danger of deadlock. It's therefore common for systems to satisfy only the weaker atomicity spec of `ObjNames`, which says that looking up a simple name is atomic, but the entire lookup process is not.

This means that `Read(/a/x)` can return `3` even though there was never any instant at which the path name `/a/x` had the value `3`, or indeed was defined at all. To see how this can happen, suppose:

initially `/a` is the directory `d1` and `/b` is undefined;

initially `x` is undefined in `d1`;

concurrently with `Read(/a/x)` we do `Rename(/a, /b); Write(/b/x, 3)`.

The following sequence of actions yields `Read(/a/x) = 3`:

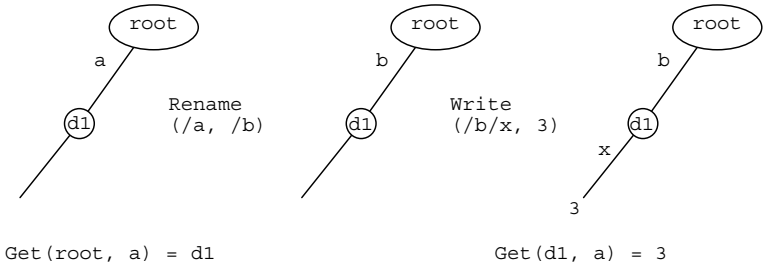
In the `Read`, `Get(root, a) = d1`

`Rename(/a, /b)` makes `/a` undefined and `d1` the value of `/b`

`Write(/b/x, 3)` makes `3` the value of `x` in `d1`

In the `Read`, `RET d1.get(x)` returns `3`.

Obviously, whether this possibility is important or not depends on how clients are using the name space.



Coherence

Other things being equal, everyone prefers a coherent or 'sequentially consistent' memory, in which there is a single order of all the concurrent operations with the property that the result of every read is the result that a simple memory would return after it has done all the preceding writes in order. Maintaining coherence has costs, however, in the amount of synchronization that is required if parts of the memory are cached, or in the amount of availability if the memory is replicated. We will discuss the first issue in detail at the end of the course. Here we consider the availability of a replicated memory.

Recall the majority register from the beginning of the course. It writes a majority of the replicas and reads from a majority, thus ensuring that every read must see the most recent write. However, this means that you can't do either a read or a write unless you can talk to a majority. There we used a general notion of majority in which the only requirement is that every two majorities have a non-empty intersection. Applying this idea, we can define separate read and write *quorums*, with the property that every read quorum intersects every write quorum. Then we can make reads more available by making every replica a read quorum, at the price of having the only write quorum be the set of all replicas, so that we have to do every write to all the replicas.

An alternative approach is to weaken the spec so that it's possible for a read to see old values. We have seen a version of this already in connection with crashes and write buffering, where it was possible for the system to revert to an old state after a crash. Now we propose to make the spec even more non-deterministic: you can read an old value at any time, and the only restriction is that you won't read a value older than the most recent `Sync`. In return, we can now have much more availability in the code, since both a read and a write can be done to a single replica. This means that if you do `Write(/a, 3)` and immediately read `a`, you may not get `3` because the `Read` might use a different replica that hasn't seen the `Write` yet. Only `Sync` requires communication among the replicas.

We give the spec for this as a variation on `ObjNames`. We allow `nil` to be in `dd(n)`, representing the fact that `n` has been undefined in `dd`.


```
TYPE DD          =  N -> SEQ Z                          % remember old values

APROC GetFromS(d, n) -> Z = <<                          % we write d.get(n)
% The non-determinism wouldn't be allowed if this were a function
     VAR z | z IN s(d)(n) => RET z [*] RET nil >>       % return any old value

PROC  SetToS(d, n, z) =                                 % we write d.set(n, z)
     s(d)(n) := ((s(d)!n => s(d)(n) [*] {}) + {z}       % add z to the state
```

```
PROC Sync(pn) RAISES {error} =
    VAR d, n, z |
         (d, n) := GetDN(pn, true); z := s(d)(n).last;
         IF z # nil => s(d)(n) := {z} [*] s(d) := s(d){n -> } FI
```

This spec is common in the naming service for a distributed system, for instance in the Internet's DNS or Microsoft's Active Directory. The name space changes slowly, it isn't critical to see the very latest value, and it *is* critical to have high availability. In particular, it's critical to be able to look up names even when network partitions make some working replicas unreachable.

Schemas

In the database world, a schema is the definition of what names are defined (and usually also of the type of each name's value).[6] Network management calls this a 'management information base' or MIB. Depending on the application there are very different rules about how the schema is defined.

In a file system, for example, there is usually no official schema written down. Nonetheless, each operating system has conventions that in practice have the force of law. A Unix system without `/bin` and `/etc` will not get very far. But other parts of the name space, especially in users' private directories, are completely variable.

By contrast, a database system takes the schema very seriously, and a management system takes at least some parts of it seriously. The choice has mainly to do with whether it is people or programs that are using the name space. Programs tend to be much less flexible; it's a lot of work to make them adapt to missing data or pay attention to unexpected additional data

Minor issues

We mention some other, less fundamental, ways in which the specs for name spaces differ.

Rules about overwriting. Some systems allow any name to be overwritten, others treat directories, or non-empty directories, specially to reduce the consequences of careless errors.

Access control. Many systems enforce rules about which users or programs are allowed to read or write various parts of the name space.

Resource control. Writes often consume resources that are expensive or in fixed supply, such as disk blocks. This means that they can fail if the resources are exhausted, and there may also be a quota system that limits the resource consumption of users or programs.

Roots

It's not turtles all the way down.

Anonymous

[6] Gray and Reuter, *Transaction Processing*, Morgan Kaufmann, 1993, pp 768-786.

So far we have ducked the question of how the `root` is represented, or the `D` in a link that plays a similar role. In `ObjNames0` we said `D = Int`, leaving its interpretation entirely to the `s` component of the state. In `ObjNames` we said `D` is a pair of procedures, begging the question of how the procedures are represented. The representation of a root depends entirely on the implementation. In a file system, for instance, a root names a disk, a disk partition, a volume, a file system exported from a server, or something like that. Thus there is another name space for the roots (another level of indirection). It works in a wide variety of ways. For example:

In MS-DOS. you name a physically connected disk drive. If the drive has removable media and you insert the wrong one, too bad.

On the Macintosh. you use the string name of a disk. If the system doesn't know where to find this disk, it asks the user. If you give the same name to two removable disks, too bad.

On Digital VMS. disks have unique identifiers that are used much like the string names on the Macintosh.

For the NFS network file system, a root is named by a host name or IP address, plus a file system name or handle on that host. If that name or address gets assigned to another machine, too bad.

In a network directory a root is named by a unique identifier. There is also a set of servers that might store replicas of that directory.

In the secure file system, a root is named by the hash of a public encryption key. There's also a network address to help you find the file system, but that's only a hint.[7]

In general it is a good idea to have absolute names (unique identifiers) for directories. This at least ensures that you won't use the wrong directory if the information about where to find it turns out to be wrong. A UID doesn't give much help in locating a directory, however. The possibilities are:

Store a set of places to look along with the UID. The problem is keeping this set up to date.

Keep another name space that maps UID's to locations (yet another level of indirection). The problem is keeping this name space up to date, and making it sufficiently available. For the former, every location can register itself periodically. For the latter, replication is good. We will talk about replication in detail later in the course.

Search some ad-hoc set of places in the hope of finding a copy. This search is often called a 'broadcast'.

We defined the interface routines to start from a fixed `root`. Some systems, such as Unix, have provisions for changing the root; the `chroot` system call does this for a process. In addition, it is common to have a more local context (called a 'working directory' for a file system), and to have syntax to specify whether to start from the root or the working directory (presence or absence of an initial '`/`' for a Unix file system).

Examples

These are to expand your mind and to help you recognize a name space when you come across it under some disguise.

[7] Mazières, Kaminsky, Kaashoek, and Witchel, Separating key management from file system security. *Proc. 17th ACM Symposium on Operating Systems Principles*, Dec. 1999. www.pdos.lcs.mit.edu/papers/sfs:sosp99.pdf.

File system directory

Example: `/udir/lampson/pocs/handouts/12-naming`

Not a tree, because of `.` and `..`, hard links, and soft links.
Devices, named pipes, and other things can appear as well as files.
Links and mounting are important for assembling the name space you want.
Files may have attributes, which are a little directory attached to the file.
Sometimes resources, fonts, and other OS rigmarole are stored this way.

inodes

There is a single inode directory, usually coded as a function rather than a table: you compute the location of the inode on the disk from the number.
For system-wide inodes, prefix a system-wide file system or volume name.

Plan 9[8]

This operating system puts all its objects into a single name space: files, devices, pipes, processes, display servers, and search paths (as union directories).

Semantic file system[9]

Not restricted to relational databases.
Free-text indexing: `~lampson/Mail/inbox/(word="compiler")`
Program cross-reference: `/project/sources/(calls="DeleteFile")`

Table (relational data base)

Example:

ID no (key)	*Name*	*Salary*	*Married?*
`1432`	`Smith`	`21,000`	`Yes`
`44563`	`Jones`	`35,000`	`No`
`8456`	`Brown`	`17,000`	`Yes`

We can view this as a naming tree in several ways:

`#44563/Name = Jones`	key's value is a `D` that defines `Name`, `Salary`, etc.
`Name/#44563 = Jones`	key's value is the `Name` field of its row

The second way, `cat Name/*` yields

```
Smith Jones Brown
```

Network naming[10]

Example: `theory.lcs.mit.edu`

Distributed code. Can share responsibility for following the path between client and server in many ways.
A directory handle is a machine address (interpreted by some communication network), plus some id for the directory on that machine.
Attractive as top levels of complete naming hierarchy.

E-mail addresses

Example: `rinard@lcs.mit.edu`

This syntax patches together the network name space and the user name space of a single host. Often there are links (called forwarding) and directories full of links (called distribution lists).

[8] Pike et al., The use of name spaces in Plan 9, *ACM Operating Systems Review* **27**, 2, Apr. 1993, pp 72-76.
[9] Gifford et al., Semantic file systems, *Proc. 13th ACM Symposium on Operating System Principles*, Oct. 1991, pp 16-25 (handout 13).
[10] B. Lampson, Designing a global name service, *Proc. 4th ACM Symposium on Principles of Distributed Computing*, Minaki, Ontario, 1986, pp 1-10. RFC 1034/5 for DNS.

SNMP[11] Example: Router with circuits, packets in circuits, headers in packets, etc.

Internet Simple Network Management Protocol
Roughly, view the state of the managed entity as a table, treating it as a name space the way we did earlier. You can read or write table entries.
The `Next` action allows a client to explore the name space, whose structure is read-only. Ad hoc `Write` actions are sometimes used to modify the structure, for instance by adding a row to a table.

Page tables Divide up the virtual address, using the first chunk to index a first level page table, later chunks for lower level tables, and the last chunk for the byte in the page.

Spec names Ex ample: `ObjNames.Enum`

LAN addresses 48-bit ethernet address. This is flat: the address is just a UID.

I/O device addressing Example:
Memory bus.
SCSI controller, by device register addresses.
SCSI device, by device number `0..7` on SCSI bus.
Disk sector, by disk address on unit.
Usually there is a pure read/write interface to the part of the I/O system that is named by memory addresses (the device registers in the example), and a message interface to the rest (the disk in the example).

Multiplexing a channel Examples:
Node-node network channel → *n* process-process channels.
Process-kernel channel → *n* inter-process channels.
ATM virtual path → *n* virtual circuits.

Given a channel, you can multiplex it to get sub-channels.
Sub-channels are identified by addresses in messages on the main channel.
This idea can be applied recursively, as in all good name spaces.

Hierarchical network addresses[12] Example: `16.24.116.42` (an IP address).

An address in a big network is hierarchical.
A router knows its parents and children, like a file directory, and also its siblings (because the parent might be missing)
To route, traverse up the name space to least common ancestor of current place and destination, then down to destination.

Network reference[13] Example: `6.24.116.42/11234/1223:44 9 Jan 1995/item 21`

Network address + port or process id + incarnation + more multiplexing + address or export index.
Some applications are remote procedure binding, network pointer, network object

[11] M. Rose, *The Simple Book*, Prentice-Hall, 1990.
[12] R. Perlman, *Connections*, Prentice-Hall, 1993.
[13] Andrew Birrell et al., Network objects, *Proc. 14th ACM Symposium on Operating Systems Principles*, Asheville, NC, Dec. 1993 (handout 25).

Abbreviations A, talking to B, wants to pass a big value `V`, say a font or security credentials.
A makes up a short name `N` for `V` (sometimes called a 'cookie', though it's not the same as a Web cookie) and passes that.
If B doesn't know `N`'s value `V`, it calls back to A to get it, and caches the result.
Sometimes A tells `V` to B when it chooses `N`, and B is expected to remember it.
This is not as good because B might run out of space or fail and restart.

World Wide Web Example: `http://ds.internic.net/ds/rfc-index.html`

This is the URL (Uniform Resource Locator) for Internet RFCs.
The Web has a read/write interface.

Telephone numbers Example: `1-617-253-6182`

Postal addresses Example:
```
Prof. Butler Lampson
Room 32-G924
MIT
Cambridge, MA 02139
```

13. Paper: Semantic File Systems

The attached paper by David Gifford, Pierre Jouvelot, Mark Sheldon, and James O'Toole was presented at the 13th ACM Symposium on Operating Systems Principles, 1991, and appeared in its proceedings, *ACM Operating Systems Review*, Oct. 1991, pp 16-25.

Read it as an adjunct to the lecture on naming

6.826—Principles of Computer Systems 2006

.

14. Practical Concurrency

We begin our study of concurrency by describing how to use it in practice; later, in handout 17 on formal concurrency, we shall study it more formally. First we explain where the concurrency in a system comes from, and discuss the main ways to express concurrency. Then we describe the difference between 'hard' and 'easy' concurrency[1]: the latter is done by locking shared data before you touch it, the former in subtle ways that are so error-prone that simple prudence requires correctness proofs. We give the rules for easy concurrency using locks, and discuss various issues that complicate the easy life: scheduling, locking granularity, and deadlocks.

Sources of concurrency

Before studying concurrency in detail, it seems useful to consider how you might get concurrency in your system. Obviously if you have a multiprocessor or a distributed system you will have concurrency, since in these systems there is more than one CPU executing instructions. Similarly, most hardware has separate parts that can change state simultaneously and independently. But suppose your system consists of a single CPU running a program. Then you can certainly arrange for concurrency by multiplexing that CPU among several tasks, but why would you want to do this? Since the CPU can only execute one instruction at a time, it isn't entirely obvious that there is any advantage to concurrency. Why not get one task done before moving on to the next one?

There are only two possible reasons:

1. A task might have to wait for something else to complete before it can proceed, for instance for a disk read. But this means that there is some concurrent task that is going to complete, in the example an I/O device, the disk. So we have concurrency in any system that has I/O, even when there is only one CPU.

2. Something else might have to wait for the result of one task but not for the rest of the computation, for example a human user. But this means that there is some concurrent task that is waiting, in the example the user. Again we have concurrency in any system that has I/O.

In the first case one task must wait for I/O, and we can get more work done by running another task on the CPU, rather than letting it idle during the wait. Thus the concurrency of the I/O system leads to concurrency on the CPU. If the I/O wait is explicit in the program, the programmer can know when other tasks might run; this is often called a 'non-preemptive' system, because it has sequential semantics except when the program explicitly allows concurrent activity by waiting. But if the I/O is done at some low level of abstraction, higher levels may be quite unaware of it. The most insidious example of this is I/O caused by the virtual memory system: every instruction can cause a disk read. Such a system is called 'preemptive'; for practical purposes a task can lose the CPU at any point, since it's too hard to predict which memory references might cause page faults.[2]

[1] I am indebted to Greg Nelson for this taxonomy, and for the object and set example of deadlock avoidance.
[2] Of course, if the system just waits for the page fault to complete, rather than running some other task, then the page fault is harmless.

In the second case we have a motivation for true preemption: we want some tasks to have higher priority for the CPU than others. An important special case is interrupts, discussed below.

A concurrent program is harder to write than a sequential program, since there are many more possible paths of execution and interactions among the parts of the program. The canonical example is two concurrent executions of

```
x := x + 1
```

Since this command is not atomic (either in Spec, or in C on most computers), `x` can end up with either `1` or `2`, depending on the order of execution of the expression evaluations and the assignments. The interleaved order

```
evaluate x + 1
evaluate x + 1
x := result
x := result
```

leaves `x = 1`, while doing both steps of one command before either step of the other leaves `x = 2`. This is called a *race*, because the two threads are racing each other to get `x` updated.

Since concurrent programs are harder to understand, it's best to avoid concurrency unless you really needed it for one of the reasons just discussed.[3] Unfortunately, it will very soon be the case that every PC is a multi-processor, since the only way to make productive use of all the transistors that Moore's law is giving us, without making drastic changes to the programming model, is to put several processors on each chip. It will be interesting to see what people do with all this concurrency.

One good thing about concurrency, on the other hand, is that when you write a program as a set of concurrent computations, you can defer decisions about exactly how to schedule them. More generally, concurrency can be an attractive way to decompose a large system: put different parts of it into different tasks, and carefully control their communication. We shall see some effective, if constraining, ways to do this.

We saw that in the absence of a multi-processor, the only reason for concurrency in your program is that there's something concurrent going on outside the program, usually an I/O operation. This external, or *heterogeneous* concurrency doesn't usually give rise to bugs by itself, since the concurrently running CPU and I/O device don't share any state directly and only interact by exchanging messages (though DMA I/O, when imprudently used, can make this false). We will concern ourselves from now on with *homogeneous* concurrency, where several concurrent computations are sharing the same memory.

Ways to package concurrency

In the last section we used the word 'task' informally to describe a more-or-less independent, more-or-less sequential part of a computation. Now we shall be less coy about how concurrency shows up in a system.

The most general way to describe a concurrent system is in terms of a set of atomic actions with the property that usually more than one of them can occur (is enabled); we will use this viewpoint in our later study of formal concurrency. In practice, however, we usually think in terms of

[3] This is the main reason why threads with RPC or synchronous messages are good, and asynchronous messages are bad. The latter force you to have concurrency whenever you have communication, while the former let you put in the concurrency just where you really need it. Of course if the implementation of threads is clumsy or expensive, as it often is, that may overwhelm the inherent advantages.

several 'threads' of concurrent execution. Within a single thread at most one action is enabled at a time; in general one action may be enabled from each thread, though often some of the threads are waiting or 'blocked', that is, have no enabled actions.

The most convenient way to do concurrent programming is in a system that allows each thread to be described as an execution path in an ordinary-looking program with modules, routines, commands, etc., such as Spec, C, or Java. In this scheme more than one thread can execute the code of the same procedure; threads have local state that is the local variables of the procedures they are executing. All the languages mentioned and many others allow you to program in this way.

In fault-tolerant systems there is a conceptual drawback to this thread model. If a failure can occur after each atomic command, it is hard to understand the program by following the sequential flow of control in a thread, because there are so many other paths that result from failure and recovery. In these systems it is often best to reason strictly in terms of independent atomic actions. We will see detailed examples of this when we study reliable messages, consensus, and replication. Applications programmed in a transaction system are another example of this approach: each application runs in response to some input and is a single atomic action.

The biggest drawback of this kind of 'official' thread, however, is the costs of representing the local state and call stack of each thread and of a general mechanism for scheduling the threads. There are several alternatives that reduce these costs: interrupts, control blocks, and SIMD computers. They are all based on restricting the freedom of a thread to block, that is, to yield the processor until some external condition is satisfied, for example, until there is space in a buffer or a lock is free, or a page fault has been processed.

Interrupts

An interrupt routine is not the same as a thread, because:

- It always starts at the same point.
- It cannot wait for another thread.

The reason for these restrictions is that the execution context for an interrupt routine is allocated on someone else's stack, which means that the routine must complete before the thread that it interrupted can continue to run. On the other hand, the hardware that schedules an interrupt routine is efficient and takes account of priority within certain limits. In addition, the interrupt routine doesn't pay the cost of its own stack like an ordinary thread.

It's possible to have a hybrid system in which an interrupt routine that needs to wait turns itself into an ordinary thread by copying its state. This is tricky if the wait happens in a subroutine of the main interrupt routine, since the relevant state may be spread across several stack frames. If the copying doesn't happen too often, the interrupt-thread hybrid is efficient. The main drawbacks are that the copying usually has to be done by hand, which is error-prone, and that without compiler and runtime support it's not possible to reconstruct the call stack, which means that the thread has to be structured differently from the interrupt routine.

A simpler strategy that is widely used is to limit the work in the interrupt routine to simple things that don't require waits, and to wake up a separate thread to do anything more complicated.

Control blocks and message queues

Another, related strategy is to package all the permanent state of a thread, including its program counter, in a record (usually called a 'control block') and to explicitly schedule the execution of the threads. When a thread runs, it starts at the saved program counter (usually a procedure entry point) and runs until it explicitly gives up control or 'yields'. During execution it can call procedures, but when it yields its stack must be empty so that there's no need to save it, because all the state has to be in the control block. When it yields, a reference to the control block is saved where some other thread or interrupt routine can find it and queue the thread for execution when it's ready to run, for instance after an I/O operation is complete.[4]

The advantages of this approach are similar to those of interrupts: there are no stacks to manage, and scheduling can be carefully tuned to the application. The main drawback is also similar: a thread must unwind its stack before it can wait. In particular, it cannot wait to acquire a lock at an arbitrary point in the program.

It is very common to code the I/O system of an operating system using this kind of thread. Most people who are used to this style do not realize that it is a restricted, though efficient, case of general programming with threads.

In 'active messages', a more recent variant of this scheme, you break your computation down into non-blocking segments; as the end of a segment, you package the state into an 'active message' and send it to the agent that can take the next step. Incoming messages are queued until the receiver has finished processing earlier ones.[5]

There are lots of other ways to use the control block idea. In 'scheduler activations', for example, kernel operations are defined so that they always run to completion; if an operation can't do what was requested, it returns intermediate state and can be retried later.[6] In 'message queuing' systems, the record of the thread state is stored in a persistent queue whenever it moves from one module to another, and a transaction is used to take the state off one queue, do some processing, and put it back onto another queue. This means that the thread can continue execution in spite of failures in machines or communication links.[7]

SIMD or data-parallel computing

This acronym stands for 'single instruction, multiple data', and refers to processors in which several execution units all execute the same sequence of instructions on different data values. In a 'pure' SIMD machine every instruction is executed at the same time by all the processors (except that some of them might be disabled for that instruction). Each processor has its own memory, and the processors can exchange data as part of an instruction. A few such machines were built between 1970 and 1993, but they are now out of favor.[8] The same programming paradigm is still used in many scientific problems however, at a coarser grain, and is called 'data-parallel' com-

[4] H. Lauer and R. Needham. On the duality of operating system structures. *Second Int. Symposium on Operating Systems*, IRIA, Rocquencourt, France, Oct. 1978 (reprinted in *Operating Systems Review* **13**,*2* (April 1979), 3-19).
[5] T. von Eiken et al., Active messages: A mechanism for integrated communication and computation. *Proc. International Symposium on Computer Architecture*, May 1992, pp 256-267.
[6] T. Anderson et al., Scheduler activations: Effective kernel support for the user-level management of parallelism. *ACM Transactions on Computer systems* **10**, 1 (Feb. 1992), pp 54-79.
[7] See www.messageq.com or A. Dickman, *Designing Applications With Msmq: Message Queuing for Developers*, Addison-Wesley, 1998.
[8] The term 'SIMD' has been recycled in the Intel MMX instruction set, and similar designs from several other manufacturers, to describe something much more prosaic: doing 8 8-bit adds in parallel on a 64-bit data path.

puting. In one step each processor does some computation on its private data. When all of them are done, they exchange some data and then take the next step. The action of detecting that all are done is called 'barrier synchronization'.

Streams and vectors

Another way to package concurrency is using vectors or streams of data, sequences (or sets) of data items that are all processed in a similar way. This has the advantage that a single action can launch a lot of computation; for example, to add the 1000-item vectors a_i and b_i. Streams can come from graphics or database applications as well as from scientific computing.

Some simple examples of concurrency

Here are a number of examples of concurrency. You can think of these as patterns that might apply to part of your problem

Vector and matrix computations have a lot of inherent concurrency, since the operations on distinct indices are usually independent. Adding two vectors can use as much concurrency as there are vector elements (except for the overhead of scheduling the independent additions). More generally, any scheme that *partitions* data so that at most one thread acts on a single partition makes concurrency easy. Computing the scalar product of two vectors is trickier, since the result is a single sum, but the multiplies can be done in parallel, and the sum can be computed in a tree. This is a special case of a *combining tree*, in which siblings only interact at their parent node. A much more complex example is a file system that supports map-reduce.[9]

Histograms, where you map each item in some set into a bucket, and count the number of items in each bucket. Like scalar product, this has a fully partitioned portion where you do the mapping, and a shared-data portion where you increment the counts. There's more chance for concurrency without interference because the counts for different buckets are partitioned.

Divide and conquer programs usually have lots of concurrency. Consider the Fibonacci function, for example, defined by

```
FUNC Fib(n) -> Int = RET (n < 2 => 1 [*] Fib(n-1) + Fib(n-2))
```

The two recursive calls of `Fib` can run concurrently. Unfortunately, there will be a lot of duplicated work; this can be avoided by caching the results of sub-computations, but the cache must be accessed concurrently by the otherwise independent sub-computations. Both the basic idea and the problem of duplicated work generalize to a wide range of functional programs. Whether it works for programs with state depends on how much interaction there is among the divisions.

Read-compute-write computations can run at the speed of the slowest part, rather than of the sum. This is a special case of pipelining, discussed in more detail later. Ideally the load will be balanced so that each of the three phases takes the same amount of time. This organization usually doesn't improve latency, since a given work item has to move through all three phases, but it does improve bandwidth by the amount of concurrency.

Background or speculative computations, such as garbage collection, spell checking, prefetching data, synchronizing replicas of data, ripping CDs, etc.

[9] Jeffrey Dean and Sanjay Ghemawat, MapReduce: Simplified Data Processing on Large Clusters, *OSDI 2004*.

Easy concurrency

Concurrency is easy when you program with locks. The rules are simple:

- Every shared variable must be protected by a lock. A variable is shared if it is touched by more than one thread. Alternatively, you can say that *every* variable must be protected b y a lock, and think of data that is private to a thread as being protected by an implicit lock that is always held by the thread.
- You must hold the lock for a shared variable before you touch the variable. The essential property of a lock is that two threads can't hold the same lock at the same time. This property is called 'mutual exclusion'; the abbreviation 'mutex' is another name for a lock.
- If you want an atomic operation on several shared variables that are protected by different locks, you must not release any locks until you are done. This is called 'two-phase locking', because there is a phase in which you only acquire locks and don't release any, followed by a phase in which you only release locks and don't acquire any.

Then your computation between the point that you acquire a lock and the point that you release it is equivalent to a single atomic action, and therefore you can reason about it sequentially. This atomic part of the computation is called a 'critical section'. To use this method reliably, you should annotate each shared variable with the name of the lock that protects it, and clearly bracket the regions of your program within which you hold each lock. Then it is a mechanical process to check that you hold the proper lock whenever you touch a shared variable.[10] It's also possible to check a running program for violations of this discipline.[11]

Why do locks lead to big atomic actions? Intuitively, the reason is that no other well-behaved thread can touch any shared variable while you hold its lock, because a well-behaved thread won't touch a shared variable without itself holding its lock, and only one thread can hold a lock at a time. We will make this more precise in handout 17 on formal concurrency, and give a proof of atomicity. Another way of saying this is that locking ensures that concurrent operations *commute*. Concurrency means that we aren't sure what order they will run in, but commuting says that the order doesn't matter because the result is the same in either order.

Actually locks give you a bit more atomicity than this. If a well-behaved thread acquires a sequence of locks (acquiring each one before touching the data it protects) and then releases them (not necessarily in the same order, but releasing each one after touching the data it protects), the entire computation from the first acquire to the last release is atomic. Once you have done a release, however, you can't do another acquire without losing atomicity. This is called *two-phase locking*.

The simple locks we have been describing are also called 'mutexes'; this is short for "mutual exclusion". As we shall see, more complicated kinds of locks are often useful.

Here is the spec for a mutex. It maintains mutual exclusion by allowing the mutex to be acquired only when no one already holds it. If a thread other than the current holder releases the mutex, the result is undefined. If you try to do an `Acquire` when the mutex is not free, you have to wait, since `Acquire` has no transition from that state because of the `m = nil` guard.

[10] This process is mechanized in ESC; see http://www.research.digital.com/SRC/esc/Esc.html.
[11] S. Savage et al. Eraser: A dynamic data race detector for multithreaded programs. *ACM Transactions on Computer Systems* **15**, 4 (Dec 1997), pp 391-411.

```
MODULE Mutex EXPORT acq, rel =                    % Acquire and Release

VAR m: (Thread + Null) := nil
% A mutex is either nil or the thread holding the mutex.
% The variable SELF is defined to be the thread currently making a transition.

APROC acq() = << m = nil  => m := SELF; RET >>
APROC rel() = << m = SELF => m := nil ; RET [*] HAVOC >>

END Mutex
```

The thing we care about is that only one thread can be between `acq` and `rel` at a time. It's pretty obvious from the spec that this is true as long as we never get to `HAVOC` in `rel`. We can make it explicit with an invariant:

```
INVARIANT (ALL h, h' | h in critical section ==> ~ h' in critical section)
```

Here we have treated the PC's very informally; see handout 17 for the precise details. This invariant follows from

```
INVARIANT (ALL h | h in critical section ==> m = h)
```

which in turn follows by induction on the actions of `h` that don't include `rel`, plus the fact that no thread `h'` does `m.rel` unless `m = h'`. An invariant like this on the spec is sometimes called a *property*. The model-checking proof of an implementation of `Mutex` at the end of handout 17 shows how to establish a property directly from the implementation. This is contrary to the religion of this course, which is to always do simulation proofs, but it can be effective nonetheless.

We usually need lots of mutexes, not just one, so we change `MODULE` to `CLASS` (see section 7 of handout 4, the Spec reference manual). This creates a module with a function variable in which to store the state of lots of mutexes, and a `Mutex` type with `new`, `acq`, and `rel` methods whose value indexes the variable.

If `m` is a mutex that protects the variable `x`, you use it like this:

```
m.acq; touch x; m.rel
```

That is, you touch `x` only while `m` is acquired.

You may be familiar with this style of programming from Java, where a synchronized object has an implicit lock that is automatically acquired at the start of every method and released at the end. This means that all the private fields are protected by the implicit lock. Another way to think about this style of programming is that the private fields are internal state of an isolated system. Only actions of this system can touch the fields, and only one such action runs at a time. As long as the actions don't touch any other objects, they are obviously atomic. When an action needs to touch more than one object this simple view is no longer adequate. We explore some of the complications below.

Note that Java locks differ from the ones we have specified in that they are *re-entrant*: the same thread can acquire the same lock repeatedly. In the spec above this would lead to deadlock.

The only problem with the lock-before-touching rule for easy concurrency is that it's easy to make a mistake and program a race, though tying the lock to a class instance makes mistakes less likely. Races are bad bugs to have, because they are hard to reproduce; for this reason they are often called *Heisenbugs*, as opposed to the deterministic *Bohrbugs*. There is a substantial literature on methods for statically detecting failure to follow the locking rules.

Invariants

In fact things are not so simple, since a computation seldom consists of a single atomic action. A thread should not hold a lock forever (except on private data) because that will prevent any other thread that needs to touch the data from making progress. Furthermore, it often happens that a thread can't make progress until some other thread changes the data protected by a lock. A simple example of this is a FIFO buffer, in which a consumer thread doing a `Get` on an empty buffer must wait until some other producer thread does a `Put`. In order for the producer to get access to the data, the consumer must release the lock. Atomicity does not apply to code like this that touches a shared variable `x` protected by a mutex `m`:

```
m.acq; touch x; m.rel; private computation; m.acq; touch x; m.rel
```

This code releases a lock and later re-acquires it, and therefore isn't atomic. So we need a different way to think about this situation, and here it is.

> After the `m.acq` the only thing you can assume about `x` is an invariant that holds whenever `m` is unlocked.

As usual, the invariant must be true initially. While `m` is locked you can modify `x` so that the invariant doesn't hold, but you must re-establish it before unlocking `m`. While `m` is locked, you can also poke around in `x` and discover facts that are not implied by the invariant, but you cannot assume that any of these facts are still true after you unlock `m`.

To use this methodology effectively, of course, you must *write the invariant down*.

The rule about invariants sheds some light on why the following simple locking strategy doesn't help with concurrent programming:

> Every time you touch a shared variable `x`, acquire a lock just before and release the lock just after.

The reason is that once you have released the lock, you can't assume anything about `x` except what is implied by the invariant. The whole point of holding the lock is that it allows you to know more about `x` as long as you continue to hold the lock.

Here is a more picturesque way of describing this method. To do easy concurrent programming:

> first you put your hand over some shared variables, say `x` and `y`, so that no one else can change them,
>
> then you look at them and perhaps do something with them, and
>
> finally you take your hand away.

The reason `x` and `y` can't change is that the rest of the program obeys some conventions; in particular, it acquires locks before touching shared variables. There are other, trickier conventions that can keep `x` and `y` from changing; we will see some of them later on.

This viewpoint sheds light on why fault-tolerant programming is hard: `Crash` is no respecter of conventions, and the invariant must be maintained even though a `Crash` may stop an update in mid-flight and reset all or part of the volatile state.

Scheduling: Condition variables

If a thread can't make progress until some condition is established, and therefore has to release a lock so that some other thread can establish the condition, the simplest idiom is

```
m.acq; DO ~ condition(x) involving x => m.rel; m.acq OD; touch x; m.rel
```

That is, you loop waiting for `condition(x)` to be true before touching `x`. This is called "busy waiting", because the thread keeps computing, waiting for the condition to become true. It tests `condition(x)` only with the lock held, since `condition(x)` touches `x`, and it keeps releasing the lock so that some other thread can change `x` to make `condition(x)` true.

This code is correct, but reacquiring the lock immediately makes it more difficult for another thread to get it, and going around the loop while the condition remains false wastes processor cycles. Even if you have your own processor, this isn't a good scheme because of the system-wide cost of repeatedly acquiring the lock.

The way around these problems is an optimization that replaces `m.rel; m.acq` in the box with `c.wait(m)`, where `c` is a 'condition variable'. The `c.wait(m)` releases `m` and then blocks the thread until some other thread does `c.signal`. Then it reacquires `m` and returns. If several threads are waiting, `signal` picks one or more to continue in a fair way. The variation `c.broadcast` continues all the waiting threads.

Here is the spec for condition variables. It says that the state is the set of threads waiting on the condition, and it allows for lots of `C`'s because it's a class. The `wait` method is especially interesting, since it's the first procedure we've seen in a spec that is not atomic (except for the clumsy non-atomic specs for disk and file writes, and `ObjNames`). This is because the whole point is that during the `wait` other threads have to run, access the variables protected by the mutex, and signal the condition variable. Note that `wait` takes an extra parameter, the mutex to release and reacquire.

The spec doesn't say anything about blocking or suspending the thread. The blocking happens at the semi-colon between the two atomic actions of `wait`. An implementation works by keeping a queue of blocked threads in the condition variable; `signal` takes a thread off this queue and makes it ready to run again. Of course the code must take care to make the queuing and blocking of the thread effectively atomic, so that the thread doesn't get unqueued and scheduled to run again before it has been suspended. It must also take care not to miss a `signal` that occurs between queuing `SELF` on `c` and blocking the thread. This is usually done with a 'wakeup-waiting switch', a bit in the thread state that is set by `signal` and checked atomically with blocking the thread. See `MutexImpl` and `ConditionImpl` in handout 17 for an example of how to do this implementation.

```
CLASS Condition EXPORT wait, signal, broadcast =

TYPE M = Mutex

VAR c           :  SET Thread := {}
% Each condition variable is the set of waiting threads.

PROC wait(m) =
    << c \/ := {SELF}; m.rel >>;                    % m.rel=HAVOC unless SELF IN m
    << ~ (SELF IN c) => m.acq >>

APROC signal() = <<
% Remove at least one thread from c.  In practice, usually just one.
    IF  VAR t: SET Thread | t <= c /\ t # {} => c - := t [*] SKIP FI >>

APROC broadcast() = << c := {} >>

END Condition
```

For this scheme to work, a thread that changes `x` so that the condition becomes true must do a `signal` or `broadcast`, in order to allow some waiting thread to continue. A foolproof but inefficient strategy is to have a single condition variable for `x` and to do `broadcast` whenever `x` changes at all. More complicated schemes can be more efficient, but are more likely to omit a `signal` and leave a thread waiting indefinitely. The paper by Birrell in handout 15[12] gives many examples and some good advice.

Note that you are *not* entitled to assume that the condition is true just because `wait` returns. That would be a little more efficient for the waiter, but it would be much more error prone, and it would require a tighter spec for `wait` and `signal` that is often less efficient to code. You are supposed to think of `c.wait(m)` as just an optimization of `m.rel; m.acq`. This idiom is very robust. Warning: many people don't agree with this argument, and define stronger condition variables; when reading papers on this subject, make sure you know what religion the author embraces.

More generally, after `c.wait(m)` you cannot assume anything about `x` beyond its invariant, since the `wait` unlocks `m` and then locks it again. After a `wait`, only the invariant is guaranteed to hold, not anything else that was true about `x` before the wait.

Really easy concurrency

An even easier kind of concurrency uses buffers to connect independent modules, each with its own set of variables disjoint from those of any other module. Each module consumes data from some predecessor modules and produces data for some successor modules. In the simplest case the buffers are FIFO, but they might be unordered or use some other ordering rule. A little care is needed to program the buffers' `Put` and `Get` operations, but that's all. This is often called 'pipelining'. The fancier term 'data flow' is used if the modules are connected not linearly but by a more general DAG.

A second really easy kind of concurrency is pure data parallelism, as in the example earlier of adding two vectors to get a third. Here there is no data shared among the threads, so no locking is needed. Unfortunately, pure data parallelism is rare—usually there is some shared data to spoil the purity, as in the example of scalar product. The same kind of thing happens in graphics: when Photoshop operates on a large array of pixels, it's possible that the operation is strictly per-pixel, but more often it involves a neighborhood of each pixel, so that there is some sharing.

A third really easy kind of concurrency is provided by transaction processing or TP systems, in which an application program accepts some input, reads and updates a shared database, and generates some output. The transaction mechanism makes this entire operation atomic, using techniques that we will describe later. The application programmer doesn't have to think about concurrency at all. In fact, the atomicity usually includes crash recovery, so she doesn't have to think about fault-tolerance either.

[12] Andrew Birrell, *An Introduction to Programming with C# Threads*, Research Report , Microsoft Corporation, May 2005 (handout 16).

In the pure version of TP, there is *no* state preserved outside the transaction except for the shared database. This means that the only invariants are invariants on the database; the programmer doesn't have to worry about mistakenly keeping private state that records something about the shared state after locks are released. Furthermore, it means that a transaction can run on any machine that can access the database, so the TP system can take care of launching programs and doing load balancing as well as locking and fault tolerance. How easy can it get?

A fourth way to get really easy concurrency is functional programs. If two threads operate on data that is either *immutable* or private to a thread, they can run concurrently without any complications. This happens automatically in *functional* programs, in which there is *no* mutable state but only the result of function executions. The most widely used example of a functional programming system is the SQL language for writing queries on relational databases. SQL lets you combine (join, in technical terms), project, filter, and aggregate information from tables that represent relations. SQL queries don't modify the tables, but simply produce results, so they can easily be run in parallel, and since there is a large market for large queries on large databases, much effort have been successfully invested in figuring out how to run such queries efficiently on machines with dozens or hundreds of processors.

More commonly a language is mostly functional: most of the computation is functional, but the results of functional computations are used to change the state. The grandfather of such languages is APL, which lets you write big functional computations on vectors and matrices, but then assigns the results to variables that are used in subsequent computations. Systems like Matlab and Mathematica are the modern descendants of APL.

Hard concurrency

If you don't program according to the rules for locks, then you are doing hard concurrency, and it will be hard. Why bother? There are three reasons:

You may have to code mutexes and condition variables on top of something weaker, such as the atomic reads and writes of memory that a basic processor or file system gives you. Of course, only the low-level runtime implementer will be in this position.

It may be cheaper to use weaker primitives than mutexes. If efficiency is important, hard concurrency may be worth the trouble. But you will pay for it, either in bugs or in careful proofs of correctness.

It may be important to avoid waiting for a lock to be released. Even if a critical section is coded carefully so that it doesn't do too much computing, there are still ways for the lock to be held for a long time. If the thread holding the lock can fail independently (for example, if it is in a different address space or on a different machine), then the lock can be held indefinitely. If the thread can get a page fault while holding the lock, then the lock can be held for a disk access time. A concurrent algorithm that prevents one slow (or failed) thread from delaying other threads too much is called 'wait-free'.[13]

[13] M. Herlihy. Wait-free synchronization. *ACM Transactions on Programming Languages and Systems* **13**, 1 (Jan. 1991), pp 124-149. There is a general method for implementing wait-free concurrency, given a primitive at least as strong as compare-and-swap; it is described in M. Herlihy. A methodology for implementing highly concurrent data objects. *ACM Transactions on Programming Languages and Systems* **15,** 9 (Nov. 1993), pp 745-770. The idea is the same as optimistic concurrency control (see handout 20): do the work on a separate version of the state, and then install it atomically with compare-and-swap, which detects when someone else has gotten ahead of you.

In fact, the "put out your hand" way of looking at things applies to hard concurrency as well. The difference is that instead of preventing `x` and `y` from changing at all, you do something to ensure that some predicate `p(x, y)` will remain true. The convention that the rest of the program obeys may be quite subtle. A simple example is the careful write solution to keeping track of free space in a file system (handout 7 on formal concurrency, page 16), in which the predicate is

```
free(da) ==> ~ Reachable(da).
```

The special case of locking maintains the strong predicate `x = x0 /\ y = y0` (unless you change `x` or `y` yourself).

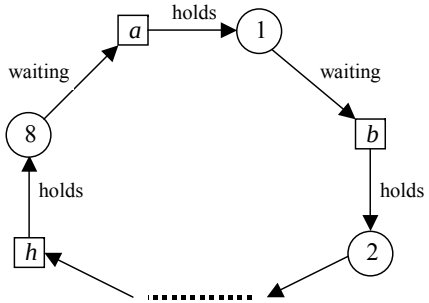
We postpone a detailed study of hard concurrency to handout 17.

Problems in easy concurrency: Deadlock

The biggest problem for easy concurrency is deadlock, in which there is a cycle of the form

Lock *a* is held by thread 1.
Thread 1 is waiting for lock *b*.
Lock *b* is held by thread 2.
...
Lock *h* is held by thread 8.
Thread 8 is waiting for lock *a*.

All the locks and threads are nodes in a lock graph with the edges "lock *a* is held by thread 1", "thread 1 is waiting for lock *b*", etc.



The way to deal with this that is simplest for the application programmer is to *detect* a deadlock[14] and automatically roll back one of the threads, undoing any changes it has made and releasing its locks. Then the rolled-back thread retries; in the meantime, the others can proceed. Unfortunately, this approach is only practical when automatic rollback is possible, that is, when all the changes are done as part of a transaction. Handout 19 on sequential transactions explains how this works.

Note that from inside a module, absence of deadlock is a safety property: something bad doesn't happen. The "bad" thing is a loop of the kind just described, which is a well-defined property of certain states, indeed, one that is detected by systems that do deadlock detection. From the outside, however, you can't see the internal state, and the deadlock manifests itself as the failure of the module to make any progress.

[14] For ways of detecting deadlocks, see Gray and Reuter, pp 481-483 and A. Thomasian, Two phase locking performance and its thrashing behavior. *ACM Transactions on Database Systems* **18**, 4 (Dec. 1993), pp. 579-625.

The main alternative to deadlock detection and rollback is to *avoid* deadlocks by defining a partial order on the locks, and abiding by a rule that you only acquire a lock if it is greater than every lock you already hold. This ensures that there can't be any cycles in the graph of threads and locks. Note that there is no requirement to release the locks in order, since a release never has to wait.

To implement this idea you

annotate each shared variable with its protecting lock (which you are supposed to do anyway when practicing easy concurrency),

state the partial order on the locks, and

annotate each procedure or code block with its 'locking level' `ll`, the maximum lock that can be held when it is entered, like this: `ll <= x`.

Then you always know textually the biggest lock that can be held (by starting at the procedure entry with the annotation, and adding locks that are acquired), and can check whether an `acq` is for a bigger lock as required, or not. With a stronger annotation that tells exactly what locks are held, you can subtract those that are released as well. You also have to check when you call a procedure that the current locking level is consistent with the procedure's annotation. This check is very similar to type checking.

Having described the basic method, we look at some examples of how it works and where it runs into difficulties.

If resources are arranged in a tree and the program always traverses the tree down from root to leaves, or up from leaves to root (in the usual convention, which draws trees upside down, with the root at the top), then the tree defines a suitable lock ordering. Examples are a strictly hierarchical file system or a tree of windows. If the program sometimes goes up and sometimes goes down, there are problems; we discuss some solutions shortly. If instead of a tree we have a DAG, it still defines a suitable lock ordering.

Often, as in the file system example, this graph is actually a data structure whose links determine the accessibility of the nodes. In this situation you can choose when to release locks. If the graph is static, it's all right to release locks at any time. If you release each lock before acquiring the next one, there is no danger of deadlock regardless of the structure of the graph, because a flat ordering (everything unordered) is good enough as long as you hold at most one lock at a time. If the graph is dynamic and a node can disappear when it isn't locked, you have to hold on to one lock at least until after you have acquired the next one. This is called 'lock coupling', and a cyclic graph can cause deadlock. We will see an example of this when we study hierarchical file systems in handout 15.

Here is another common locking pattern. Consider a program that manipulates objects named by handles and maintains a set of these objects. For example, the objects might be buffers, and the set the buffers that are non-empty. One thread works on an object and sometimes needs to mess with the set, for instance when a buffer changes from empty to non-empty. Another thread processes the set and needs to mess with some of the objects, for instance to empty out the buffers at regular intervals. It's natural to have a lock `h.m` on each object and a lock `ms` on the set. How should they be ordered? We work out a solution in which the ordering of locks is every `h.m < ms`.


```
TYPE H          =  Int WITH {acq:=(\h|ot(h).m.acq),   % Handle (index in ot)
                             rel:=(\h|ot(h).m.rel),
                             y  :=(\h|ot(h).y ), empty:=...}


VAR  s          :  SET H                              % ms protects the set s
     ms         :  Mutex
     ot         :  H -> [m: Mutex, y: Any]            % Object Table. m protects y,
                                                      % which is the object's data
```

Note that each piece of state that is not a mutex is annotated with the lock that protects it: `s` with `ms` and `y` with `m`. The 'object table' `ot` is fixed and therefore doesn't need a lock.

We would like to maintain the invariant "object is non-empty" = "object in set": `~ h.empty = h IN s`. This requires holding both `h.m` and `ms` when the emptiness of an object changes. Actually we maintain "`h.m` is locked `\/ (~ h.empty = h IN s)`", which is just as good. The `Fill` procedure that works on objects is very straightforward; `Add` and `Drain` are functions that compute the new state of the object in some unspecified way, leaving it non-empty and empty respectively. Note that `Fill` only acquires `ms` when it becomes non-empty, and we expect this to happen on only a small fraction of the calls.

```
PROC Fill(h, x: Any) =
% Update the object h using the data x
     h.acq;
     IF h.empty => ms.acq; s \/ := {h}; ms.rel [*] SKIP FI;
     ot(h).y := Add(h.y, x);
     h.rel
```

The `Demon` thread that works on the set is less straightforward, since the lock ordering keeps it from acquiring the locks in the order that is natural for it.

```
THREAD Demon() = DO
     ms.acq;
     IF   VAR h | h IN s  =>
               ms.rel;
               h.acq; ms.acq;                         % acquire locks in order
               IF   h IN s =>                         % is h still in s?
                    s - := {h}; ot(h).y := Drain(h.y)
               [*]  SKIP
               FI;
               ms.rel; h.rel
     [*]  ms.rel
     FI
  OD
```

`Drain` itself does no locking, so we don't show its body.

The general idea, for parts of the program like `Demon` that can't acquire locks in the natural order, is to collect the information you need, one mutex at a time, without making any state changes. Then reacquire the locks according to the lock ordering, check that things haven't changed (or at least that your conclusions still hold), and do the updates. If it doesn't work out, retry. Version numbers can make the 'didn't change' check cheap. This scheme is closely related to optimistic concurrency control, which we discuss later in connection with concurrent transactions.

An alternative approach in the hybrid scheme allows you to make state changes while acquiring locks, but then you must undo all the changes before releasing the locks. This is called 'compen-

sation'. It makes the main line code path simpler, and it may be more efficient on average, but coding the compensating actions and ensuring that they are always applied can be tricky.

It's possible to use a hybrid scheme in which you keep locks as long as you can, rather than preparing to acquire a lock by always releasing any larger locks. This works if you can acquire a lower lock 'cautiously', that is, with a failure indication rather than a wait if you can't get it. If you fail in getting a lower lock, fall back to the conservative scheme of the last paragraph. This doesn't simplify the code (in fact, it makes the code more complicated), but it may be faster.

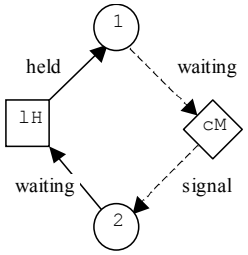
Deadlock with condition variables: Nested monitors

Since a thread can wait on a condition variable as well as on a lock, it's possible to have a deadlock that involves condition variables as well as locks. Usually this isn't a problem because there are many fewer conditions than locks, and the thread that signals a condition is coupled to the thread that waits on it only through the single lock that the waiting thread releases. This is fortunate, because there is no simple rule like the ordering rule for locks that can avoid this kind of deadlock. The lock ordering rule depends on the fact that a thread must be holding a lock in order to keep another thread waiting for that lock. In the case of conditions, the thread that will signal can't be distinguished in such a simple way.

The canonical example of deadlock involving conditions is known as "nested monitors". It comes up when there are two levels of abstraction, `H` and `M` (for high and medium; low would be confused with the `L` of locks), each with its own lock `lH` and `lM`. `M` has a condition variable `cM`. The code that deadlocks looks like this, if two threads `1` and `2` are using `H`, `1` needs to wait on `cM`, and `2` will signal `cM`.

```
H1: lH.lock; call M1
M1: lM.lock; cM.wait(lM)

H2: lH.lock; call M2
M2: lM.lock; cM.signal
```

This will deadlock because the `wait` in `M1` releases `lM` but not `lH`, so that `H2` can never get past `lH.lock` to reach `M2` and do the `signal`. This is not a lock-lock deadlock because it involves the condition variable `cM`, so a straightforward deadlock detector will not find it. The picture below illustrates the point.



To avoid this deadlock, don't wait on a condition with *any* locks held, unless you know that the `signal` can happen without acquiring any of these locks. The 'don't wait' is simple to check, given the annotations that the methodology requires, but the 'unless' may not be simple.

People have proposed to solve this problem by generalizing `wait` so that it takes a set of mutexes to release instead of just one. Why is this a bad idea? Aside from the problems of passing the right mutexes down from `H` to `M`, it means that any call on `M` might release `lH`. The `H` programmer

must to be careful not to depend on anything more than the `lH` invariant across any call to `M`. This style of programming is very error-prone.

Problems in easy concurrency: Scheduling

If there is a shortage of processor resources, there are various ways in which the simple easy concurrency method can go astray. In this situation we may want some threads to have priority over others, but subject to this constraint we want the processor resources allocated fairly. This means that the amount of time a task takes should be roughly proportional to the amount of work it does; in particular, we don't want short tasks to be blocked by long ones. A naïve view of fairness gives poor results, however. If all the tasks are equally important, it seems fair to give them equal shares of the CPU, but consider what happens if two tasks of the same length l start at the same time. With equal shares, both will take time $2l$ to finish. Running one to completion, however, means that it finishes in time l, while the other still takes only $2l$, for an average completion time of $1.5l$, which is clearly better.

What if all tasks are not equally important? There are various way this can happen. If there are deadlines, usually it's best to run the task with the closest deadline first. If you want to give different shares of the resource to different tasks, there's a wide variety of schemes to choose from. The simplest to understand is lottery scheduling, which gives each task lottery tickets in proportion to its share, and then chooses a ticket at random and runs that task.[15]

Starvation

If threads are competing for a resource, it's important to schedule them *fairly*. CPU time, just discussed, is not the only resource. In fact, the most common resources that need to be scheduled are locks. The specs we gave for locks and condition variables say nothing about the order in which competing threads acquire a lock when it's released. If a thread can be repeatedly passed over for `acq`, it will make no progress, and we say that it's *starved*. This is obviously bad. It's easy to avoid simple cases of starvation by keeping the waiting threads in a queue and serving the one at the head of the queue. This scheme fails if a thread needs to release and re-acquire locks to get its job done, as in some of the schemes for handling deadlock discussed above. Methods that detect a deadlock and abort one of the threads, requiring it to reacquire its locks, may never give a thread a chance to get all the locks it needs, and the `Demon` thread has the same problem. The opposite of starvation is *progress*.

A variation on starvation that is less serious but can still have crippling effects on performance is the *convoy* phenomenon, in which there is a high-traffic resource protected by a lock which is acquired frequently by many threads. The resource should be designed so that the lock only needs to be held for a short time; then the lock will normally be free and the resource won't be a bottleneck. If there is a pre-emptive scheduler, however, a thread can acquire the lock and get pre-empted. Lots of other threads will then pile up waiting for the lock. Once you get into this state it's hard to get out of it, because a thread that acquires the lock will quickly release it, but then, since it keeps the CPU, attempts to acquire the lock again and end up at the end of the queue, so that the queue never gets empty as it's supposed to and the computation is effectively

[15] C.A. Waldspurger and W.E. Weihl. Lottery scheduling: Flexible proportional-share resource management. In *Proceedings of the First Symposium on Operating Systems Design and Implementation*, pages 1–11, November 1994.

serialized. The solution is to immediately give the CPU to a waiting thread when the lock is released if the queue is long; this increases scheduling overhead, but only until the queue empties.

Priority inversion

When there are priorities there can be "priority inversion". This happens when a low-priority thread `A` acquires a lock and then loses the CPU, either to a higher-priority thread or to round-robin scheduling. Now a high-priority thread `B` tries to acquire the lock and ends up waiting for `A`. Clearly the priority of `A` should be temporarily increased to that of `B` until `A` completes its critical section, so that `B` can continue. Otherwise `B` may wait for a long time while threads with priorities between `A` and `B` run, which is not what we had in mind when we set up the priority scheme. Unfortunately, many thread systems don't raise `A`'s priority in this situation.

Granularity of locks

A different issue is the 'granularity' of the locks: how much data each lock protects. A single lock is simple and cheap, but doesn't allow any concurrency. Lots of fine-grained locks allow lots of concurrency, but the program is more complicated, there's more overhead for acquiring locks, and there's more chance for deadlock (discussed earlier). For example, a file system might have a single global lock, one lock on each directory, one lock on each file, or locks only on byte ranges within a file. The goal is to have fine enough granularity that the queue of threads waiting on a mutex is empty most of the time. More locks than that don't accomplish anything.

It's possible to have an adaptive scheme in which locks start out fine-grained, but when a thread acquires too many locks they are collapsed into fewer coarser ones that cover larger sets of variables. This process is called 'escalation'. It's also possible to go the other way: a process keeps track of the exact variables it needs to lock, but takes out much coarser locks until there is contention. Then the coarse locks are 'de-escalated' to finer ones until the contention disappears.

Closely related to the choice of granularity is the question of how long locks are held. If a lock that protects a lot of data is held for a long time (for instance, across a disk reference or an interaction with the user) concurrency will obviously suffer. Such a lock should protect the minimum amount of data that is in flux during the slow operation. The concurrent buffered disk example in handout 15 illustrates this point.

On the other hand, sometimes you want to minimize the amount of communication needed for acquiring and releasing the same lock repeatedly. To do this, you hold onto the lock for longer than is necessary for correctness. Another thread that wants to acquire the lock must somehow signal the holder to release it. This scheme is commonly used in distributed coherent caches, in which the lock only needs to be held across a single read, write, or test-and-set operation, but one thread may access the same location (or cache line) many times before a different thread touches it.

Lock modes

Another way to get more concurrency at the expense of complexity is to have many lock 'modes'. A mutex has one mode, usually called 'exclusive' since 'mutex' is short for 'mutual exclusion'. A reader/writer lock has two modes, called exclusive and 'shared'. It's possible to have as many modes as there are different kinds of commuting operations. Thus all reads commute and therefore need only shared mode (reader) locks. But a write commutes with nothing and therefore needs an exclusive mode (write) lock. The commutativity of the operations is reflected in a 'conflict relation' on the locks. For reader/writer or shared/exclusive locks this matrix is:

	None	Shared (read)	Exclusive (write)
None	OK	OK	OK
Shared (read)	OK	OK	Conflict
Exclusive (write)	OK	Conflict	Conflict

Just as different granularities bring a need for escalation, different modes bring a need for 'lock conversion', which upgrades a lock to a higher mode, for instance from shared to exclusive, or downgrades it to a lower mode.

Explicit scheduling

In simple situations, queuing for locks is an adequate way to schedule threads. When things are more complicated, however, it's necessary to program the scheduling explicitly because the simple first-come first-served queuing of a lock isn't what you want. A set of printers with different properties, for example, can be optimized across a set of jobs with different priorities, requirements for paper handling, paper sizes, color, etc. There have been many unsuccessful attempts to build general resource allocation systems to handle these problems. They fail because they are too complicated and expensive for simple cases, and not flexible enough for complicated ones. A better strategy is to program the scheduling as part of the application, using as many condition variables as necessary to queue threads that are waiting for resources. Application-specific data structures can keep track of the various resource demands and application-specific code, perhaps written on top of a library, can do the optimization.

Just as you must choose the granularity of locks, you must also choose the granularity of conditions. With just a few conditions (in the limit, only one), it's easy to figure out which one to wait on and which ones to signal. The price you pay is that a thread (or many threads) may wake up from a `wait` only to find that it has to wait again, and this is inefficient. On the other hand, with many conditions you can make useless wakeups very rare, but more care is needed to ensure that a thread doesn't get stuck because its condition isn't signaled.

Simple vs. fancy locks

We have described a number of features that you might want in a locking system:

- multiple modes with conversion, for instance from shared to exclusive;
- multiple granularities with escalation from fine to coarse and de-escalation from coarse to fine;
- deadlock detection.

Database systems typically provide these features. In addition, they acquire locks automatically based on how an application transaction touches data, choosing the mode based on what the operation is, and they can release locks automatically when a transaction commits. For a thorough discussion of database locking see Jim Gray and Andreas Reuter, *Transaction Processing: Concepts and Techniques*, Morgan Kaufmann, 1993, Chapter 8, pages 449-492.

The main reason that database systems have such elaborate locking facilities is that the application programmers are quite naive and can't be expected to understand the subtleties of concurrent

programming. Instead, the system does almost everything automatically, and the programmers can safely assume that execution is sequential. Automatic mechanisms that work well across a wide range of applications need to adapt in the ways listed above.

By contrast, a simple mutex has only one mode (exclusive), only one granularity, and no deadlock detection. If these features are needed, the programmer has to provide them using the mutex and condition primitives. We will study one example of this in detail in handout 17 on formal concurrency: building a reader/writer lock from a simple mutex. Many others are possible.

Summary of easy concurrency

There are four simple steps:

1. Protect each shared data item with a lock, and acquire the lock before touching the data.
2. Write down the invariant which holds on shared data when a lock isn't held, and don't depend on any property of the shared data unless it follows from the invariant. Establish the invariant before releasing the lock.
3. If you have to wait for some other thread to do something before you can continue, avoid busy waiting by waiting on a condition; beware of holding any locks when you do this. When you take some action that might allow a waiting thread to continue, signal the proper condition variable.
4. To avoid deadlock, define a partial order on the locks, and acquire a lock only if it is greater in the order than any lock you already hold. To make this work with procedures, annotate a procedure with a pre-condition: the maximum set of locks that are held whenever it's called.

15. Concurrent Disks and Directories

In this handout we give examples of more elaborate concurrent programs:

Code for `Disk.read` using the same kind of caching used in `BufferedDisk` from handout 7 on file systems, but now with concurrent clients.

Code for a directory tree or graph, as discussed in handout 12 on naming, but again with concurrent clients.

Concurrent buffered disk

The `ConcurrentDisk` module below is similar to `BufferedDisk` in handout 7 on file systems; both implement the `Disk` spec. For simplicity, we omit the complications of crashes. As in handout 7, the buffered disk is based on underlying code for `Disk` called `UDisk`, and calls on `UDisk` routines are in bold so you can pick them out easily.

We add a level of indirection so that we can have names (called `B`'s) for the buffers; a `B` is just an integer, and we keep the buffers in a sequence called `bv`. `B` has methods that let us write `b.db` for `bv(b).db` and similarly for other fields.

The `cache` is protected by a mutex `mc`. Each cache buffer is protected by a mutex `b.m`; when this is held, we say that the buffer is *locked*. Each buffer also has a count `users` of the number of `b`'s to the buffer that are outstanding. This count is also protected by `mc`. It plays the role of a readers lock on the cache reference to the buffer during a disk transfer: if it's non-zero, it is not OK to reassign the buffer to a different disk page. `GetBufs` increments `users`, and `InstallData` decrements it. No one waits explicitly for this lock. Instead, `read` just waits on the condition `moreSpace` for more space to become available.

Thus there are three levels of locking, allowing successively more concurrency and held for longer times:

`mc` is global, but is held only during pointer manipulations;

`b.m` is per buffer, but exclusive, and is held during data transfers;

`b.users` is per buffer and shared; it keeps the assignment of a buffer to a `DA` from changing.

There are three design criteria for the code:

1. Don't hold `mc` during an expensive operation (a disk access or a block copy).
2. Don't deadlock.
3. Handle additional threads that want to read a block being read from the disk.

You can check by inspection that the first is satisfied. As you know, the simple way to ensure the second is to define a partial order on the locks, and check that you only acquire a lock when it is greater than one you already have. In this case the order is `mc` < every `b.m`. The `users` count takes care of the third.

The loop in `read` calls `GetBufs` to get space for blocks that have to be read from the disk (this work was done by `MakeCacheSpace` in handout 7). `GetBufs` may not find enough free buffers, in which case it returns an empty set to `read`, which waits on `moreSpace`. This condition is signaled by the demon thread `FlushBuf`. A real system would have signaling in the other direction too, from `GetBufs` to `FlushBuf`, to trigger flushing when the number of clean buffers drops below some threshold.

The boxes in `ConcurrentDisk` highlight places where it differs from `BufferedDisk`. These are only highlights, however, since the code differs in many details.

`CLASS ConcurrentDisk` `EXPORT read, write, size, check, sync =`

```
TYPE
    % Data, DA, DB, Blocks, Dsk, E as in Disk
    I           =  Int
    J           =  Int

    Buf         =  [db, m, users: I, clean: Bool]     % m protects db, mc the rest
    M           =  Mutex
    B           =  Int WITH {m    :=(\b|bv(b).m),     % index in bv
                             db   :=(\b|bv(b).db),
                             users:=(\b|bv(b).users),
                             clean:=(\b|bv(b).clean)}
    BS          =  SET B

CONST
    DBSize      := Disk.DBSize
    nBufs       := 100
    minDiskRead := 5                                  % wait for this many Bufs

VAR
    % uses UDisk's disk, so there's no state for that
    udisk       :  Disk
    cache       := (DA -> B){}                        % protected by mc
    mc          :  M                                  % protects cache, users
    moreSpace   :  Condition.C                        % wait for more space
    bv          :  (B -> Buf)                         % see Buf for protection
    flushing    :  (DA + Null) := nil                 % only for the AF

% ABSTRACTION FUNCTION Disk.disk(0) = (\ da |
    (    cache!da /\ (cache(da).m not held \/ da = flushing) => cache(da).db
     [*] UDisk.disk(0)(da) ))
```

The following invariants capture the reasons why this code works. They are not strong enough for a formal proof of correctness.

```
% INVARIANT 1: ( ALL da :IN cache.dom, b |
    b = cache(da) /\ b.m not held /\ b.clean ==> b.db = UDisk.disk(0)(da) )
```

A buffer in the cache, not locked, and clean agrees with the disk (if it's locked, the code in `FlushBuf` and the caller of `GetBufs` is responsible for keeping track of whether it agrees with the disk).

```
% INVARIANT 2: (ALL b | {da | cache!da /\ cache(da) = b}.size <= 1)
```

A buffer is in the cache at most once.

```
% INVARIANT 3: mc not held ==> (ALL b :IN bv.dom | b.clean /\ b.users = 0
                                                  ==> b.m not held)
```

If `mc` is not held, a clean buffer with `users = 0` isn't locked.


```
PROC new(size: Int) -> Disk =
    self := StdNew(); udisk := udisk.new(size);
    mc.acq; DO VAR b | ~ bv!b => VAR m := m.new() |
        bv(b) := Buf{m := m, db := {}, users := 0, clean := true}
    OD; mc.rel
    RET self

PROC read(e) -> Data RAISES {notThere} =
    udisk.check(e);
    VAR data := Data{}, da := e.da, upto := da + e.size, i |
        mc.acq;
        % Note that we release mc before a slow operation (bold below)
        % and reacquire it afterward.
        DO da < upTo => VAR b, bs |                    % read all the blocks
            IF  cache!da =>
                b := cache(da);                        % yes, in buffer b; copy it
                % Must increment users before releasing mc.
                bv(b).users + := 1; mc.rel;
                % Now acquire m before copying the data.
                % May have to wait for m if the block is being read.
                b.m.acq; data + := b.db; b.m.rel;
                mc.acq; bv(b).users - := 1;
                da := da + 1
             [*] i := RunNotInCache(da, upTo);         % da not in the cache
                bs := GetBufs(da, i); i := bs.size;    % GetBufs is fast
                IF  i > 0 =>
                    mc.rel; data + := InstallData(da, i); mc.acq;
                    da + := i
                [*] moreSpace.wait(mc)
                FI
            FI
        OD; mc.rel; RET data

FUNC RunNotInCache(da, upTo: DA) -> I =                % mc locked
    RET {i | da + i <= upTo /\ (ALL j :IN i.seq | ~ cache!(da + j)).max
```

`GetBufs` tries to return `i` buffers, but it returns at least `minDiskRead` buffers (unless `i` is less than this) so that `read` won't do lots of tiny disk transfers. It's tempting to make `GetBufs` always succeed, but this means that it must do a `Wait` if there's not enough space. While `mc` is released in the `Wait`, the state of the cache can change so that we no longer want to read `i` pages. So the choice of `i` must be made again after the `Wait`, and it's most natural to do the `Wait` in `read`.

If `users` and `clean` were protected by `m` (as `db` is) rather than by `mc`, `GetBufs` would have to acquire pages one at a time, since it would have to acquire the `m` to check the other fields. If it couldn't find enough pages, it would have to back out. This would be both slow and clumsy.

```
PROC GetBufs(da, i) -> BS =
% mc locked. Return some buffers assigned to da, da+1, ..., locked, and
% with users = 1, or {} if there's not enough space. No slow operations.
  VAR bs := {b | b.users = 0 /\ b.clean} |             % the usable buffers
    IF  bs.size >= {i, minDiskRead}.min =>             % check for enough buffers
        i := {i, bs.size}.min;
        DO VAR b | b IN bs /\ b.users = 0 =>
            % Remove the buffer from the cache if it's there.
            IF VAR da' | cache(da') = b => cache := cache{da' -> } [*] SKIP FI;
            b.m.acq; bv(b).users := 1; cache(da) := b; da + := 1
        OD; RET {b :IN bs | b.users > 0}
    [*] RET {}                                         % too few; caller must wait
```

```
    FI
```

In handout 7, `InstallData` is done inline in `read`.

```
PROC InstallData(da, i) = VAR data, j := 0 |
% Pre: cache(da) .. cache(da+i-1) locked by SELF with users > 0.
    data := udisk.read(E{da, i});
    DO j < i => VAR b := cache(da + j) |
        bv(b).db := udisk.DToB(data).sub(j); b.m.rel;
        mc.acq; bv(b).users - := 1; mc.rel;
        j + := 1
    OD; RET data
```

`PROC write` is omitted. It sets `clean` to `false` for each block it writes. The background thread `FlushBuf` does the writes to disk. Here is a simplified version that does not preserve write order. Note that, like `read`, it releases `mc` during a slow operation.

```
THREAD FlushBuf() = DO                                  % flush a dirty buffer
    mc.acq;
    IF  VAR da, b | b = cache(da) /\ b.users = 0 /\ ~ b.clean =>
        flushing := true;                               % just for the AF
        b.m.acq; bv(b).clean := true; mc.rel;
        udisk.write(da, b.db);
        flushing := false;
        b.m.rel; moreSpace.signal
     [*] mc.rel
    OD

% Other procedures omitted

END ConcurrentDisk
```

Concurrent directory operations

In handout 12 on naming we gave an `ObjNames` spec for looking up path names in a tree of graph of directories. Here are the types and state from `ObjNames`:

```
TYPE D          =  Int                                  % Just an internal name
                   WITH {get:=GetFromS, set:=SetInS}    % get returns nil if undefined
    Link        =  [d: (D + Null), pn]                  % d=nil means the containing D
    Z           =  (V + D + Link + Null)                % nil means undefined
    DD          =  N -> Z

CONST
    root        :  D := 0
    s           := (D -> DD){}{root -> DD{}}            % initially empty root

APROC GetFromS(d, n) -> Z =                             % d.get(n)
    << RET s(d)(n) [*] RET nil >>

APROC SetInS  (d, n, z)   =                             % d.set(n, z)
% If z = nil, SetInS leaves n undefined in s(d).
    << IF z # nil => s(d)(n) := z [*] s(d) := s(d){n -> } FI >>
```

We wrote the spec to allow the bindings of names to change during lookups, but it never reuses a `D` value or an entry in `s`. If it did, a lookup of `/a/b` might obtain the `D` for `/a`, say `dA`, and then `/a` might be deleted and `dA` reused for some entirely different directory. When the lookup continues it will look for `b` in that directory. This is definitely not what we have in mind.

Code, however, will represent a `DD` by some data structure on disk (and cached in RAM), and if the directory is deleted it will reuse the space. This code needs to prevent the anomalous behavior we just described. The simplest way to do so is similar to the `users` device in `ConcurrentDisk` above: a shared lock that prevents the directory data structure from being deleted or reused.

The situation is trickier here, however. It's necessary to make sufficiently atomic the steps of first looking up `a` to obtain `dA`, and then incrementing `s(dA).users`. To do this, we make `users` a true readers lock, which prevents changes to its directory. In particular, it prevents an entry from being deleted or renamed, and thus prevents a subdirectory from being deleted. Then it's sufficient to hold the lock on `dA`, look up `b` to obtain `dB`, and acquire the lock on `dB` before releasing the lock on `dA`. This is called 'lock coupling'.

As we saw in handout 12, the amount of concurrency allowed there makes it possible for lookups done during renames to produce strange results. For example, `Read(/a/x)` can return `3` even though there was never any instant at which the path name `/a/x` had the value `3`, or indeed was defined at all. We copy the scenario from handout 12. Suppose:

initially `/a` is the directory `d1` and `/b` is undefined;

initially `x` is undefined in `d1`;

concurrently with `Read(/a/x)` we do `Rename(/a, /b); Write(/b/x, 3).`

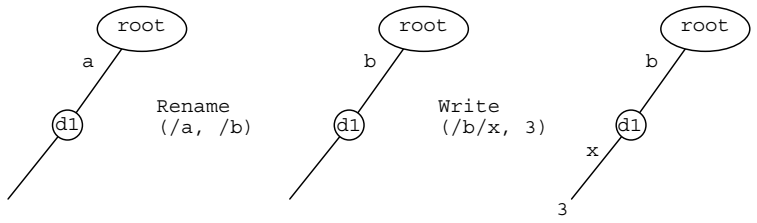
The following sequence of actions yields `Read(/a/x) = 3`:

In the `Read`, `Get(root, a) = d1`

`Rename(/a, /b)` makes `/a` undefined and `d1` the value of `/b`

`Write(/b/x, 3)` makes `3` the value of `x` in `d1`

In the `Read`, `RET d1.get(x)` returns `3`.



Obviously, whether this possibility is important or not depends on how clients are using the name space.

To avoid this kind of anomaly, it's necessary to hold a read lock on every directory on the path. When the directory graph is cyclic, code that acquires each lock in turn can deadlock. To avoid this deadlock, it's necessary to write more complicated code. Here is the idea.

Define some arbitrary ordering on the directory locks (say based on the numeric value of `D`). When doing a lookup, if you need to acquire a lock that is less than the biggest one you hold, release the bigger locks, acquire the new one, and then repeat the lookup from the point of the first

released lock to reacquire the released locks and check that nothing has changed. This may happen repeatedly as you look up the path name.

This can be made more efficient (and more complicated, alas) with a 'tentative' `Acquire` that returns a failure indication rather than waiting if it can't acquire the lock. Then it's only necessary to backtrack when another thread is actually holding a conflicting write lock.

16. Paper: Programming with Threads

The attached paper by Andrew Birrell, *Introduction to Programming with Threads*, originally appeared as report 35 of the Systems Research Center, Digital Equipment Corp., Jan. 1989. A somewhat revised version appears as chapter 4 of *Systems Programming with Modula-3*, Greg Nelson ed., Prentice-Hall, 1991, pp 88-118. The current version has been revised to use the C# language (similar to Java) and incorporate some new material.

Read it as an adjunct to the lecture on practical concurrency. It explains how to program with threads, mutexes, and condition variables, and it contains a lot of good advice and examples.

.

17. Formal Concurrency

In this handout we take a more formal view of concurrency than in handout 14 on practical concurrency. Our goal is to be able to prove that a general concurrent program implements a spec.

We begin with a fairly precise account of the non-atomic semantics of Spec, though our treatment is less formal than the one for atomic semantics in handout 9. Next we explain the general method for making large atomic actions out of small ones (easy concurrency) and prove its correctness. We continue with a number of examples of concurrency, both easy and hard: mutexes, condition variables, read-write locks, buffers, and non-atomic clocks. Finally, we give fairly careful proofs of correctness for some of the examples.

Non-atomic semantics of Spec

We have already seen that a Spec module is a way of defining an automaton, or state machine, with transitions corresponding to the invocations of external atomic procedures. This view is sufficient if we only have functions and atomic procedures, but when we consider concurrency we need to extend it to include internal transitions. To properly model crashes, we introduced the idea of atomic commands that may not be interrupted. We did this informally, however, and since a crash "kills" any active procedure, we did not have to describe the possible behaviors when two or more procedures are invoked and running concurrently. This section describes the concurrent semantics of Spec.

The most general way to describe a concurrent system is as a collection of independent atomic actions that share a collection of variables. If the actions are `A1, ..., An` then the entire system is just the 'or' of all these actions: `A1 [] ... [] An`. In general only some of the actions will be enabled, but for each transition the system non-deterministically chooses an action to execute from all the enabled actions. Thus non-determinism encompasses concurrency.

Usually, however, we find it convenient to carry over into the concurrent world as much of the framework of sequential computing as possible. To this end, we model the computation as a set of *threads* (also called 'tasks' or 'processes'), each of which executes a sequence of atomic actions; we denote threads by variables `h`, `h'`, etc. To define its sequence, each thread has a state variable called its 'program counter' `$pc`, and each of its actions has the form `(h.$pc = α) => c`, so that `c` can only execute when `h`'s program counter equals `α`. Different actions have different values for `α`, so that at most one action of a thread is enabled at a time. Each action advances the program counter with an assignment of the form `h.$pc := β`, thus enabling the thread's next action.

It's important to understand there is nothing truly fundamental about threads, that is, about organizing the state transitions into sets such that at most one action is enabled in each set. We do so because we can then carry forward much of what we know about sequential computing into the concurrent world. In fact, we want to achieve our performance goals with as little concurrency as possible, since concurrency is confusing and error-prone.

We now explain how to use this view to understand the non-atomic semantics of Spec.

Non-atomic commands and threads

Unlike an atomic command, a non-atomic command cannot be described simply as a relation between states and outcomes, that is, an atomic transition. The simple example, given in handout 14, of a non-atomic assignment `x := x + 1` executed by two threads should make this clear: the outcome can increase `x` by `1` or `2`, depending on the interleaving of the transitions in the two threads. Rather, a non-atomic command corresponds to a *sequence* of atomic transitions, which may be interleaved with the sequences of other commands executing concurrently. To describe this interleaving, we use *labels* and *program counters*. We also need to distinguish the various *threads* of concurrent computation.

Intuitively, threads represent sequential processes. Roughly speaking, each point in the program between two atomic commands is assigned a label. Each thread's program counter `$pc` takes a label as its value,[1] indicating where that thread is in the program, that is, what command it is going to execute next.

Spec threads are created by top level `THREAD` declarations in a module. They make all possible concurrency explicit at the top level of each module. A thread is syntactically much like a procedure, but instead of being invoked by a client or by another procedure, it is automatically invoked in parallel initially, for every possible value of its arguments.[2] When it executes a `RET` (or reaches the end of its body), a thread simply makes no more transitions. However, threads are often written to loop indefinitely.

Spec does not have `COBEGIN` or `FORK` constructs, as many programming languages do, these are considerably more difficult to define precisely, since they are tangled up with the control structure of the program. Also, because one Spec thread starts up for every possible argument of the `THREAD` declaration, they tend to be more convenient for most of the specs and code in this course. To keep the thread from doing anything until a certain point in the computation (or at all), use an initial guard for the entire body as in the `Sieve` example below.

A thread is named by the name in the declaration and the argument values. Thus, the threads declared by `THREAD Foo(bool) = ...`, for example, would be named `Foo(true)` and `Foo(false)` The names of local variables are qualified by both the name of the thread that is the root of the call stack, and by the name of the procedure invoked.[3] In other words, each procedure in each thread has its own set of local variables. So for example, the local variable `p` in the `Sieve` example appears in the state as `Sieve(0).p, Sieve(1).p, ...`. If there were a `PROC Foo` called by `Sieve` with a local variable `baz`, the state might be defined at `Sieve(0).Foo.baz, Sieve(1).Foo.baz, ...`. The pseudo-names `$a`, `$x`, and `$pc` are qualified only by the thread.

Each atomic command defines a transition, just as in the sequential semantics. However, now a transition is enabled by the program counter value. That is, a transition can only occur if the program counter of some thread equals the label before the command, and the transition sets the program counter of that thread to the label after the command. If the command at the label in the program counter fails (for example, because it has a guard that tests for a buffer to be non-empty,

[1] The variables declared by a program are not allowed to have labels as their values, hence there is no `Label` type.
[2] This differs from the threads in Java, in Modula 3, or in many C implementations. These languages start a computation with one thread and allow it to create and destroy threads dynamically using `fork` and `join` operations.
[3] This works for non-recursive procedures. To accommodate recursive procedures, the state must involve something equivalent to a stack. Probably the simplest solution is to augment the state space by tacking on the nesting depth of the procedure to all the names and program counter values defined above. For example, `h + ".P.v"` becomes `h + ".P.v" + d.enc`, for every positive integer `d`. An invocation transition at depth `d` goes to depth `d+1`.

and the buffer is empty in the current state), the thread is "stuck" and does not make any transitions. However, it may become unstuck later, because of the transitions of some other threads. Thus, a command failing does not necessarily (or even usually) mean that the thread fails.

We won't give the non-atomic semantics precisely here as we did for the atomic semantics in handout 9, since it involves a number of fussy details that don't add much insight. Also, it's somewhat arbitrary. You can always get exactly the atomicity you want by adding local variables and semicolons to increase the number of atomic transitions (see the examples below), or `<<...>>` brackets to decrease it.

It's important, however, to understand the basic ideas.

- Each atomic command in a thread or procedure defines a transition (atomic procedures and functions are taken care of by the atomic semantics).
- The program counters enable transitions: a transition can only occur if the program counter for some thread equals the label before the command, and the transition sets that program counter to the label after the command.

Thus the state of the system is the global state plus the state of all the threads. The state of a thread is its `$pc`, `$a`, and `$x` values, the local variables of the thread, and the local variables of each procedure that has been called by the thread and has not yet returned.

Suppose the label before the command `c` is α and the one after the command is `β`, and the transition function defined by `MC(c)` in handout 9 is `(\ s, o | rel)`. Then if `c` is in thread `h`, its transition function is

```
(\ s, o | s(h+".$pc") = α /\ o(h+".$pc) = β /\ rel')
```

If `c` is in procedure `P`, that is, `c` can execute for any thread whose program counter has reached α, its transition function is

```
(\ s, o | (EXISTS h: Thread |
    s(h+".P.$pc") = α /\ o(h+".P.$pc) = β /\ rel'))
```

Here `rel'` is `rel` with each reference to a local variable `v` changed to `h + ".v"` or `h + ".P.v"`.

Here are some examples of a non-atomic program translated into the non-deterministic form. The first one is very simple:

```
[α1] C1 ; [α2] C2 [α3]          pc= α1 => << C1;          pc:= α2 >>
                              [] pc= α2 => << C2;          pc:= α3 >>
                              [] pc= α3 => ...
```

As you can see, `[α] C; [β]` translates to `pc=α => << C; pc:=β >>`. If there's a non-deterministic choice, that shows up as a choice in one of the actions of the translation:

```
[α1] IF C1 ; [α2] C2 [] C3 FI [α3]   pc= α1 => IF << C1;          pc:= α2 >>
                                                [] << C3;          pc:=α3 >>
                                                FI
                              [] pc= α2 => << C2;          pc:= α3 >>
                              [] pc= α3 => ...
```

You might find it helpful to write out the state transition diagram for this program.

The second example is a simple real Spec program. The next section explains the detailed rules for where the labels go in Spec; note that an assignment is two atomic actions, one to evaluate the expression on the right and the other to change the variable on the left. The extra local variable `t` is a place to store the value between these two actions.

```
                                        VAR t |
[α1] DO i < n =>                           << pc=α1 /\ i < n => pc:=α2 [*] pc:= α6 >>
    [α2] sum := [α3] sum + x(i);        []<< pc= α2 => t := sum + x(i);     pc:= α3 >>
                                        []<< pc= α3 => sum := t;            pc:= α4 >>
    [α4] i := [α5] i + 1                []<< pc= α4 => t := i + 1;          pc:= α5 >>
                                        []<< pc= α5 => i := 1;              pc:= α1 >>
OD [α6]                                 []<< pc= α6 => ...
```

Labels in Spec

What are the atomic transitions in a Spec program? In other words, where do we put the labels? The basic idea is to build in as little atomicity as possible (since you can always put in what you need with `<<...>>`). However, expression evaluation must be atomic, or reasoning about expressions would be a mess. To use Spec to model code in which expression evaluation is not atomic (C code, for example), you must add temporary variables. Thus `x := a + b + c` becomes

```
VAR t | << t := a >>; << t := t + b >>; << x := t + c >>
```

For a real-life example of this, see `MutexImpl.acq` below.

The simple commands, `SKIP`, `HAVOC`, `RET`, and `RAISE`, are atomic, as is any command in atomicity brackets `<<...>>`.

For an invocation, there is a transition to evaluate the argument and set the `$a` variable, and one to send control to the start of the body. The `RET` command's transition sets `$a` and leaves control at the end of the body. The next transition leaves control after the invocation. So every procedure invocation has at least four transitions: evaluate the argument and set `$a`, send control to the body, do the `RET` and set `$a`, and return control from the body. The reason for these fussy details is to ensure that the invocation of an external procedure has start and end transitions that do not change any other state. These are the transitions that appear in the trace and therefore must be identical in both the spec and the code that implements it.

Minimizing atomicity means that an assignment is broken into separate transitions, one to evaluate the right hand side and one to change the left hand variable. This also has the advantage of consistency with the case where the right hand side is a non-atomic procedure invocation. Each transition happens atomically, even if the variable is "big". Thus `x := exp` is

```
VAR t | << t := exp >> ; << x := t >>
```

and `x := p(y)` is

```
p(y); << x := $a >>
```

Since there are no additional labels for the `c1 [] c2` command, the initial transition of the compound command is enabled exactly when the initial transition of either of the subcommands is enabled (or if they both are). Thus, the choice is made based only on the first transition. After that, the thread may get stuck in one branch (though, of course, some other thread might unstick it later). The same is true for `[*]`, except that the initial transition for `c1 [*] c2` can only be the initial transition of `c2` if the initial transition of `c1` is not enabled. And the same is also true for `VAR`. The value chosen for `id` in `VAR id | c` must allow `c` to make at least one transition; after that the thread might get stuck.

`DO` has a label, but `OD` does not introduce any additional labels. The starting and ending program counter value for `c` in `DO c OD` is the label on the `DO`. Thus, the initial transition of `c` is enabled when the program counter is the label on the `DO`, and the last transition sets the program counter back to that label. When `c` fails, the program counter is set to the label following the `OD`.

To sum up, there's a label on each `:=`, `=>`, '`;`', `EXCEPT`, and `DO` outside of `<<...>>`. There is never any label inside atomicity brackets. It's convenient to write the labels in brackets after these symbols.

There's also a label at the start of a procedure, which we write on the `=` of the declaration, and a label at the end. There is one label for a procedure invocation, after the argument is evaluated; we write it just before the closing '`)`'. After the invocation is complete, the PC goes to the next label after the invocation, which is the one on the `:=` if the invocation is in an assignment.

As a consequence of this labeling, as we said before, a procedure invocation has
one transition to evaluate the argument expression,
one to set the program counter to the label immediately before the procedure body,
one for every transition of the procedure body (using the labels of the body),
one for the `RET` command in the body, which sets the program counter after the body,
and a final transition that sets it to the label immediately following the invocation.

Here is a meaningless sequential example, just to show where the labels go. They are numbered in the order they are reached during execution.

```
PROC P() = [P1] VAR x, y |
    IF x > 5 => [P2] x := [P4] Q(x + 1, 2 [P3]); [P5] y := [P6] 3
    [] << y := 4 >>
    FI; [P7]
    VAR z | DO [P8] << P() >> OD [P9]
```

External actions

In sequential Spec a module has only external actions; each invocation of a function or atomic procedure is an external action. In concurrent Spec there are two differences:

There are internal actions. These can be actions of an externally invoked `PROC` or actions of a thread declared and executing in the module.

There are two external actions in the external invocation of a (non-atomic) procedure: the call, which sends control from after evaluation of the argument to the entry point of the procedure, and the return, which sends control from after the `RET` command in the procedure to just after the invocation in the caller. These external transitions do not affect the `$a` variable that communicates the argument and result values. That variable is set by the internal transitions that compute the argument and do the `RET` command.

There's another style of defining external interfaces in which every external action is an `APROC`. If you want to get the effect of a non-atomic procedure, you have to break it into two `APROC`'s, one that delivers the arguments and sets the internal state so that internal actions will do the work of the procedure body, and a second that retrieves the result. This style is used in I/O automata[4], but we will not use it.

Examples

Here are two Spec programs that search for prime numbers and collect the result in a set `primes`; both omit the even numbers, initializing `primes` to `{2}`. Both are based on the *sieve of Eratos-*

[4] Nancy Lynch, *Distributed Algorithms*, Morgan Kaufmann, 1996, Chapter 8.

thenes, testing each prime less than $n^{1/2}$ to see whether it divides *n*. Since the threads may not be synchronized, we must ensure that all the numbers $\le n^{1/2}$ have been checked before we check *n*.

The first example is more like a spec, using an infinite number of threads, one for every odd number.

```
CONST Odds       =  {i: Nat | i // 2 = 1 /\ i > 1 }

VAR  primes      :  SET Nat := {2}
     done        :  SET Nat := {}                          % numbers checked

INVARIANT (ALL n: Nat |   n IN done /\ IsPrime(n) ==> n IN primes
                        /\ n IN primes ==> IsPrime(n))

THREAD Sieve1(n :IN Odds) =
      {i :IN Odds | i <= Sqrt(n)} <= done =>               % Wait for possible factors
         IF   (ALL p :IN primes | p <= Sqrt(n) ==> n // p # 0) =>
              << primes \/ := {n} >>
         [*]  SKIP
         FI;
         << done \/ := {n} >>                              % No more transitions

FUNC Sqrt(n: Nat) -> Int = RET { i: Nat | i*i <= n }.max
```

The second example, on the other hand, is closer to code, running ten parallel searches. Although there is one thread for every integer, only threads `Sieve(0)`, `Sieve(1)`, …, `Sieve(9)` are "active", because of the initial guard, Differences from `Sieve1` are boxed.

```
CONST nThreads   := 10

VAR  primes      :  SET Int := {2}
     next        := nThreads.seq

THREAD Sieve(i: Int) = next!i =>
     next(i) := 2*i + 3;
     DO VAR n: Int  := next(i) |
          (ALL j :IN next.rng | j >= Sqrt(n)) =>
               IF   (ALL p :IN primes | p <= Sqrt(n) ==> n // p # 0) =>
                    << primes \/ := {n} >>
               [*]  SKIP
               FI;
               next(i) := n + 2*nThreads
     OD
```

Big atomic actions

As we saw earlier, we need atomic actions for practical, easy concurrency. Spec lets you specify any grain of atomicity in the program just by writing << … >> brackets.[5] It doesn't tell you where to write the brackets. If the environment in which the program has to run doesn't impose any constraints, it's usually best to make the atomic actions as big as possible, because big atomic actions are easier to reason about. But big atomic actions are often too hard or too expensive to code, or the reduction in concurrency hurts performance too much, so that we have to make do with small ones. For example, in a shared-memory multiprocessor typically only the individual

[5] As we have seen, Spec does treat expression evaluation as atomic. Recall that if you are dealing with an environment in which an expression like `x(i) + f(y)` can't be evaluated atomically, you should model this by writing `VAR t1, t2 | t1 := x(i); t2 := f(y); ... t1 + t2 ....`

instructions are atomic, and we can only write one disk block atomically. So we are faced with the problem of showing that code with small atomic actions satisfies a spec with bigger ones.

The idea

The standard way of doing this is by some kind of 'non-interference'. The idea is based on the following observation. Suppose we have a program with a thread `h` that contains the sequence

```
A; B                                                    (1)
```

as well as an arbitrary collection of other commands. We denote the program counter value before `A` by α and at the semi-colon by β. We are thinking of the program as

```
h.$pc = α => A [] h.$pc = β => B [] C1 [] C2 [] ...
```

where each command has an appropriate guard that enables it only when the program counter for its thread has the right value. We have written the guards for `A` and `B` explicitly.

Suppose `B` denotes an arbitrary atomic command, and `A` denotes an atomic command that commutes with every command in the program (other than `B`) that is enabled when `h` is at the semicolon, that is, when `h.$pc = β`. (We give a precise meaning for 'commutes' below.) In addition, both `A` and `B` have only internal actions. Then it's intuitively clear that the program with (1) simulates a program with the same commands except that instead of (1) it has

```
<< A; B >>                                              (2)
```

Informally this is true because any `C`'s that happen between `A` and `B` have the same effect on the state that they would have if they happened before `A`, since they all commute with `A`. Note that the `C`'s don't have to commute with `B`; commuting with `A` is enough to let us 'push' `C` before `A`. A symmetric argument works if all the `C`'s commute with `B`, even if they don't commute with `A`.

Thus we have achieved the goal of making a bigger atomic command `<< A; B >>` out of two small ones `A` and `B`. We can call the big command `D` and repeat the process on `E; D` to get a still bigger command `<< E; A; B >>`.

How do we ensure that only a command `C` that commutes with `A` can execute while `h.$pc = β`? The simplest way is to ensure that the variables that `A` touches (reads or writes) are disjoint from the variables that `C` writes, and vice versa; then they will surely commute. Two such commands are called 'non-interfering'. There are two easy ways to show that commands are non-interfering. One is that `A` touches only local variables of `h`. Only actions of `h` touch local variables of `h`, and the only action of `h` that is enabled when `h.$pc = β` is `B`. So any sequence of commands that touch only local variables is atomic, and if it is preceded or followed by a single arbitrary atomic command the whole thing is still atomic.[6]

The other easy case is a critical section protected by a mutex. Recall that a critical section for `v` is a command with the property that if some thread's `PC` is in the command, then no other thread's `PC` can be in any critical section for `v`. If the only commands that touch `v` are in critical sections for `v`, then we know that only `B` and commands that don't touch `v` can execute while `h.$pc = β`. So if every command in any critical section for `v` only touches `v` (and perhaps local variables), then the program simulates another program in which every critical section is an atomic command. A critical section is usually coded by acquiring a *lock* or *mutex* and holding it for the duration of the section. The property of a lock is that it's not possible to acquire it when it is already held, and this ensures the mutual exclusion property for critical sections.

[6] See Leslie Lamport and Fred B. Schneider. Pretending atomicity. Research Report 44, Digital Equipment Corporation Systems Research Center, Palo Alto, CA, May 1989. http://gatekeeper.dec.com/pub/DEC/SRC/research-reports/abstracts/src-rr-044.html

It's not necessary to have exclusive locks; reader/writer locks are sufficient for non-interference, because read operations all commute with each other. Indeed, any locking scheme will work in which non-commuting operations hold mutually exclusive locks; this is the basis of rules for 'lock conflicts'. See handout 14 on practical concurrency for more details on different kinds of locks.

Another important case is mutex acquire and release operations. These operations only touch the mutex, so they commute with everything else. What about these operations on the same mutex in different threads? If both can execute, they certainly don't yield the same result in either order; that is, they don't commute. When can both operations execute? We have the following cases (writing the executing thread as an explicit argument of each operation):

A	C	Possible sequence?
`m.acq(h)`	`m.acq(h')`	No: `C` is blocked by `h` holding `m`
`m.acq(h)`	`m.rel(h')`	No: `C` won't be reached because `h'` doesn't hold `m`
`m.rel(h)`	`m.acq(h')`	OK
`m.rel(h)`	`m.rel(h')`	No: one thread doesn't hold `m`, hence won't do `rel`

So `m.acq` commutes with everything that's enabled at β, since neither mutex operation is enabled at β in a program that avoids havoc. But `m.rel(h)` doesn't commute with `m.acq(h')`. The reason is that the `A; C` sequence can happen, but the `C; A` sequence `m.acq(h'); m.rel(h)` cannot, because in this case `h` doesn't hold `m` and therefore can't be doing a `rel`. Hence it's not possible to flip every `C` in front of `m.rel(h)` in order to make `A; B` atomic.

What does this mean? You can acquire more locks and still keep things atomic, but as soon as you release one, you no longer have atomicity.[7]

A third important case of commuting operations, producer-consumer, is similar to the mutex case. A producer and a consumer thread share no state except a buffer. The operations on the buffer are `put` and `get`, and these operations commute with each other. The interesting case is when the buffer is empty. In this case `get` is blocked until a `put` occurs, just as in the mutex example when `h` holds the lock `m.acq(h')` is blocked until `m.rel(h)` occurs. This is why programming with buffers, or dataflow programming, is so easy.

Proofs

How can we make all this precise and *prove* that a program containing (1) implements the same program with (2) replacing (1), using our trusty method of abstraction relations? For easy reference, we repeat (1) and (2).

```
A; [β] B                (1)
<< A; B >>              (2)
```

As usual, we call the complete program containing (2) the spec *S* and the complete program containing (1) the code *T*. We need an abstraction relation `AR` between the states of *T* and the states of *S* under which every transition of *T* simulates a (possibly empty) trace of *S*. Note that the state spaces of *T* and *S* are the same, except that `h.$pc` can never be β in *S*. We use *s* and *u* for states of *S* and *T*, to avoid confusion with various other uses of *t*.

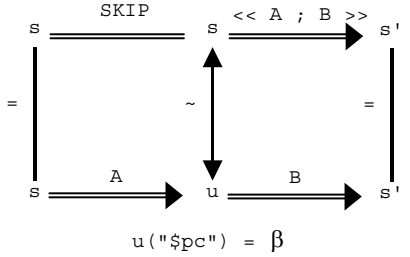
[7] Actually, this is not quite right. If you hold several locks, and touch data only when you hold its lock, you have atomicity until you release all the locks.

First we need a precise definition of "`C` is enabled at β and commutes with `A`". For any command `X`, we write `u X u'` for `MC(X)(u, u')`, that is, if `X` relates `u` to `u'`. The idea of 'commutes' is that `<<A; C>>`, which is a relation between states, is a *superset* of the relation, `<<C; A>>`, and the definition follows from the meaning of semicolon:

```
(ALL u1, u2 |   (EXISTS u  | u1 A u  /\ u  C u2 /\ u("h.$pc") = β)
            ==> (EXISTS u' | u1 C u' /\ u' A u2) )
```

This says that any result that you could get by doing `A; C` you could also get by doing `C; A`. Note that it's OK for `C; A` to have more transitions, since we want to show that `A; C; B` implements `C; << A; B >>`, not that they are equivalent. This is not just nit-picking; if `C` acquires a lock that `A` holds, there is no transition from `A` to `C` in the first case.

It seems reasonable to do the proof by making `A` simulate the empty trace and `B` simulate `<<A; B>>`, since we know more about `A` than about `B`; every other command simulates itself.

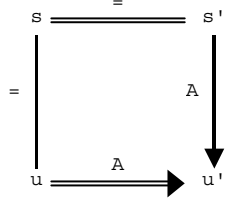


So we make `AR` the identity everywhere except at β, where it relates any state `u` that can be reached from `s` by `A` to `s`. This expresses the intention that at β we haven't yet done `A` in *S*, but we have done `A` in *T*. (Since `A` may take many states to `s`, this can't just be an abstraction function.) We write `u ~ s` for "`AR` relates `u` to `s`". Precisely, we say that `u ~ s` if

```
    u("h.$pc") ≠ β /\ s = u
\/ u("h.$pc") = β /\ s A u.
```

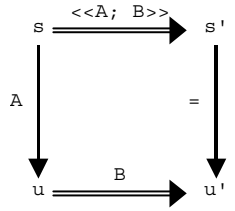
Why is this an abstraction relation? It certainly relates an initial state to an initial state, and it certainly works for any transition `u -> u'` that stays away from β, that is, in which `u("h.$pc") ≠ β` and `u'("h.$pc") ≠ β`, since the abstract and concrete states are the same. What about transitions that do involve β?

- If `h.$pc` changes to β then we must have executed `A`. The picture is



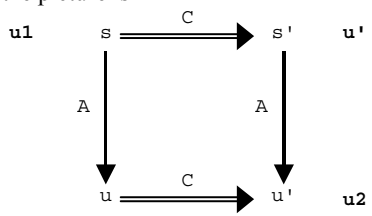
  The abstract trace is empty, so the abstract state doesn't change: `s = s'`. Also, `s' = u` because only equal states are related when `h.$pc # β`. But we executed `A`, so `u A u'`, so `s' ~ u'` because of the equalities.

- If `h.$pc` starts at β then the command must be either `B` or some `C` that commutes with `A`. If the command is `B`, then the picture is



  To show the top relation, we have to show that there exists an `s0` such that `s A s0` and `s0 B s'`, by the meaning of semicolon. But `u` has exactly this property, since `s' = u'`.

- If the command is `C`, then the picture is



  But this follows from the definition of 'commutes': we are given `s`, `u`, and `u'` related as shown, and we need `s'` related as shown, which is just what the definition gives us, with `u1 = s`, `u2 = u'`, and `u' = s'`.

Examples of concurrency

This section contains a number of example specs and codes that illustrate various aspects of concurrency. The specs have large atomic actions that keep them simple. The codes have smaller atomic actions that reflect the realities of the machines on which they have to run. Some of the examples of code illustrate easy concurrency (that is, that use locks): `RWLockImpl` and `BufferImpl`. Others illustrate hard concurrency: `SpinLock`, `Mutex2Impl`, `ClockImpl`, `MutexImpl`, and `ConditionImpl`.

Incrementing a register

The first example involves incrementing a register that has `Read` and `Write` operations. Here is the unsurprising spec of the register, which makes both operations atomic:

```
MODULE Register EXPORT Read, Write =

VAR x           :  Int := 0

APROC Read() -> Int = << RET x  >>
APROC Write(i: Int) = << x := i >>

END Register
```

To increment the register, we could use the following procedure:

```
PROC Increment() = VAR t: Int | t := Register.Read(); t := t + 1; Register.Write(t
```


Suppose that, starting from the initial state where $x = 0$, n threads execute `Increment` in parallel. Then, depending on the interleaving of the low-level steps, the final value of the register could be anything from 1 to n. This is unlikely to be what was intended. Certainly this code doesn't implement the spec

```
PROC Increment() = << x := x + 1 >>
```

Exercise: Suppose that we weaken our atomicity assumptions to say that the value of a register is represented as a sequence of bits, and that the only atomic operations are reading and writing individual bits. Now what are the possible final states if n threads execute `Increment` in parallel?

Alternatively, consider a new module `RWInc` that explicitly supports `Increment` operations in addition to `Read` and `Write`. This might add the following (exported) procedure to the `Register` module:

```
PROC Increment() = x := x+1
```

Or, more explicitly:

```
PROC Increment() =  VAR t: Int | << t := x >>; << x := t+1 >>
```

Because of the fine grain of atomicity, it is still true that if n threads execute `Increment` in parallel then, depending on the interleaving of the low-level steps, the final value of the register could be anything from 1 to n. Putting the procedure inside the `Register` module doesn't help. Of course, making `Increment` an `APROC` would certainly do the trick.

Mutexes

Here is a spec of a simple `Mutex` module, which can be used to ensure mutually exclusive execution of critical sections; it is copied from handout 14 on practical concurrency. The state of a mutex is `nil` if the mutex is free, and otherwise is the thread that holds the mutex.

```
CLASS Mutex EXPORT acq, rel =

VAR m           :  (Thread + Null) := nil
% Each mutex is either nil or the thread holding the mutex.
% The variable SELF is defined to be the thread currently making a transition.

APROC acq() = << m = nil  => m := SELF; RET >>
APROC rel() = << m = SELF => m := nil ; RET [*] HAVOC >>

END Mutex
```

If a thread invokes `acq` when `m` ≠ `nil`, then the body fails, This means that there's no possible transition for that thread, and the thread is blocked, waiting at this point until the guard becomes true. If many threads are blocked at this point, then when `m` is set to `nil`, one is scheduled first, and it sets `m` to itself atomically; the other threads are still blocked.

The spec says that if a thread that doesn't hold `m` does `m.rel`, the result is `HAVOC`. As usual, this means that the code is free to do anything when this happens. As we shall see in the `SpinLock` code below, one possible 'anything' is to free the mutex anyway.

Here is a simple use of a mutex `m` to make the `Increment` procedure atomic:

```
PROC Increment() = VAR t: Int |
    m.acq; t := Register.Read(); t := t + 1; Register.Write(t); m.rel
```

This keeps concurrent calls of `Increment` from interfering with each other. If there are other write accesses to the register, they must also use the mutex to avoid interfering with threads executing `Increment`.

Spin locks

A simple way to code a mutex is to use a *spin lock*. The name is derived from the behavior of a thread waiting to acquire the lock—it "spins", repeatedly attempting to acquire the lock until it is finally successful.

Here is incorrect code:

```
CLASS BadSpinLock EXPORT acq, rel =

TYPE FH          =  ENUM[free, held]
VAR  fh          := free

PROC acq() =
     DO << fh = held => SKIP >> OD;                    % wait for fh = free
     << fh := held >>                                  % and acquire it
PROC rel() = << fh := free >>

END BadSpinLock
```

This is wrong because two concurrent invocations of `acq` could both find `fh = free` and subsequently both set `fh := held` and return.

Here is correct code. It uses a more complex atomic command in the `acq` procedure. This command corresponds to the atomic "test-and-set" instruction provided by many real machines to code locks. It records the initial value of the lock, and then sets it to `held`. Then it tests the initial value; if it was `free`, then this thread was successful in atomically changing the state of the lock from `free` to `held`. Otherwise some other thread must hold the lock, so we "spin", repeatedly trying to acquire it until we succeed. The important difference in `SpinLock` is that the guard now involves only the local variable `t`, instead of the global variable `fh` in `BadSpinLock`. A thread acquires the lock when it is the one that changes it from `free` to `held`, which it checks by testing the value returned by the test-and-set.

```
CLASS SpinLock EXPORT acq, rel =

TYPE FH          =  ENUM[free, held]
VAR  fh          := free

PROC acq() = VAR t: FH |
     DO << t := fh; fh := held >>; IF t = free => RET [*] SKIP FI OD

PROC rel() = << fh := free >>

END SpinLock
```

Of course this code is not practical in general unless each thread has its own processor; it is used, however, in the kernels of most operating systems for computers with several processors. Later, in `MutexImpl`, we give practical code that queues a waiting thread.

The `SpinLock` code differs from the `Mutex` spec in another important way. It "forgets" which thread owns the mutex. The following `ForgetfulMutex` module is useful in understanding the

`SpinLock` code—in `ForgetfulMutex`, the threads get forgotten, but the atomicity is the same as in `Mutex`.

```
CLASS ForgetfulMutex EXPORT acq, rel =

TYPE FH          =  ENUM[free, held]
VAR  fh          := free

PROC acq() = << fh = free => fh := held >>
PROC rel() = << fh := free >>

END ForgetfulMutex
```

Note that `ForgetfulMutex` releases a mutex regardless of which thread acquired it, and it does a `SKIP` if the mutex is already free. This is one of the behaviors permitted by the `Mutex` spec, which allows anything under these conditions.

Later we will show that `SpinLock` implements `ForgetfulMutex` and that `ForgetfulMutex` implements `Mutex`, from which it follows that `SpinLock` implements `Mutex`. We don't give the abstraction function here because it involves the details of program counters.

Wait-free primitives

It's also possible to implement spin locks with a *wait-free* primitive rather than with test-and-set, although this rather misses the point of wait-free synchronization, which is discussed informally in handout 14.

The simplest wait-free primitive is compare-and-swap (CAS), which is illustrated in the following code for `acq`. It stores `new` into `fh` (which is an address parameter in real life) and returns true if the current contents of `fh` is `free`, otherwise it is `SKIP`. Now `acq` has no atomicity brackets.

```
VAR x : Any

PROC acq() = DO VAR t := CAS(free, held); IF t => RET [*] SKIP FI OD

APROC CAS(old: Any, new: Any)-> Bool =
     << IF x = old => x := new; RET true [*] RET false >>
```

A more general form of compare-and-swap allows you to do an arbitrary computation on the old contents of a variable. It is called load-locked/store-conditional. The idea is that if anyone writes the variable in between the load-locked and the store-conditional, the store fails.

```
VAR lock : Bool := false                               % a global variable; could be per variabl

APROC LL() -> Any = << lock := true; RET x >>

APROC SC(new: Any) -> BOOL = << IF lock => x := new; RET true [*] RET false >>
```

Now we can write `acq`:

```
PROC acq() = VAR fh', OK: Bool |
     DO   fh' := LL();
          IF fh' = free => OK := SC(held); IF OK => RET [*] SKIP FI
          [*] SKIP FI
     OD
```

Of course we can program `CAS` using `LL/SC`:

```
PROC CAS(old: Any, new: Any)-> Bool = VAR fh', OK: Bool |
```

```
    fh' := LL(); IF fh' = old => OK := SC(new); RET OK [*] RET false FI
```

We can also program operations such as incrementing a variable, either with CAS:

```
    VAR OK: Bool := false | DO ~OK => i := x; OK := CAS(i, i+1) OD
```

or with LL/SC:

```
    VAR OK: Bool := false | DO ~OK => i := LL(); OK := SC(i+1) OD
```

More generally, you can update an arbitrary data structure with an arbitrary function f by replacing i+1 in the CAS implementation with f(i). The way to think about this is that f computes a new version of i, and you install it if the version hasn't changed since you started. This is a form of optimistic concurrency control; see handout 20 for a more general discussion of this subject. Like optimistic concurrency control in general, the approach runs the danger of doing a lot of work that you then have to discard, or of starving some of the threads because they never get done before other threads sneak in and change the version out from under them. There are clever tricks for minimizing this danger; the basic idea is to queue your f for some other thread to execute along with its own.

Read/write locks

Here is a spec of a module that provides locks with two modes, read and write, rather than the single mode of a mutex. Several threads can hold a lock in read mode, but only one thread can hold a lock in write mode, and no thread can hold a lock in read mode if some thread holds it in write mode. In other words, read locks can be shared, but write locks are exclusive; hence the locks are also known as 'shared' and 'exclusive'.

CLASS RWLock EXPORT rAcq, rRel, wAcq, wRel =

```
TYPE ST          =  SET Thread

VAR  r           :  ST := {}
     w           :  ST := {}

APROC rAcq() =                                          % Acquires r if no current write locks
     <<    SELF IN (r \/ w) => HAVOC [*] w        = {} => r \/ := {SELF} >>

APROC wAcq() =                                          % Acquires w if no current locks
     <<    SELF IN (r \/ w) => HAVOC [*] (r \/ w) = {} => w    := {SELF} >>

APROC rRel() =                                          % Releases r if the thread has it
     << ~ (SELF IN r)       => HAVOC [*]                  r  - := {SELF} >>

APROC wRel() =
     << ~ (SELF IN w)       => HAVOC [*]                  w    := {}     >>

END RWLock
```

The following simple code is similar to ForgetfulMutex. It has the same atomicity as RWLock, but uses a different data structure to represent possession of the lock. Specifically, it uses a single integer variable rw to keep track of the number of readers (positive) or the existence of a writer (-1).

CLASS ForgetfulRWL EXPORT rAcq, rRel, wAcq, wRel =

```
VAR  rw          := 0
% >0 gives number of readers, 0 means free, -1 means one writer

APROC rAcq() = << rw >= 0 => rw + :=  1 >>
APROC wAcq() = << rw  = 0 => rw   := -1 >>
```



```
APROC rRel() = << rw - := 1 >>
APROC wRel() = << rw   := 0 >>

END ForgetfulRWL
```

We will see later how to code ForgetfulRWL using a mutex.

Condition variables

Mutexes are used to protect shared variables. Often a thread h cannot proceed until some condition is true of the shared variables, a condition produced by some other thread. Since the variables are protected by a lock, and can be changed only by the thread holding the lock, h has to release the lock. It is not efficient to repeatedly release the lock and then re-acquire it to check the condition. Instead, it's better for h to *wait* on a *condition variable*, as we saw in handout 14. Whenever any thread changes the shared variables in such a way that the condition might become true, it *signals* the threads waiting on that variable. Sometimes we say that the waiting threads 'wake up' when they are signaled. Depending on the application, a thread may signal one or several of the waiting threads.

Here is the spec for condition variables, copied from handout 14 on practical concurrency.

CLASS Condition EXPORT wait, signal, broadcast =

```
TYPE M = Mutex

VAR  c           :  SET Thread := {}
% Each condition variable is the set of waiting threads.

PROC wait(m) =
     << c \/ := {SELF}; m.rel >>;                       % m.rel=HAVOC unless SELF IN m
     << ~ (SELF IN c) => m.acq >>

APROC signal() = <<
% Remove at least one thread from c.  In practice, usually just one.
     IF   VAR t: SET Thread | t <= c /\ t # {} => c - := t [*] SKIP FI >>

APROC broadcast() = << c := {} >>

END Condition
```

As we saw in handout 14, it's not necessary to have a single condition for each set of shared variables. We want enough condition variables so that we don't wake up too many threads whose conditions are not yet satisfied, but not so many that the cost of doing all the signals is excessive.

Coding read/write lock using condition variables

This example shows how to use easy concurrency to make more complex locks and scheduling out of basic mutexes and conditions. We use a single mutex and condition for all the read-write locks here, but we could have separate ones for each read-write lock, or we could partition the locks into groups that share a mutex and condition. The choice depends on the amount of contention for the mutex.

Compare the code with ForgetfulRWL; the differences are highlighted with boxes. The <<...>> in ForgetfulRWL have become m.acq ... m.rel; this provides atomicity because shared variables are only touched while the lock is held. The other change is that each guard that could

block (in this example, all of them) is replaced by a loop that tests the guard and does `c.wait` if it doesn't hold. The release operations do the corresponding signal or broadcast operations.

```
CLASS RWLockImpl EXPORT rAcq, rRel, wAcq, wRel =   % implements ForgetfulRWL

VAR rw          :  Int := 0
    m           := m.new()
    c           := c.new()

% ABSTRACTION FUNCTION ForgetfulRWL.rw = rw

PROC rAcq(l) = m.acq; DO ~ rw >= 0 => c.wait(m) OD; rw + :=  1; m.rel
PROC wAcq(l) = m.acq; DO ~ rw  = 0 => c.wait(m) OD; rw   := -1; m.rel

PROC rRel(l) =
    m.acq; rw - := 1; IF rw = 0 => c.signal [*] SKIP FI; m.rel
PROC wRel(l) =
    m.acq; rw := 0;              c.broadcast;          m.rel

END RWLockImpl
```

This is the prototypical example for scheduling resources. There are mutexes (just `m` in this case) to protect the scheduling data structures, conditions (just `c` in this case) on which to delay threads that are waiting for a resource, and logic that figures out when it's all right to allocate a resource (the read or write lock in this case) to a thread.

Note that this code may starve a writer: if readers come and go but there's always at least one of them, a waiting writer will never acquire the lock. How could you fix this?

An unbounded FIFO buffer

In this section, we give a spec and code for a simple unbounded buffer that could be used as a communication channel between two threads. This is the prototypical example of a *producer-consumer* relation between threads. Other popular names for `Produce` and `Consume` are `Put` and `Get`.

```
MODULE Buffer[T] EXPORT Produce, Consume =

VAR b           :  SEQ T := {}

APROC Produce(t) = << b + := {t} >>
APROC Consume() -> T = VAR t | << b # {} => t := b.head; b := b.tail; RET t >>

END Buffer
```

The code is another example of easy concurrency.

```
MODULE BufferImpl[T] EXPORT Produce, Consume =

VAR b           :  SEQ T := {}
    m           := m.new()
    c           := c.new()

% ABSTRACTION FUNCTION Buffer.b = b

PROC Produce(t) =  m.acq; IF b = {} => c.signal [*] SKIP FI; b + := {t}; m.rel

PROC Consume() -> T = VAR t |
    m.acq; DO b = {} => c.wait(m) OD; t := b.head; b := b.tail; m.rel; RET t
```



```
END BufferImpl
```

Coding `Mutex` with memory

The usual way to code `Mutex` is to use an atomic test-and-set operation; we saw this in the `MutexImpl` module above. If such an operation is not available, however, it's possible to code `Mutex` using only atomic read and write operations on memory. This requires an amount of storage linear in the number of threads, however. We give a fair algorithm due to Peterson[8] for two threads; if thread `h` is competing for the mutex, we write `h*` for its competitor.

```
CLASS Mutex2Impl EXPORT acq, rel =

VAR req         :  Thread -> Bool := {* -> false}
    lastReq     :  Int

PROC acq() =
    [a0] req(SELF) := true;
    [a1] lastReq := SELF;
    DO [a2] (req(SELF*) /\ lastReq = SELF) => SKIP OD [a3]

PROC rel() = req(SELF) := false

END Mutex2Impl
```

This is hard concurrency, and it's tricky to show that it works. To see the idea, consider first a simpler version of `acq` that ensures mutual exclusion but can deadlock:

```
PROC acq0() =
    [a0] req(SELF) := true;
    DO [a2] req(SELF*) => SKIP OD [a3]                  % busy wait
```

We get mutual exclusion because once `req(h)` is true, `h*` can't get from a_2 to a_3. Thus `req(h)` acts as a lock that keeps the predicate `h*.$pc` = a_2 true once it becomes true. Only one of the threads can get to a_3 and acquire the lock. We might call the algorithm 'polite' because each thread defers to the other one at a_2.

Of course, `acq0` is no good because it can deadlock—if both threads get to a_2 then neither can progress. `acq` avoids this problem by making it a little easier for a thread to progress: even if `req(h*)`, `h` can take (a_2, a_3) if `lastReq # h`. Intuitively this maintains mutual exclusion because:

> If both threads are at a_2, only the one ≠ `lastReq`, say `h`, can progress to a_3 and acquire the lock. Since `lastReq` won't change, `h*` will remain at a_2 until `h` releases the lock.
>
> Once `h` has acquired the lock with `h*` not at a_2, `h*` can only reach a_2 by setting `lastReq := h*`, and again `h*` will remain at a_2 until `h` releases the lock.

It ensures progress because the `DO` is the only place to get stuck, and whichever thread is not in `lastReq` will get past it. It ensures fairness because the first thread to get to a_2 is the one that will get the lock first.

Abstractly, `h` has the mutex if `req(h) /\ h.$pc #` a_2, and the transition from a_2 to a_3 simulates the body of `Mutex.acq`. Precisely, the abstraction function is

[8] G. Peterson, A new solution to Lamport's concurrent programming problem using small shared variables. *ACM Trans. Programming Languages and Systems* **5**, 1 (Jan. 1983), pp 56-65.

```
    Mutex.m = (Holds0.set = {} => nil [*] Holds0.set.choose)
```

We sketch the proof that `Mutex2Impl` implements `Mutex` later.

There is lots more to say about coding `Mutex` efficiently, especially in the context of shared-memory multiprocessors.[9] Even on a uniprocessor you still need an implementation that can handle pre-emption; often the most efficient implementation gets the necessary atomicity by modifying the code for pre-emption to detect when a thread is pre-empted in the middle of the mutex code and either complete the operation or back up the state.

Multi-word clock

Often it's possible to get better performance by avoiding locking. Algorithms that do this are called 'wait-free'; we gave a brief discussion in handout 14. Here we present a wait-free algorithm due to Lamport[10] for reading and incrementing a clock, even if clock values do not fit into a single memory location that can be read and written atomically.

We begin with the spec. It says that a `Read` returns some value that the clock had between the beginning and the end of the `Read`. As we saw in handout 8 on generalized abstraction functions, where this spec is called `LateClock`, it takes a prophecy variable to show that this spec is equivalent to the simpler spec that just reads the clock value.

```
MODULE Clock EXPORT Read =

VAR  t            :  Int := 0                              % the current time

THREAD Tick() = DO << t + := 1 >> OD                       % demon thread advances t

PROC Read() -> Int = VAR t1: Int |
     << t1 := t >>; << VAR t2 | t1 <= t2 /\ t2 <= t => RET t2 >>

END Clock
```

The code below is based on the idea of doing reads and writes of the same multi-word data in opposite orders. `Tick` writes `hi2`, then `lo`, then `hi1`. `Read` reads `hi1`, then `lo`, then `hi2`; if it sees different values in `hi1` and `hi2`, there must have been at least one carry during the read, so `t` must have taken on the value `hi2 * base`. The function `T` expresses this idea. The atomicity brackets in the code are the largest ones that are justified by big atomic actions.

```
MODULE ClockImpl EXPORT Read =

CONST base       := 2**32

TYPE Word        =  Int SUCHTHAT word IN base.seq)

VAR  lo          :  Word := 0
     hi1         :  Word := 0
     hi2         :  Word := 0

% ABSTRACTION FUNCTION Clock.t = T(lo, hi1, hi2), Clock.Read.t1 = Read.t1Hist,
  Clock.Read.t2 = T(Read.tLo, Read.tH1, read.tH2)
```

[9] J. Mellor-Crummey and M. Scott, Algorithms for scalable synchronization of shared-memory multiprocessors. *ACM Transactions on Computer Systems* **9**, 1 (Feb. 1991), pp 21-65. A. Karlin et al., Empirical studies of competitive spinning for a shared-memory multiprocessor. *ACM Operating Systems Review* **25**, 5 (Oct. 1991), pp 41-55.
[10] L. Lamport, Concurrent reading and writing of clocks. *ACM Transactions on Computer Systems* **8**, 4 (Nov. 1990), pp 305-310.


```
THREAD Tick() = DO VAR newLo: Word, newHi: Word |
     << newLo := lo + 1 // base; newHi := hi1 + 1 >>;
     IF   << newLo # 0  => lo := newLo >>
     [*] << hi2 := newHi >>; << lo := newLo >>; << hi1 := newHi >>
     FI OD

PROC Read() -> Int = VAR tLo: Word, tH1: Word, tH2: Word |
     << tH1 := h1 >>;
     << tLo := lo >>;
     << tH2 := h2; RET T(tLo, tH1, tH2) >>

FUNC T(l: Int, h1: Int, h2: Int) -> Int = h2 * base + (h1 = h2 => l [*] 0)

END ClockImpl
```

Given this code for reading a two-word clock atomically starting with atomic reads of the low and high parts, it's obvious how to apply it recursively *n*–1 times to read an *n* word clock.

User and kernel mutexes and condition variables

This section presents code for mutexes and condition variables based on the Taos operating system from DEC SRC. Instead of spinning like `SpinLock`, it explicitly queues threads waiting for locks or conditions. The code for mutexes has a fast path that stays out of the kernel in `acq` when the mutex is free, and in `rel` when no other thread is waiting for the mutex. There is also a fast path for `signal`, for the common case that there's nobody waiting on the condition. There's no fast path for `wait`, since that always requires the kernel to run in order to reschedule the processor (unless a `signal` sneaks in before the kernel gets around to the rescheduling).

Notes on the code for mutexes:

1. `MutexImpl` maintains a queue of waiting threads, blocks a waiting thread using `Deschedule`, and uses `Schedule` to hand a ready thread over to the scheduler to run.

2. `SpinLock` and `ReleaseSpinLock` acquire and release a global lock used in the kernel to protect thread queues. This is OK because code running in the kernel can't be pre-empted.

3. The loop in `acq` serves much the same purpose as a loop that waits on a condition variable. If the mutex is already held, the loop calls `KernelQueue` to wait until it becomes free, and then tries again. `rel` calls `KernelRelease` if there's anyone waiting, and `KernelRelease` allows just one thread to run. That thread returns from its call of `KernelQueue`, and it will acquire the mutex unless another thread has called `acq` and slipped in since the mutex was released (roughly).

4. There is clumsy code in `KernelQueue` that puts the thread on the queue and then takes it off if the mutex turns out to be free. This is not a mistake; it avoids a race with `rel`, which calls `KernelRelease` to take a thread off the queue only if it sees that the queue is not empty. `KernelQueue` changes `q` and looks at `s`; `rel` uses the opposite order to change `s` and look at `q`.

This opposite-order access pattern often works in hard concurrency, that is, when there's not enough locking to do the job in a straightforward way. We saw another version of it in `Mutex2Impl`, which sets `req(h)` before reading `req(h*)`. In this case `req(h)` acts like a lock to keep `h*.$pc` = a_2 from changing from true to false. We also saw it in `ClockImpl`, where the reader and the writer of the clock touch its pieces in the opposite order.

The boxes show how the state, `acq`, and `rel` differ from the versions in `SpinLock`.

```
CLASS MutexImpl EXPORT acq, rel =                    % implements ForgetfulMutex

TYPE FH          =  Mutex.FH
VAR  fh          := free
     q           :  SEQ Thread := {}

PROC acq() = VAR t: FH |
     DO << t := fh; fh := held >>; IF t#held => RET [*] SKIP FI; KernelQueue() OD

PROC rel() = fh := free; IF q # {} => KernelRelease() [*] SKIP FI

% KernelQueue and KernelRelease run in the kernel so they can hold the spin lock and call the scheduler.

PROC KernelQueue() =
% This is just a delay until there's a chance to acquire the lock. When it returns acq will retry.
% Queuing SELF before testing fh ensures that the test in rel doesn't miss us.
% The spin lock keeps KernelRelease from getting ahead of us.
     SpinLock();                                     % indented code holds the lock
          q + := {SELF};
          IF   fh = free => q := q.reml              % undo previous line; will retry at acq
          [*]  Deschedule(SELF)                      % wait, then retry at acq
          FI;
     ReleaseSpinLock()

PROC KernelRelease() =
     SpinLock();                                     % indented code holds the lock
          IF q # {} => Schedule(q.head); q := q.tail [*] SKIP FI;
     ReleaseSpinLock()
     % The newly scheduled thread competes with others to acquire the mutex.

END MutexImpl
```

Now for conditions. Note that:

1. The 'event count' `ecSig` deals with the standard 'wakeup-waiting' race condition: the `signal` arrives after the `m.rel` but before the thread is queued. Note the use of the global spin lock as part of this. It looks as though `signal` always schedules exactly one thread if the queue is not empty, but other threads that are in `wait` but have not yet acquired the spin lock may keep running; in terms of the spec they are awakened by `signal` as well.

2. `signal` and `broadcast` test for any waiting threads without holding any locks, in order to avoid calling the kernel in this common case. The other event count `ecWait` ensures that this test doesn't miss a thread that is in `KernelWait` but hasn't yet blocked.

```
CLASS ConditionImpl EXPORT wait, signal, broadcast =     % implements Condition

TYPE M           =  Mutex

VAR  ecSig       :  Int := 0
     ecWait      :  Int := 0
     q           :  SEQ Thread := {}

PROC wait(m) = VAR i := ecSig | m.rel; KernelWait(i); m.acq

PROC signal() = VAR i := ecWait |
     ecSig + := 1; IF q # 0 \/ i # ecWait => KernelSig

PROC broadcast() = VAR i := ecWait |
     ecSig + := 1; IF q # 0 \/ i # ecWait => KernelBroadcast

PROC KernelWait(i: Int) =                                % internal kernel procedure
     SpinLock();                                         % indented code holds the lock
          ecWait + := 1;
          % if ecSig changed, there must have been a Signal, so return, else queue
          IF i = ecSig => q + := {SELF}; Deschedule(SELF) [*] SKIP FI;
     ReleaseSpinLock()

PROC KernelSig() =                                       % internal kernel procedure
     SpinLock();                                         % indented code holds the lock
          IF q # {} => Schedule(q.head); q := q.tail [*] SKIP FI;
     ReleaseSpinLock()

PROC KernelBroadcast() =
     SpinLock();                                         % indented code holds the lock
          DO q # {} => Schedule(q.head); q := q.tail OD;
     ReleaseSpinLock()

END ConditionImpl
```

The code for mutexes and conditions are quite similar; in fact, both are cases of a general semaphore.

Proving concurrent modules correct

This section explains how to prove the correctness of concurrent program modules. It reviews the simulation method that we have already studied, which works just as well for concurrent as for sequential modules. Then several examples illustrate how the method works in practice. Things are more complicated in the concurrent case because there are many more atomic transitions, and because the program counters of the threads are part of the state.

Before using this method in its full generality, you should first apply the theorem on big atomic actions as much as possible, in order to reduce the number of transitions that your proofs need to consider. If you are programming with easy concurrency, that is, if your code uses a standard locking discipline, this will get rid of nearly all the work. If you are doing hard concurrency, there will still be lots of transitions, and in doing the proof you will probably find bugs in your program.

The formal method

We use the same simulation technique that we used for sequential modules, as described in handouts 6 and 8 on abstraction functions. In particular, we use the most general version of this method, presented near the end of handout 8. This version does not require the transitions of the code to correspond one-for-one with the transitions of the spec. Only the external behavior (invocations and responses) must be the same—there can be any number of internal steps. The method proves that every trace (external behavior sequence) produced by the code can also be produced by the spec.

Of course, the utility of this method depends on an assumption that the external behavior of a module is all that is of interest to callers of the module. In other words, we are assuming here, as everywhere in this course, that the only interaction between the module and the rest of the program is through calls to the external routines provided by the module.

We need to show that each transition of the code simulates a sequence of transitions of the spec. An external transition must simulate a sequence that contains exactly one instance of the same external transition and no other external transitions; it can also contain any number of internal transitions. An internal transition must simulate a sequence that contains only internal transitions.

Here, once again, are the definitions:

Suppose T and S are modules with same external interface. An abstraction function F is a function from *states*(T) to *states*(S) such that:

Start: If u is any initial state of T then $F(u)$ is an initial state of S.

Step: If u and $F(u)$ are reachable states of T and S respectively, and (u, π, u') is a step of T, then there is an execution fragment of S from $F(u)$ to $F(u')$, having the same trace.

Thus, if π is an invocation or response, the fragment consists of a single π step, with any number of internal steps before and/or after. If π is internal, the fragment consists of any number (possibly 0) of internal steps.

As we saw in handout 8, we may have to add history variables to T in order to find an abstraction function to S (and perhaps prophecy variables too). The values of history variables are calculated in terms of the actual variables, but they are not allowed to affect the real steps.

An alternative to adding history variables is to define an abstraction relation instead of an abstraction function. An abstraction relation AR is a relation between *states*(T) and *states*(S) such that:

Start: If u is any initial state of T then there exists an initial state s of S such that $(u, s) \in AR$.

Step: If u and s are reachable states of T and S respectively, $(u, s) \in AR$, and (u, π, u') is a step of T, then there is an execution fragment of S from s to some s' having the same trace, and such that $(u', s') \in AR$.

Theorem: If there exists an abstraction function or relation from T to S then T implements S; that is, every trace of T is a trace of S.

Proof: By induction.

The strategy

The formal method suggests the following strategy for doing hard concurrency proofs.

1. Start with a spec, which has an abstract state.
2. Choose a concrete state for the code.
3. Choose an abstraction function, perhaps with history variables, or an abstraction relation.
4. Write code, identifying the critical actions that change the abstract state.
5. While (checking the simulation fails) do

   Add an invariant, checking that all actions of the code preserve it, or

   Change the abstraction function (step 3), the code (step 4), the invariant (step 5), or more than one, or

   Change the spec (step 1).

This approach always works. The first four steps require creativity; step 5 is quite mechanical except when you find an error. It is somewhat laborious, but experience shows that if you are doing hard concurrency and you omit any of these steps, your program won't work. Be warned.

Owicki-Gries proofs

Owicki and Gries invented a special case of this general method that is well known and sometimes useful.[11] Their idea is to do an ordinary sequential proof of correctness for each thread `h`, annotating each atomic command in the usual style with an assertion that is true at that point if `h` is the only thread running. This proof shows that the code of `h` establishes each assertion. Then you show that each of these assertions remains true after any command that any other thread can execute while `h` is at that point. This condition is called 'non-interference'; meaning not that other threads don't interfere with *access* to shared variables, but rather that they don't interfere with the *proof*.

The Owicki-Gries method amounts to defining an invariant of the form

`h.$pc = α ==> A`$_\alpha$ `/\ h.$pc = β ==> A`$_\beta$ `/\ ...`

and showing that it's an invariant in two steps: first, that every step of `h` maintains it, and then that every step of any other thread maintains it. The hope is that this decomposition will pay because the most complicated parts of the invariant have to do with private variables of `h` that aren't affected by other threads.

Prospectus for proofs

The remainder of this handout contains example proofs of correctness for several of the examples above: the `RWLockImpl` code for a read/write lock, the `BufferImpl` code for a FIFO buffer, the `SpinLock` code for a mutex (given in two versions), the `Mutex2Impl` code for a mutex used by two threads, and the `ClockImpl` code for a multi-word clock.

The amount of detail in these proofs is uneven. The proof of the FIFO buffer code and the second proof of the `Spinlock` code are the most detailed. The others give the abstraction functions and key invariants, but do not discuss each simulation step.

Read/write locks

We sketch how to prove directly that the module `RWLockImpl` implements `ForgetfulRWL`.This could be done by big atomic actions, since the code uses easy concurrency, but as an easy introduction discuss how to do it directly. The two modules are based on the same data, the variable `rw`. The difference is that `RWLockImpl` uses a condition variable to prevent threads in `acq` from busy-waiting when they don't see the condition they require. It also uses a mutex to restrict accesses to `rw`, so that a series of accesses to `rw` can be done atomically.

An abstraction function maps `RWLockImpl` to `ForgetfulRWL`. The interesting part of the state of `ForgetfulRWL` is the `rw` variable. We define that by the identity mapping from `RWLockImpl`.

[11] S. Owicki and D. Gries, An axiomatic proof technique for parallel programs. *Acta Informatica* **6**, 1976, pp 319-340.

The mapping for steps is mostly determined by the `rw` identity mapping: the steps that assign to `rw` in `RWLockImpl` are the ones that correspond to the procedure bodies in `ForgetfulRWL` Then the checking of the state and step correspondences is pretty routine.

There is one subtlety. It would be bad if a series of `rw` steps done atomically in `ForgetfulRWL` were interleaved in `RWLockImpl`. Of course, we know they aren't, because they are always done by a thread holding the mutex. But how does this fact show up in the proof?

The answer is that we need some invariants for `RWLockImpl`. The first, a "dominant thread invariant", says that only a thread whose name is in `m` (a 'dominant thread') can be in certain portions of its code (those guarded by the mutex). The dominant thread invariant is in turn used to prove other invariants called "data protection invariants".

For example, one data protection invariant says that if a thread (in `RWLockImpl`) is in middle of the assignment statement `rw + := 1`, then in fact `rw` ≥ 0 (that is, the test is still true). We need this data protection invariant to show that the corresponding abstract step (the body of `rAcq` in `ForgetfulRWLock`) is enabled.

`BufferImpl` implements `Buffer`

The FIFO buffer is another example of easy concurrency, so again we don't need to do a transition-by-transition proof for it. Instead, it suffices to show that a thread holds the lock `m` whenever it touches the shared variable `b`. Then we can treat the whole critical section during which the lock is held as a big atomic action, and the proof is easy. We will work out the important details of a low-level proof, however, in order to get some practice in a situation that is slightly more complicated but still straightforward, and in order to convince you that the theorem about big atomic actions can save you a lot of work.

First, we give the abstraction function; then we use it to show that the code simulates the spec. We use a slightly simplified version of `Produce` that always signals, and we introduce a local variable `temp` to make explicit the atomicity of assignment to the shared variable `b`.

Abstraction function

The abstraction function on the state must explain how to interpret a state of the code as a state of the spec. Remember that to prove a concurrent program correct, we need to consider the entire state of a module, including the program counters and local variables of threads. For sequential programs, we can avoid this by treating each external operation as a single atomic action.

To describe the abstraction function, we thus need to explain how to construct a state of the spec from a state of the code. So what is a state of the `Buffer` module above? It consists of:

- A sequence of items `b` (the buffer itself);
- for each thread that is active in the module, a program counter; and
- for each thread that is active in the module, values for local variables.

A state of the code is similar, except that it includes the state of the `Mutex` and `Condition` modules.

To define the mapping, we need to enumerate the possible program counters. For the spec, they are:

P_1 — before the body of `Produce`
P_2 — after the body of `Produce`
C_1 — before the body of `Consume`
C_2 — after the body of `Consume`

or as annotations to the code:

```
PROC Produce(t) = [P1] << b + := {t} >> [P2]

PROC Consume() -> T =
      [C1] << b # {} => VAR t := b.head | b := b.tail; RET t >> [C2]
```

For the code, they are:

- For a thread in `Produce`:

  p_1 — before `m.acq`
  in `m.acq`—either before or after the action
  p_2 — before `temp := b + {t}`
  p_3 — before `b := temp`
  p_4 — before `c.signal`
  in `c.signal`—either before or after the action
  p_5 — before `m.rel`
  in `m.rel`—either before or after the action
  p_6 — after `m.rel`

- For a thread in `Consume`:

  c_1 — before `m.acq`
  in `m.acq`—either before or after action
  c_2 — before the test `b # {}`
  c_3 — before `c.wait`
  in `c.wait`—at beginning, in middle, or at end
  c_4 — before `t := b.head`
  c_5 — before `temp := b.tail`
  c_6 — before `b := temp`
  c_7 — before `m.rel`
  in `m.rel`—either before or after action
  c_8 — before `RET t`
  c_9 — after `RET t`

or as annotations to the code:

```
PROC Produce(t) = VAR temp |
      [p1] m.acq;
      [p2] temp = b + {t};
      [p3] b := temp;
      [p4] c.signal;
      [p5] m.rel  [p6]

PROC Consume() -> T = VAR t, temp |
     [c1] m.acq;
     DO [c2] b # {} => [c3] c.wait OD;
     [c4] t := b.head;
     [c5] temp := b.tail; [c6] b := temp;
```

```
[c7] m.rel;
[c8] RET t [c9]
```

Notice that we have broken the assignment statements into their constituent atomic actions, introducing a temporary variable `temp` to hold the result of evaluating the right hand side. Also, the PC's in the `Mutex` and `Condition` operations are taken from the specs of those modules (*not* the code; we prove their correctness separately). Here for reference is the relevant code.

```
APROC acq() = << m = nil  => m := SELF; RET >>
APROC rel() = << m = SELF => m := nil ; RET [*] HAVOC >>

APROC signal() = << VAR hs: SET Thread |
    IF  hs <= c /\ hs # {} => c - := hs [*] SKIP FI >>
```

Now we can define the mapping on program counters:

- If a thread `h` is not in `Produce` or `Consume` in the code, then it is not in either procedure in the spec.
- If a thread `h` is in `Produce` in the code, then:

  If `h.$pc` is in $\{p_1, p_2, p_3\}$ or is in `m.acq`, then in the spec `h.$pc` = P_1.

  If `h.$pc` is in $\{p_4, p_5, p_6\}$ or is in `m.rel` or `c.signal` then in the spec `h.$pc` = P_2.
- If a thread `h` is in `Consume` in the code, then:

  If `h.$pc` $\in \{c_1, \ldots, c_6\}$ or is in `m.acq` or `c.wait` then in the spec `h.$pc` = C_1.

  If `h.$pc` is in $\{c_7, c_8, c_9\}$ or is in `m.rel` then in the spec `h.$pc` = C_2.

The general strategy here is to pick, for each atomic transition in the spec, some atomic transition in the code to simulate it. Here, we have chosen the modification of `b` in the code to simulate the corresponding operation in the spec. Thus, program counters before that point in the code map to program counters before the body in the spec, and similarly for program counters after that point in the code.

This choice of the abstraction function for program counters determines how each transition of the code simulates transitions of the spec as follows:

- If π is an external transition, π simulates the singleton sequence containing just π.
- If π takes a thread from a PC of p_3 to a PC of p_4, π simulates the singleton sequence containing just the body of `Produce`.
- If π takes a thread from a PC of c_6 to a PC of c_7, π simulates the singleton sequence containing just the body of `Consume`.
- All other transitions π simulate the empty sequence.

This example illustrates a typical situation: we usually find that a transition in the code simulates a sequence of either zero or one transitions in the spec. Transitions that have no effect on the abstract state simulate the empty sequence, while transitions that change the abstract state simulate a single transition in the spec. The proof technique used here works fine if a transition simulates a sequence with more than one transition in it, but this doesn't show up in most examples.

In addition to defining the abstract program counters for threads that are active in the module, we also need to define the values of their local variables. For this example, the only local variables are `temp` and the item `t`. For threads active in either `Produce` or `Consume`, the abstraction function on `temp` and `t` is the identity; that is, it defines the values of `temp` and `t` in a state of the spec to be the value of the identically named variable in the corresponding operation of the code.

Finally, we need to describe how to construct the state of the buffer `b` from the state of the code. Given the choices above, this is simple: the abstraction function is the identity on `b`.

Proof sketch

To prove the code correct, we need to prove some invariants on the state. Here are some obvious ones; the others we need will become clear as we work through the rest of the proof.

First, define a thread `h` to be *dominant* if `h.$pc` is in `Produce` and `h.$pc` is in $\{p_2, p_3, p_4, p_5\}$ or is at the end of `m.acq`, in `c.signal`, or at the beginning of `m.rel`, or if `h.$pc` is in `Consume` and `h.$pc` is in $\{c_2, c_3, c_4, c_5, c_6, c_7\}$ or is at the end of `m.acq`, at the beginning or end of `c.wait` (but not in the middle), or at the beginning of `m.rel`.

Now, we claim that the following property is invariant: a thread `h` is dominant if and only if `Mutex.m` = `h`. This simply says that `h` holds the mutex if and only if its PC is at an appropriate point. This is the basic mutual exclusion property. Amazingly enough, given this property we can easily show that operations are mutually exclusive: for all threads `h`, `h'` such that `h` ≠ `h'`, if `h` is dominant then `h'` is not dominant. In other words, at most one thread can be in the middle of one of the operations in the code at any time.

Now let's consider what needs to be shown to prove the code correct. First, we need to show that the claimed invariants actually are invariants. We do this using the standard inductive proof technique: Show that each initial state of the code satisfies the invariants, and then show that each atomic action in the code preserves the invariants. This is left as an exercise.

Next, we need to show that the abstraction function defines a simulation of the spec by the code. Again, this is an inductive proof. The first step is to show that an initial state of the code is mapped by the abstraction function to an initial state of the spec. This should be straightforward, and is left as an exercise. The second step is to show that the effects of each transition are preserved by the abstraction function. Let's consider a couple of examples.

- Consider a transition π from r to r' in which an invocation of an operation occurs for thread `h`. Then in state r, `h` was not active in the module, and in r', its PC is at the beginning of the operation. This transition simulates the identical transition in the spec, which has the effect of moving the PC of this thread to the beginning of the operation. So $AF(r)$ is taken to $AF(r')$ by the transition.
- Consider a transition in which a thread `h` moves from `h.$pc` = p_3 to `h.$pc` = p_4, setting `b` to the value stored in temp. The corresponding abstract transition sets `b` to `b + {t}`. To show that this transition does the right thing, we need an additional invariant:

  If `h.$pc` = p_3, then `temp = b + {t}`.

  To prove this, we use the fact that if `h.$pc` = p_3, then no other thread is dominant, so no other transition can change `b`. We also have to show that any transition that puts `h.$pc` at this point establishes the consequent of the implication — but there is only one transition that does this (the one that assigns to `temp`), and it clearly establishes the desired property.

The transition in `Consume` that assigns to `b` relies on a similar invariant. The rest of the transitions involve straightforward case analyses. For the external transitions, it is clear that they correspond directly. For the other internal transitions, we must show that they have no abstract effect, i.e., if they take *r* to *r*', then *AF*(*r*) = *AF*(*r*'). This is left as an exercise.

`SpinLock` implements `Mutex`, first version

The proof is done in two layers. First, we show that `ForgetfulMutex` implements `Mutex`. Second, we show that `SpinLock` implements `ForgetfulMutex`. For convenience, we repeat the definitions of the two modules.

```
CLASS Mutex EXPORT acq, rel =

VAR  m            :  (Thread + Null) := nil

PROC acq() = << m = nil  => m := SELF; RET >>
PROC rel() = << m = SELF => m := nil ; RET [*] HAVOC >>

END Mutex
```

```
CLASS ForgetfulMutex EXPORT acq, rel =

TYPE M            =  ENUM[free, held]
VAR  m            := free

PROC acq() = << m = free => m := held; RET >>
PROC rel() = << m := free; RET >>

END ForgetfulMutex
```

Proof that `ForgetfulMutex` implements `Mutex`

These two modules have the same atomicity. The difference is that `ForgetfulMutex` forgets which thread owns the mutex, and so it can't check that the "right" thread releases it. We use an abstraction relation *AR*. It needs to be multi-valued in order to put back the information that is forgotten in the code. Instead of using a relation, we could use a function and history variables to keep track of the owner and havoc. The single-level proof given later on that `Spinlock` implements `Mutex` uses history variables.

The main interesting relationship that *AR* must express is:

> *s*.`m` is non-`nil` if and only if *u*.`m` = `held`.

In addition, *AR* must include less interesting relationships. For example, it has to relate the `$pc` values for the various threads. In each module, each thread is either not there at all, before the body, or after the body. Thus, *AR* also includes the condition:

> The `$pc` value for each thread is the same in both modules.

Finally, there is the technicality of the special `$havoc = true` state that occurs in `Mutex`. We handle this by allowing *AR* to relate all states of `ForgetfulMutex` to any state with `$havoc = true`.

Having defined *AR*, we just show that the two conditions of the abstraction relation definition are satisfied.

The start condition is obvious. In the unique start states of both modules, no thread is in the module. Also, if *u* is the state of `ForgetfulMutex` and *s* is the state of `Mutex`, then we have *u*.`m = free` and *s*.`m = nil`. It follows that (*u*, *s*) ∈ *AR*, as needed.

Now we turn to the step condition. Let *u* and *s* be reachable states of `ForgetfulMutex` and `Mutex`, respectively, and suppose that (*u*, π, *u*') is a step of `ForgetfulMutex` and that (*u*, *s*) ∈ *AR*. If *s*.`$havoc`, then it is easy to show the existence of a corresponding execution fragment of `Mutex`, because any transition is possible. So we suppose that *s*.`$havoc = false`. Invocation and response steps are straightforward; the interesting cases are the internal steps.

So suppose that π is an internal action of `ForgetfulMutex`. We argue that the given step corresponds to a single step of `Mutex`, with "the same" action. There are two cases:

1. π is the body of an `acq`, by some thread `h`. Since `acq` is enabled in `ForgetfulMutex`, we have *u*.`m = free`, and `h.$pc` is right before the `acq` body in *u*. Since (*u*, *s*) ∈ *AR*, we have *s*.`m = nil`, and also `h.$pc` is just before the `acq` body in *s*. Therefore, the `acq` body for thread `h` is also enabled in `Mutex`. Let *s*' be the resulting state of `Mutex`.

   By the code, *u*'.`m = held` and *s*'.`m` = `h`, which correspond correctly according to *AR*. Also, since the same thread `h` gets the mutex in both steps, the PC's are changed in the same way in both steps. So (*u*', *s*') ∈ *AR*.

2. π is the body of a `rel`, by some thread `h`. If *u*.`m = free` then `ForgetfulMutex` does something sensible, as indicated by its code. But since (*u*, *s*) ∈ *AR*, *s*.`m = nil` and `Mutex` does `HAVOC`. Since `$havoc` in `Mutex` is defined to correspond to everything in `ForgetfulMutex`, we have (*u*', *s*') ∈ *AR* in this case.

   On the other hand, if *u*.`m = held` then `ForgetfulMutex` sets *u*'.`m := free`. Since (*u*, *s*) ∈ *AR*, we have *s*.`m` ≠ `nil`. Now there are two cases: If *s*.`m` = `h`, then corresponding changes occur in both modules, which allows us to conclude (*u*', *s*') ∈ *AR*. Otherwise, `Mutex` goes to `$havoc = true`. But as before, this is OK because `$havoc = true` corresponds to everything in `ForgetfulMutex`.

The conclusion is that every trace of `ForgetfulMutex` is also a trace of `Mutex`. Note that this proof does not imply anything about liveness, though in fact the two modules have the same liveness properties.

Proof that `SpinLock` implements `ForgetfulMutex`

We repeat the definition of `SpinLock`.

```
CLASS SpinLock EXPORT acq, rel =

TYPE M            =  ENUM[free, held]
VAR  m            := free

PROC acq() = VAR t: FH |
     DO << t := m; m := held >>; IF t # held => RET [*] SKIP FI OD
PROC rel() = << m := free >>

END SpinLock
```

These two modules use the same basic data. The difference is their atomicity. Because they use the same data, an abstraction function *AF* will work. Indeed, the point of introducing `ForgetfulMutex` was to take care of the need for history variables or abstraction relations there.

The key to defining *AF* is to identify the exact moment in an execution of SpinLock when we want to say the abstract acq body occurs. There are two logical choices: the moment when a thread converts *u*.m from free to held, or the later moment when the thread discovers that it has done this. Either will work, but to be definite we consider the earlier of these two possibilities.

Then *AF* is defined as follows. If *u* is any state of SpinLock then *AF*(*u*) is the unique state *s* of ForgetfulMutex such that:

- *s*.m = *u*.m, and
- The PC values of all threads "correspond".

We must define the sense in which the PC values correspond. The correspondence is straightforward for threads that aren't there, or are engaged in a rel operation. For a thread h engaged in an acq operation, we say that

- h.$pc in ForgetfulMutex, *s*.h.$pc, is just before the body of acq if and only if *u*.h.$pc is in SpinLock either (a) at the DO, and before the test-and-set ,or (b) after the test-and-set with h's local variable t equal to held.
- h.$pc in ForgetfulMutex, *s*.h.$pc, is just after the body of acq if and only if *u*.h.$pc is either (a) after the test-and-set with h's local variable t equal to free or (b) after the t # held test.

The proof that this is an abstraction function is interesting. The start condition is easy. For the step condition, the invocation and response cases are easy, so consider the internal steps. The rel body corresponds exactly in both modules, so the interesting steps to consider are those that are part of the acq. acq in SpinLock has two kinds of internal steps: the atomic test-and-set and the test for t # held. We consider these two cases separately:

1) The atomic test-and-set, (*u*, *test-and-set*, *u'*). Say this is done by thread h. In this case, the value of m *might* change. It depends on whether the step of SpinLock changes m from free to held. If it does, then we map the step to the acq body. If not, then we map it to the empty sequence of steps. We consider the two cases separately:

   3. The step changes m. Then in SpinLock, h.$pc moves after the test-and-set with h's local variable t = free. We claim first that the acq body in ForgetfulMutex is enabled in state *AF*(*u*). This is true because it requires only that *s*.m = free, and this follows from the abstraction function since *u*.m = free. Then we claim that the new states in the two modules are related by *AF*. To see this, note that m = held in both cases. And the new PC's correspond: in ForgetfulMutex, h.$pc moves to right after the acq body, which corresponds to the position of h.$pc in SpinLock, by the definition of the abstraction function.

   4. The step does not change m. Then h.$pc in SpinLock moves to the test, with t = held. Thus, there is no change in the abstract value of h.$pc.

2) The test for t # held, (*u*, *test*, *u'*). Say this is done by thread h. We always map this to the empty sequence of steps in ForgetfulMutex. We must argue that this step does not change anything in the abstract state, i.e., that *AF*(*u'*) = *AF*(*u*). There are two cases:

   5. If t = held, then the step of SpinLock moves h.$pc to after the DO. But this does not change the abstract value of h.$pc, according to the abstraction function, because both before and after the step, the abstract h.$pc value is before the body of acq.

   6. On the other hand, if t = free, then the step of SpinLock moves h.$pc to after the =>. Again, this does not change the abstract value of h.$pc because both before and after the step, the abstract h.$pc value is after the body of acq.

SpinLock implements Mutex, second version

Now we show again that SpinLock implements Mutex, this time with a direct proof that combines the work done in both levels of the proof in the previous section. For contrast, we use history variables instead of an abstraction relation.

Abstraction function

As usual, we need to be precise about what constitutes a state of the code and what constitutes a state of the spec. A state of the spec consists of:

- A value for m (either a thread or nil); and
- for each thread that is active in the module, a program counter.

There are no local variables for threads in the spec.

A state of the code is similar; it consists of:

- A value for m (either free or held);
- for each thread that is active in the module, a program counter; and
- for each thread that is active in acq, a value for the local variable t.

Now we have a problem: there is no way to define an abstraction function from a code state to a spec state. The problem here is that the code does not record which thread holds the mutex, yet the spec keeps track of this information. To solve this problem, we have to introduce a history variable or use an abstraction relation. We choose the history variable, and add it as follows:

- We augment the state of the code with two additional variables:

```
ms: (Thread + Null) := nil                    % m in the Spec
hs: Bool := false                             % $havoc in the Spec
```

- We define the effect of each atomic action in the code on the history variable; written in Spec, this results in the following modified code:

```
PROC acq() = VAR t: FH |
   DO <<t  := m; m := held>>; IF t # held => <<ms := SELF>>; RET [*] SKIP FI O

PROC rel() = << m := free; hs := hs \/ (ms # SELF); ms := nil >>
```

You can easily check that these additions to the code satisfy the constraints required for adding history variables.

This treatment of ms is the obvious way to keep track of the spec's m. Unfortunately, it turns out to require a rather complicated proof, which we now proceed to give. At the end of this section we will see a less obvious ms that allows a much simpler proof; skip to there if you get worn out.

Now we can proceed to define the abstraction function. First, we enumerate the program counters. For the spec, they are:

A_1 — before the body of acq
A_2 — after the body of acq
R_1 — before the body of rel
R_2 — after the body of rel

For the code, they are:

- For a thread in acq:

  a_1 — before the VAR t
  a_2 — after the VAR t and before the DO loop
  a_3 — before the test-and-set in the body of the DO loop
  a_4 — after the test-and-set in the body of the DO loop
  a_5 — before the assignment to ms
  a_6 — after the assignment to ms

- For a thread in rel:

  r_1 — before the body
  r_2 — after the body

The transitions in acq may be a little confusing: there's a transition from a_4 to a_3, as well as transitions from a_4 to a_5.

Here are the routines in Mutex annotated with the PC values:

```
APROC acq() = [A1] << m = nil => m := SELF >> [A2]

APROC rel() = [R1] << m # SELF => HAVOC [*] m := nil >> [R2]
```

Here are the routines in SpinLock annotated with the PC values:

```
PROC acq() = [a1] VAR t := FH |
     [a2] DO [a3] << t := m; m := held >>;
     [a4] IF t # held =>  [a5] << ms := SELF >>; [a6] RET [*] SKIP FI  OD;

PROC rel() =  [r1] << m := free; hs := hs \/ (ms # SELF); ms := nil >> [r2]
```

Now we can define the mappings on program counters:

- If a thread is not in acq or rel in the code, then it is not in either in the spec.
- $\{a_1, a_2, a_3, a_4, a_5\}$ maps to A_1, a_6 maps to A_2
- r_1 maps to R_1, r_2 maps to R_2

The part of the abstraction function dealing with the global variables of the module simply defines m in the spec to have the value of ms in the code, and $havoc in the spec to have the value of hs in the code. As in handout 8, we just throw away all but the spec part of the state.

Since there are no local variables in the spec, the mapping on program counters and the mapping on the global variables are enough to define how to construct a state of the spec from a state of the code.

Once again, the abstraction function on program counters determines how transitions in the code simulate sequences of transitions in the spec:

- If π is an external transition, π simulates the singleton sequence containing just π.
- If π takes a thread from a_5 to a_6, π simulates the singleton sequence containing just the transition from A_1 to A_2.
- If π takes a thread from r_1 to r_2, π simulates the singleton sequence containing just the transition from R_1 to R_2.
- All other transitions simulate the empty sequence.

Proof sketch

As in the previous example, we will need some invariants to do the proof. Rather than trying to write them down first, we will see what we need as we do the proof.

First, we show that initial states of the code map to initial states of the spec. This is easy; all the thread states correspond, and the initial state of ms in the code is nil.

Next, we show that transitions in the code and the spec correspond. All transitions from outside the module to just before a routine's body are straightforward, as are transitions from the end a routine's body to outside the module (i.e., when a routine returns). The transition in the body of rel is also straightforward. The hard cases are in the body of acq.

Consider all the transitions in acq before the one from a_5 to a_6. These all simulate the null transition, so they should leave the abstract state unchanged. And they do, because none of them changes ms.

The transition from a_5 to a_6 simulates the transition from A_1 to A_2. There are two cases: when ms = nil, and when ms ≠ nil.

1. In the first case, the transition from A_1 to A_2 is enabled and, when taken, changes the state so that m = SELF. This is just what the transition from a_5 to a_6 does.
2. Now consider the case when ms ≠ nil. We claim this case is possible only if a thread that didn't hold the mutex has done a rel. Then hs = true, the spec has done HAVOC, and anything can happen. In the absence of havoc, if a thread is at a_5, then ms = nil. But even though this invariant is what we want, it's too weak to prove itself inductively; for that, we need the following, stronger invariant:

   Either

   If m = free then ms = nil, and

   If a thread is at a_5, or at a_4 with t = free, then ms = nil, m = held, there are no other threads at a_5, and for all other threads at a_4, t = held

   or hs is true.

Given this invariant, we are done: we have shown the appropriate correspondence for all the transitions in the code. So we must prove the invariant. We do this by induction. It's vacuously true in the initial state, since no thread could be at a_4 or a_5 in the initial state. Now, for each transition, we assume that the invariant is true before the transition and prove that it still holds afterwards.

The external transitions preserve the invariant, since they change nothing relevant to it.

The transition in `rel` preserves the first conjunct of the invariant because afterwards both `m = free` and `ms = nil`. To prove that the transition in `rel` preserves the second conjunct of the invariant, there are two cases, depending on whether the spec allows `HAVOC`.

1. If it does, then the code sets `hs` true; this corresponds to the `HAVOC` transition in the spec, and thereafter anything can happen in the spec, so any transition of the code simulates the spec. The reason for explicitly simulating `HAVOC` is that the rest of the invariant may not hold after a rogue thread does `rel`. Because the rogue thread resets `m` to `free`, if there's a thread at a_5 or at a_4 with `t = free`, and `m = held`, then after the rogue `rel`, `m` is no longer `held` and hence the second conjunct is false This means that it's possible for several threads to get to a_5, or to a_4 with `t = free`. The invariant still holds, because `hs` is now true.

2. In the normal case `ms` ≠ `nil`, and since we're assuming the invariant is true before the transition, this implies that no thread is at a_4 with `t = free` or at a_5. After the transition to r_2 it's still the case that no thread is at a_4 with `t = free` or at a_5, so the invariant is still true.

Now we consider the transitions in `acq`. The transitions from a_1 to a_2 and from a_2 to a_3 obviously preserve the invariant. The transition from a_4 to a_5 puts a thread at a_5, but `t = free` in this case so the invariant is true after the transition by induction. The transition from a_4 to a_3 also clearly preserves the invariant.

The transition from a_3 to a_4 is the first interesting one. We need only consider the case `hs = false`, since otherwise the spec allows anything. This transition certainly preserves the first conjunct of the invariant, since it doesn't change `ms` and only changes `m` to `held`. Now we assume the second conjunct of the invariant true before the transition. There are two cases:

1. Before the transition, there is a thread at a_5, or at a_4 with `t = free`. Then we have `m = held` by induction, so after the transition both `t = held` and `m = held`. This preserves the invariant.

2. Before the transition, there are no threads at a_5 or at a_4 with `t = free`. Then after the transition, there is still no thread at a_5, but there is a new thread at a_4. (Any others must have `t = held`.) Now, if this thread has `t = held`, the second part of the invariant is true vacuously; but if `t = free`, then we have:

   `ms = nil` (since when the thread was at a_3 `m` must have been `free`, hence the first part of the invariant applies);

   `m = held` (as a direct result of the transition);

   there are no threads at a_5 (by assumption); and

   there are no other threads at a_4 with `t = free` (by assumption).

   So the invariant is still true after the transition.

Finally, assume a thread `h` is at a_5, about to transition to a_6. If the invariant is true here, then `h` is the only thread at a_5, and all threads at a_4 have `t = held`. So after it makes the transition, the invariant is vacuously true, because there is no other thread at a_5 and the threads at a_4 haven't changed their state.

We have proved the invariant. The invariant implies that if a thread is at a_5, `ms = nil`, which is what we wanted to show.

Simplifying the proof

This proof is a good example of how to use invariants and of the subtleties associated with preconditions. It's possible to give a considerably simpler proof, however, by handling the history variable `ms` in a less natural way. This version is closer to the two-stage proof we saw earlier. In particular, it uses the transition from a_3 to a_4 to simulate the body of `Mutex.acq`. We omit the `hs` history variable and augment the code as follows:

```
PROC acq() = [a1] VAR t := FH |
     [a2] DO [a3] << t := m; m := held; IF t # held => ms := SELF [*] SKIP FI >>;
     [a4] IF t # held =>  [a6]  RET  [a7] [*] SKIP FI  OD;

PROC rel() =  [r1] << m := free; ms := nil >> [r2]
```

The abstraction function maps `ms` to `Mutex.m` as before, and it maps PC's a_1- a_3 to A_1 and a_6-a_7 to A_2. It maps a_4 to A_1 if `t = held`, and to A_2 if `t = free`; thus a_3 to a_4 simulates `Mutex.acq` only if `m` was `free`, as we should expect. There is no need for an invariant; we only used it at a_5 to a_6, which no longer exists.

The simulation argument is the same as before except for a_3 to a_4, which is the only place where we changed the code. If `m = held`, then `m` and `ms` don't change; hence `Mutex.m` doesn't change, and neither does the abstract PC; in this case the transition simulates the empty trace. If `m = free`, then `m` becomes `held`, `ms` becomes `SELF`, and the abstract PC becomes A_2; in this case the transition simulates A_1 to A_2, as promised.

The moral of this story is that it can make a big difference how you choose the abstraction function. The crucial decision is the choice of the 'critical transition' that models the body of `Mutex.acq`, that is, how to abstract the PC. It seems very natural to change `ms` in the code after the test of `t # held` that is already there, but this forces the critical transition to be after the test. Then there has to be an invariant to carry forward the relationship between the local variable `t` and the global variable `m`, which complicates things, and the `HAVOC` case in `rel` complicates them further by falsifying the natural statement of the invariant and requiring the additional `hs` variable to patch things up. The uglier code with a second test of `t # held` inside the atomic test-and-set command makes it possible to use that action, which does the real work, to simulate the body of `Mutex.acq`, and then everything falls out nicely.

More complicated code requires invariants even when we choose the best abstraction function, as we see in the next two examples.

`Mutex2Impl` implements `Mutex`

This is the rather subtle code that implements a mutex for two threads using only memory reads and writes. We begin with a proof in the style of the last few, and then give an entirely different proof based on model checking.

First we show that the simple, deadlocking version `acq0` maintains mutual exclusion. Recall that we write `h*` for the thread that is the partner of thread `h`. Here are the spec and code again:

```
CLASS Mutex EXPORT acq, rel =

VAR m           :  (Thread + Null) := nil

PROC acq() = << m = nil  => m := SELF; RET >>
PROC rel() = << m = SELF => m := nil ; RET [*] HAVOC >>

END Mutex
```

```
CLASS Mutex2Impl0 EXPORT acq, rel =

VAR req         :  Thread -> Bool := {* -> false}
    lastReq     :  Int

PROC acq0() =
    [a1] req(SELF) := true;
    DO [a2] req(SELF*) => SKIP OD [a3]

PROC rel() = req(SELF) := false

END Mutex2Impl0
```

Intuitively, we get mutual exclusion because once `req(h)` is true, `h*` can't get from a_2 to a_3. It's convenient to define

```
FUNC Holds0(h: Thread) = RET req(h) /\ h.$pc # a2
```

Abstractly, `h` has the mutex if `Holds0(h)`, and the transition from a_2 to a_3 simulates the body of `Mutex.acq`. Precisely, the abstraction function is

```
    Mutex.m = (Holds0.set = {} => nil [*] Holds0.set.choose)
```

Recall that if `P` is a predicate, `P.set` is the set of arguments for which it is true.

To make precise the idea that `req(h)` stops `h*` from getting to a_3, the invariant we need is

```
    Holds0.set.size <= 1 /\ (h.$pc = a2 ==> req(h))
```

The first conjunct is the mutual exclusion. It holds because, given the first conjunct, only (a_2, a_3) can increase the size of `Holds0.set`, and `h` can take that step only if `req(h*) = false`, so `Holds0.set` goes from `{}` to `{h}`. The second conjunct holds because it can never be `true ==> false`, since only the step (a_1, `req(h) := true`, a_2) can make the antecedent true, this step also makes the consequent true, and no step away from a_2 makes the consequent false.

This argument applies to `acq0` as written, but you might think that it's unrealistic to fetch the shared variable `req(SELF*)` and test it in a single atomic action; certainly this will take more than one machine instruction. We can appeal to big atomic actions, since the whole sequence from a_2 to a_3 has only one action that touches a shared variable (the fetch of `req(SELF*)`) and therefore is atomic.

This is the right thing to do in practice, but it's instructive to see how to do it by hand. We break the last line down into two atomic actions:

```
    VAR t | DO [a2] << t := req(SELF*) >>; [a21] << t => SKIP >> OD [a3]
```


We examine several ways to show the correctness of this; they all have the same idea, but the details differ. The most obvious one is to add the conjunct `h.$pc # a21` to `Holds0`, and extend the mutual exclusion conjunct of the invariant so that it covers a thread that has reached a_{21} with `t = false`:

```
    (Holds0.set \/ {h | h.$pc = a21 /\ h.t = false}).size <= 1
```

Or we could get the same effect by saying that a thread acquires the lock by reaching a_{21} with `t = false`, so that it's the transition (a_2, a_{21}) with `t = false` that simulates the body of `Mutex.acq`, rather than the transition to a_3 as before. This means changing the definition of `Holds0` to

```
FUNC Holds0(h: Thread) =
    RET req(h) /\ h.$pc # a2 /\ (h.$pc = a21 ==> h.t = false)
```

Yet another approach is to make explicit in the invariant what `h` knows about the global state. One purpose of an invariant is to remember things about the global state that a thread has discovered in the past; the fact that it's an invariant means that those things stay true, even though other threads are taking steps. In this case, `t = false` in `h` means that either `req(h*) = false` or `h*` is at a_2 or a_{21}, in other words, `Holds(h*) = false`. We can put this into the invariant with the conjunct

```
    h.$pc = a21 /\ h.t = false ==> Holds(h*) = false
```

and this is enough to ensure that the transition (a_{21}, a_3) maintains the invariant.

We return from this digression on proof methodology to study the non-deadlocking `acq` from `Mutex2Impl`:

```
PROC acq() =
    [a0] req(SELF) := true;
    [a1] lastReq := self;
    DO [a2] (req(SELF*) /\ lastReq = SELF) => SKIP OD [a3]
```

We discussed liveness informally earlier, and we don't attempt to prove it. To prove mutual exclusion, we need to extend `Holds0` in the obvious way:

```
FUNC Holds(h: Thread) = req(h) /\ h.$pc # a1 /\ h.$pc # a2
```

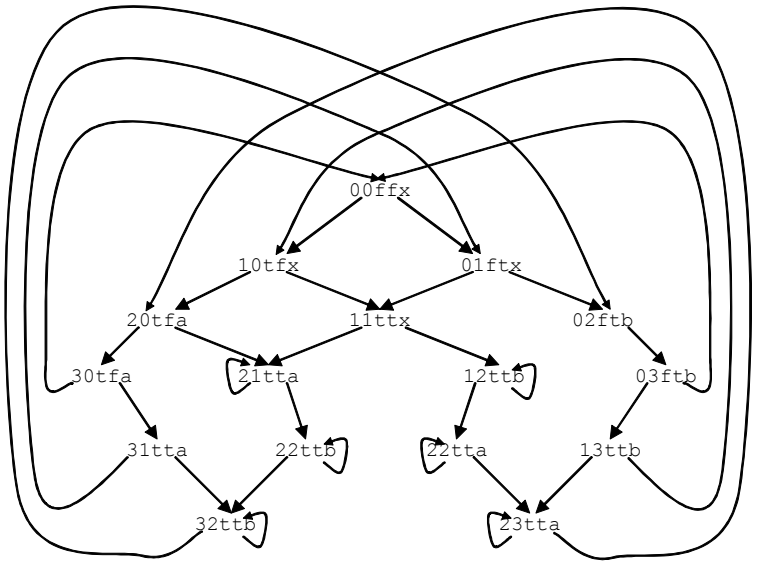
and add `\/ h.$pc = a1` to the antecedent of the invariant In order to have mutual exclusion, it must be true that `h` won't find `lastReq = h*` as long as `h*` holds the lock. We need to add a conjunct to the invariant to express this. This leaves us with:

```
    Holds0.set.size <= 1
/\ (h.$pc = a2 \/ h.$pc = a1 ==> req(h))
/\ (Holds(h*) /\ h.$pc = a2   ==> lastReq = h)
```

The last conjunct holds because (a_1, a_2) makes it true, and the only way to make it false is for `h*` to do `lastReq := SELF`, which it can only do from a_1, so that `Holds(h*)` is false. With this invariant it's obvious that (a_2, a_3) maintains the invariant.

Proof by model checking

We have been doing all our proofs by establishing invariants; these are called *assertional* proofs. An alternative method is to explore the state space exhaustively; this is called *model checking*. It



State machine for `Mutex2Impl.acq`, assuming `(req(SELF*) /\ lastReq = SELF)` is atomic

only works when the state space not too big. In this case, if the two threads are `a` and `b`, the state space is just:

```
a.$pc  IN {a0, a1, a2, a3}
b.$pc  IN {a0, a1, a2, a3}
req(a) IN {false, true}
req(b) IN {false, true}
lastReq IN {a, b}
```

We can write down a state concisely with one digit to represent each PC, a `t` or `f` for each `req`, and an `a` or `b` for `lastReq`. Thus `00ffa` is `a.$pc = a0`, `b.$pc = a0`, `req(a) = false`, `req(b) = false`, `lastReq = a`. When the value of a component is unimportant we write `x` for it.

The figure displays the complete state machine for `Mutex2Impl.acq`. Note the extensive symmetries. Nominally there are 128 states, but many are not reachable:

1. The value of `req` follows from the PC's, which cuts the number of reachable states to 32.
2. `33xxx` is not reachable. This is the *mutual exclusion invariant*, which is that both PC's cannot be in the critical section at the end of `acq`.This removes 2 states.
3. At the top of the picture the value of `lastReq` is not important, so we have shown it as `x`. This removes 4 states.
4. We can't have `20xxb` or `21xxb` or `30xxb` or `31xxb` or `32xxa`, or the 5 symmetric states, because of the way `lastReq` is set. This removes 10 states.

In the end there are only 16 reachable states, and 7 of them are obtained from the others simply by exchanging the two threads `a` and `b`.

Since there is no non-determinism in this algorithm and a thread is never blocked from making a transition, there are two transitions from each state, one for each thread. If there were no transitions from a state, the system would deadlock in that state. It's easy to see that the algorithm is live if both threads are scheduled fairly, since there are no non-trivial cycles that don't reach the end of `acq`. It is fair because the transitions from `00ffx` and `11ttx` are fair.

The appeal of model-checking should be clear from the example: we don't have to think, but can just search the state space mechanically. The drawback is that the space may be too large. This small example illustrates that symmetries can cut the size of the search dramatically, but the symmetries are often not obvious.

Unfortunately, this story is incomplete, because it assumed that we can evaluate `(req(SELF*) /\ lastReq = SELF)` atomically, which is not true. To fix this we have to break this evaluation down into two steps, with a new program counter value in the middle. We reproduce the whole procedure for easy reference:

```
PROC acq() =
     [a0] req(SELF) := true;
     [a1] lastReq := self;
     DO [a2] req(SELF*) => [a4] IF lastReq = SELF => SKIP FI OD [a3]

PROC rel() = req(SELF) := false
```

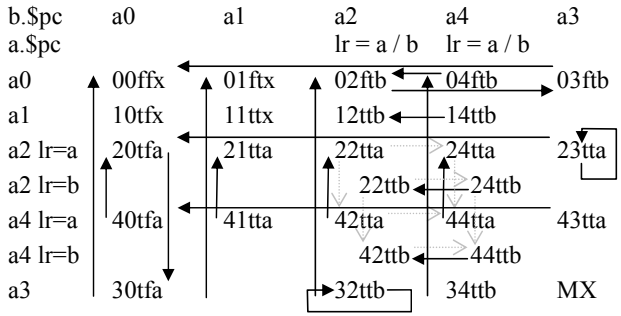
This increases the number of states from 128 to 200; the state space is:

```
a.$pc  IN {a0, a1, a2, a3, a4}
b.$pc  IN {a0, a1, a2, a3, a4}
req(a) IN {false, true}
req(b) IN {false, true}
lastReq IN {a, b}
```

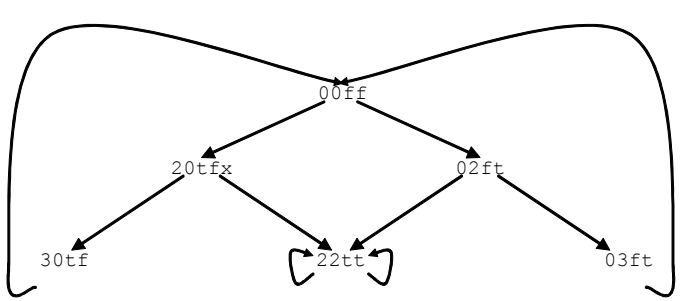
Most of the states are still unreachable, but there are 26 reachable states that we need to distinguish (using `x` as before when the value of a component doesn't affect the possible transitions). Instead of drawing a new state diagram like the previous one, we present a matrix that exposes the symmetries in a different way, by using the two PCs as the x and y coordinates. Except for the `x` in the `22ttx`, `24ttx`, and `42ttx` states, the PC values determine the other state components. The convention in this table is that for each state there's a transition that advances a thread's PC to the next non-blank row (for `a`) or column (for `b`) unless there's an arrow going somewhere else. Like the diagram, the table is symmetric.

The new transitions are the ones that involve a PC of a_4. These transitions don't change any state variables. As the table shows, what they do is to articulate the fine structure of the `22ttx` loops in the simpler state diagram. If `a` gets to `2xtta` it loops between that state and `4xttx`; similarly `b` loops between `x2ttb` and `x4ttb`. On the other hand, when `lastReq = b`, thread a can get through the sequence `2xttb`, `4xttb` to `3xttb` (where `x` can be `2` or `4`) and similarly `b` can get through the sequence `x2tta`, `x4tta` to `x3tta`.

These complications should suffice to convince you that model checking is for machines, not for people.



State machine for `Mutex2Impl.acq`, `(req(SELF*) /\ lastReq = SELF)` not atomic



State machine for `Mutex2Impl.acq`, which deadlocks in `22tt`

It's instructive to tell the same story for `acq0`, the implementation that can deadlock. The code is:

```
PROC acq0() =
    [a0] req(SELF) := true;
    DO [a2] req(SELF*) => SKIP OD [a3]
```

and there's no `lastReq` variable. The picture is much simpler; too bad it doesn't work. As you can see, there is no progress from `22tt`.

`ClockImpl` implements `Clock`

We conclude with the proof of the clock implementation. The spec says that a `Read` returns some value that the clock had between the beginning and the end of the `Read`. Here it is, with labels.

```
MODULE Clock EXPORT Read =
VAR t           :  Int := 0                          % the current time
THREAD Tick() = DO << t + := 1 >> OD                 % demon thread advances t
PROC Read() -> Int = VAR t1: Int |
    [R1] << t1 := t >>; [R2] << VAR t2 | t1 <= t2 /\ t2 <= t => RET t2 >> [R3]
END Clock
```


To show that `ClockImpl` implements this we introduce a history variable `t1Hist` in `Read` that corresponds to `t1` in the spec, recording the time at the beginning of `Read`'s execution. The invariant that is needed is based on the idea that `Read` might complete before the next `Tick`, and therefore the value `Read` would return by reading the rest of the shared variables must be between `t1Hist` and `Clock.t`. We can write this most clearly by annotating the labels in `Read` with assertions that are true when the PC is there.

```
MODULE ClockImpl EXPORT Read =
CONST base       := 2**32
TYPE Word        =  Int SUCHTHAT word IN base.seq)
VAR  lo          :  Word := 0
     hi1         :  Word := 0
     hi2         :  Word := 0
% ABSTRACTION FUNCTION Clock.t = T(lo, hi1, hi2), Clock.Read.t1 = Read.t1Hist,
  Clock.Read.t2 = T(Read.tLo, Read.tH1, read.tH2)
% The PC correspondence is R1 ↔ r1, R2 ↔ r2, r3, R3 ↔ r4

THREAD Tick() = DO VAR newLo: Word, newHi: Word |
     << newLo := lo + 1 // base; newHi := hi1 + 1 >>;
     IF   << newLo # 0  => lo := newLo >>
     [*] << hi2 := newHi >>; << lo := newLo >>; << hi1 := newHi >>
     FI OD

PROC Read() -> Int = VAR tLo: Word, tH1: Word, tH2: Word, t1Hist: Int |
     [r1] << tH1 := hi1; t1Hist := T(lo, hi1, hi2) >>;
     [r2] % I2: T(lo , tH1, hi2) IN t1Hist .. T(lo, hi1, hi2)
          << tLo := lo; >>
     [r3] % I3: T(tLo, tH1, hi2) IN t1Hist .. T(lo, hi1, hi2)
          << tH2 := hi2; RET T(tLo, tH1, tH2) >>
     [r4] % I4: $a IN t1Hist .. T(lo, hi1, hi2)

FUNC T(l: Int, h1: Int, h2: Int) -> Int = h2 * base + (h1 = h2 => l [*] 0)
END ClockImpl
```

The whole invariant is thus

```
h.$pc = r2 ==> I2 /\ h.$pc = r3 ==> I3 /\ h.$pc = r4 ==> I4
```

The steps of `Read` clearly maintain this invariant, since they don't change the value before `IN`. The steps of `Tick` maintain it by case analysis.

18. Consensus

Consensus (sometimes called 'reliable broadcast' or 'atomic broadcast') is a fundamental building block for distributed systems. Informally, we say that several processes achieve consensus if they all agree on some value. Three obvious applications are:

Distributed transactions, where all the processes need to agree on whether a transaction commits or aborts. Each transaction needs a new consensus on its outcome.

Membership, where a set of processes cooperating to provide a highly available service need to agree on which processes are currently functioning as members of the set. Every time a process fails or starts working again there must be a new consensus.

Electing a leader of a group of processes.

A less obvious, but much more powerful application is to replicate that state machines, which are discussed in detail below and in handout 28.

There are four important things to learn from this part of the course:

The idea of replicated state machines as a completely general method for building highly available, fault tolerant systems. In handout 28 we will discuss replicated state machines and other methods for fault tolerance in more detail.

The Paxos algorithm for distributed, fault tolerant consensus: how and why it works .

Paxos as an example of the best style for distributed, fault tolerant algorithms.

The correctness of Paxos as an example of the abstraction functions and simulation proofs applied to a very subtle algorithm.

Replicated state machines

There is a much more general way to use consensus, as the mechanism for coding a highly available state machine, which is the basic tool for building a highly available system. The way to get availability is to have either perfect components or redundancy. Perfect components are too hard, which leaves redundancy. The simplest form of redundancy is replication: have several copies or *replicas* of each component, and make sure that all the non-faulty components do the same thing. Since any computation can be expressed as a state machine, a replicated state machine can make any computation highly available.

Recall the basic idea of a replicated state machine:

If the transition relation is deterministic (in other words, is a function from (state, input) to (new state, output)), then several copies of the state machine that start in the same state and see the same sequence of inputs will do the same thing, that is, end up in the same state and produce the same outputs.

So if several processes are implementing the same state machine and achieve consensus on the values and order of the inputs, they will do the same thing. In this way it's possible to replicate an *arbitrary* computation and thus make it highly available. Of course we can make the order a

part of the value of the input by defining some total order on the set of possible inputs;[1] the easiest way to do this is simply to number them 1, 2, 3, ... We have already seen one application of this replicated state machine idea, in the code for transactions; there the replication takes the form of redoing a sequence of actions that is remembered in a log.

Suppose, for example, that we want to build a highly available file system. The transitions are read and write operations on the files (and rename, list, … as well). We make several copies of the file system and make sure that they process read and write operations in the same order. A client sends its operation to some copy, which gets consensus that it is the next operation. Then all the copies do the operation, and one of them returns the result to the client.

In many applications the inputs are requests from clients to the replicated service. Typically different clients generate their requests independently, so it's necessary to agree not only on what the requests are, but also on the order in which to serve them. The simplest way to do this is to number them with consecutive integers, starting at 1. This is especially easy in the usual implementation, 'primary copy' replication, since there's one place (the primary) to assign consecutive numbers. As we shall see, however, it's straightforward in any consensus scheme: you get consensus on input 1, then on input 2, etc.

You might think that a read could be handled by any copy with no need for consensus, since it doesn't change the state of the file system. Without consensus, however, a read might fail to see the result of a write that finished before the read started, since the read might be handled by a copy whose state is behind the current state of the file system. This result violates "external consistency", which is a formal expression of the usual intuition about state machines. In some applications, however, it is acceptable to get a possibly old result from a read, and then any copy can satisfy it without consensus. Another possibility is to notice that a given copy will have done all the operations up to *n*, and define a read operation that returns *n* along with the result value, and possibly the real time of operation *n* as well. Then it's up to the client to decide whether this is recent enough.

The literature is full of other schemes for achieving consensus on the order of requests when their total order is not derived from consecutive integers. These schemes label each input with some label from a totally ordered set (for instance, (client UID, timestamp) pairs) and then devise some way to be certain that you have seen all the inputs that can ever exist with labels smaller than a given value. They are complicated, and of doubtful utility.[2] People who do it for money use primary copy.[3]

Unfortunately, consensus is expensive. The section on optimizations at the end of this handout explains a variety of ways to make a replicated state machine run efficiently: leases, transactions, and batching.

[1] This approach was first proposed in a classic paper by Leslie Lamport: Time, clocks, and the ordering of events in a distributed system, *Comm. ACM* **21**, 7, July 1978, pp 558-565. This paper is better known for its analysis of the partial ordering of events in a distributed system, which is too bad.

[2] For details, see F. Schneider, Implementing fault-tolerant services using the state-machine approach: A tutorial, *ACM Computing Surveys* **22** (Dec 1990). This paper is reprinted in the book *Distributed Systems*, 2nd edition, ed. S. Mullender, Addison-Wesley, 1993, pp 169-197.

[3] Jim Gray said this about using locks for concurrency control; see handout 20.

Spec for consensus

Here is the spec for consensus; we have seen it already in handout 8 on history and prophecy variables. The idea is that the outcome of consensus should be one and only one of the allowed values. In the spec there is an `outcome` variable initialized to `nil`, and an action `Allow(v)` that can be invoked any number of times. There is also an action `Outcome` to read the `outcome` variable; it must return either `nil` or a `v` which was the argument of some `Allow` action, and if it doesn't return `nil` it must always return the *same* `v`.

More precisely, we have two requirements:

Agreement: Every non-`nil` result of `Outcome` is the same.

Validity: A non-`nil` `outcome` equals some allowed value.

Validity means that the outcome can't be any arbitrary value, but must be a value that was allowed. Consensus is reached by choosing some allowed value and assigning it to `outcome`. This spec makes the choice on the fly as the allowed values arrive.

```
MODULE Consensus [V] EXPORT Allow, Outcome =        % data value to agree on

VAR outcome   : (V + Null) := nil

APROC Allow(v) = << outcome = nil => outcome := v [] SKIP >>
APROC Outcome() -> (V + Null) = << RET outcome [] RET nil >>

END Consensus
```

Note that `Outcome` is allowed to return `nil` even after the choice has been made. This reflects the fact that in code with several replicas, `Outcome` is often coded by talking to just one of the replicas, and that replica may not yet have learned about the choice.

If only one `Allow` action occurs, there's no need to choose a `v`, and the code's only problem is to ensure termination. An algorithm that does so is said to implement 'reliable' or 'atomic' broadcast; there is only one sender, and either everyone or no one gets the message. The single `Allow` might not set `outcome`, which corresponds to failure of the sender of the broadcast message; in this case no one gets the message.

Here is an equivalent spec, slightly more complicated but perhaps more intuitive, and certainly closer to an implementation. It accumulates the allowed values and then chooses one of them in the internal action `Agree`.

```
MODULE LateConsensus [V] EXPORT Allow, Outcome =

VAR outcome     :  (V + Null) := nil
    allowed     :  SET V := {}

APROC Allow(v) = << allowed \/ := {v} >>

APROC Outcome() -> (V + Null) = << RET outcome [] RET nil >>
% Only outcome is visible

APROC Decide() = << VAR v :IN allowed | outcome = nil => outcome := v >>

END LateConsensus
```


It should be fairly clear that `LateConsensus` implements `Consensus`. An abstraction function to prove this, however, requires a prophecy variable, because `Consensus` decides on the outcome (in the `Allow` action) before `LateConsensus` does (in the `Decide` action). We saw these specs in handout 8 on generalized abstraction functions, where prophecy variables are explained.

In the code we have in mind, there are some processes, each with its own `outcome` variable initialized to `nil`. The `outcome` variables are supposed to reach consensus, that is, become equal to the argument of some `Allow` action. An `Outcome` can be directed to any process, which returns the value of its `outcome` variable. The tricky part is to ensure that two non-nil `outcome` variables are always equal, so that the agreement property is satisfied.

We would also like to have the property that eventually `Outcome` stops returning `nil`. In the code, this happens when every process' `outcome` variable is non-`nil`. However, this could take a long time if some process is very slow (or down for a long time).

We can change `Consensus` to express this with an internal action `Done`:

```
MODULE TerminatingConsensus [V] EXPORT Allow, Outcome =

VAR outcome     :  (V + Null) := nil
    done        :  Bool := false

APROC Allow(v) = << outcome = nil => outcome := v [] SKIP >>
APROC Outcome() -> (V + Null) = << RET outcome [] ~ done => RET nil >>

THREAD Done() = << outcome # nil => done := true >>

END TermConsensus
```

Note that this spec does not say anything about the processes in the assumed code; the abstraction function will say that `done` is true when all the processes have `outcome` ≠ `nil`.

An even stronger spec returns an `outcome` only when it's done:

```
APROC Outcome() -> (V + Null) = << done => RET outcome [] ~ done => RET nil >>
```

This is usually too strong for distributed code. It means that a process may not be able to respond to an `Outcome` request, since it can't return a value if it doesn't know the outcome yet, and it can't return `nil` if anyone else has already returned a value. If either the processes or the communication are asynchronous, it won't be possible in general for one process to know whether another one no longer matters because it has failed, or is just slow.

Facts about consensus

In this section we summarize the most important facts about when consensus is possible and what it costs. You can learn more about this in Nancy Lynch's course on distributed algorithms, 6.852J, or in her book cited in handout 2.

Fault models

To devise code for `Consensus` we need a precise model for the general setting of processes connected by links that can communicate messages from one process to another. In particular, the model must define what faults are possible. There are lots of ways to do this, and we have space to describe only the models that are most popular and closest to reality.

There are two broad classes of models:

- *Synchronous*, in which a non-faulty component makes its state transitions within a known amount of time. Usually this is coded by using a timeout, and declaring a component faulty if it fails to perform within the specified time.
- *Asynchronous*, in which nothing is known about the relative rate of progress of different components. In particular, a process can take an arbitrary amount of time to make a transition, and a link can take an arbitrary amount of time to deliver a message.

In general a process can send a message only to certain other processes; this "can send message" relation defines a graph whose edges are the links. The graph may be directed (it's possible that *A* can talk to *B* but *B* can't talk to *A*), but we will assume that communication is full-duplex so that the graph is undirected. Links are either working or faulty; a faulty link delivers no messages. Even a working link may lose messages, and in some models may lose any number of messages; it's helpful to think of such a system as one with totally asynchronous communication.

Processes are either working or faulty. There are two models for a faulty process:

- *Stopping* faults: a faulty process stops making transitions and doesn't start again. In an asynchronous model there's no way for another process to distinguish a stopped process or link from one that is simply very slow.
- *Byzantine* faults: a faulty process makes arbitrary transitions; these are named after the Byzantine Empire, famous for treachery. The motivation for this model is usually not fear of treachery, but ignorance of the ways in which a process might fail. Clearly Byzantine failure is an upper bound on how bad things can be.

Is consensus possible (will it terminate)?

A consensus algorithm terminates when the outcome variables of all non-faulty processes equal some allowed value. Here are the basic facts about consensus in some of these models.

- There is no consensus algorithm that is guaranteed to terminate in an asynchronous system with perfect links and even one process that has a stopping fault. This startling result is due to Fischer, Lynch, and Paterson.[4] It holds even if the communication system provides reliable broadcast that delivers each message either to all the non-faulty processes or to none of them. Real systems get around it by using timeout to make the system synchronous, or by using randomness.
- Even in a synchronous system with perfect processes there is no consensus algorithm that is guaranteed to terminate if an unbounded number of messages can be lost (that is, if communication is effectively asynchronous). The reason is that the last message sent must be pointless, since it might be lost. So it can be dropped to get a shorter algorithm. Repeat this argument to drop all the messages. But clearly an algorithm with no messages can't achieve consensus. The simplest case of this problem, with just two processes, is called the "two generals problem".
- In a system with both synchronous processes and synchronous communication, terminating consensus is possible. If *f* faults are allowed, then:

  For processes with stopping faults, consensus requires f+1 processes and an f+1-connected[5] network (that is, at least one good process and a connected subnet of good processes after all the allowed faults have happened). Even if the network is fully connected, it takes f+1 rounds to reach consensus in the worst case.

  For processors with Byzantine faults, consensus requires 3f+1 processes, a 2f+1-connected network, at least f+1 rounds of communication, and 2^f bits of data communicated.

  For processors with Byzantine faults and digital signatures (so that a process can present unforgeable evidence that another process sent it a message), consensus requires f+1 processes. Even if the network is fully connected, it takes f+1 rounds to reach consensus in the worst case.

The amount of communication required depends on the number of faults, the complexity of the algorithm, etc. Randomized algorithms can achieve better results with arbitrarily high probability.

Warning: In many applications the model of no more than *f* faults may not be realistic if the system is allowed to do the wrong thing when the number of faults exceeds *f*. It's often more important to do either the right thing or nothing.

The simplest consensus algorithms

There are two simple and popular algorithms for consensus. Both have the problem that they are not very fault-tolerant.

- A fixed 'leader', 'master', or 'coordinator' process that works like the `Consensus` spec: it gets all the `Allow` actions, chooses the outcome, and tells everyone. If it fails, you are out of luck. The abstraction function is just the identity on the leader's state; `TerminatingConsensus.done` is true iff everyone has gotten the outcome (or failed permanently). Standard two-phase commit for distributed transactions works this way.
- Simple majority voting. The abstraction function for `outcome` is the value that has a majority, or `nil` if there isn't one. This fails if you don't get a majority, or if enough members of a majority fail that it isn't a majority any more. In the latter case you can't determine the outcome. Example: `a` votes for `11`, `b` and `c` vote for `12`, and `b` fails. Now all you can see is one vote for `11` and one for `12`, so you can't tell that `12` had a majority.

The Paxos algorithm: The idea

In the rest of this handout, we describe Lamport's Paxos algorithm for coding asynchronous consensus; Liskov and Oki independently invented this algorithm as part of a replicated data storage system.[6] Its heart is the best asynchronous algorithm known, which is

---

[4] Fischer, M., Lynch, N., and Paterson, M., Impossibility of distributed consensus with one faulty process, *J. ACM* **32**, 2, April 1985, pp 374-382.

[5] A graph is connected if there is a path (perhaps traversing several links) between any two nodes, and disconnected otherwise. It is *k*-connected if *k* is the smallest number such that removing *k* links can leave it disconnected.

[6] L. Lamport, The part-time parliament, Technical report 49, Systems Research Center, Digital Equipment Corp, Palo Alto, Sep. 1989, finally published in *ACM Transactions on Computer Systems* **16**, 2 (May 1998), pp 133-169. Unfortunately, the terminology of this paper is confusing. B. Liskov and B. Oki, Viewstamped replication: A new

run by a set of *proposer* processes that guide a set of *acceptor* processes to achieve consensus,

correct no matter how many simultaneous proposers there are and no matter how often proposer or acceptor processes fail and recover or how slow they are, and

guaranteed to terminate if there is a single proposer for a long enough time during which each member of a majority of the acceptor processes is up for long enough, but

possibly non-terminating if there are always too many proposers (fortunate, since we know that guaranteed termination is impossible).

To get a complete consensus algorithm we combine this with a sloppy timeout-based algorithm for choosing a single proposer. If the sloppy algorithm leaves us with no proposer or more than one proposer for a time, the consensus algorithm may not terminate during that time. But if the sloppy algorithm ever produces a single proposer for long enough the algorithm will terminate, no matter how messy things were earlier.

Paxos is the way to do consensus if you want a high degree of fault-tolerance, don't have any real-time requirements, and can't tightly control the time to transmit and process a message. There isn't any simpler algorithm that has the same fault-tolerance. There is lots of code for consensus that doesn't work.

The grand plan of the algorithm is to have a sequence of *rounds*, each with a single proposer. This attacks the problem with simple majority voting, which is that a single attempt to get a majority may fall victim to failure. Each Paxos round is a distinct attempt to get a majority. Each acceptor has a state variable `s(a)` that is a function of the round; that is, there's a state value `s(a)(n)` for each round `n`. To reduce clutter, we write this s_n^a. In each round the proposer:

queries the acceptors to learn their state for past rounds,

chooses a *safe* value `v`,

commands the acceptors, trying to get a majority to *accept* `v`, and

if it gets a majority, that's a *decision*, and it distributes `v` as the *outcome* to everyone.

The outcome is the value accepted by a majority in some round. The tricky part of the algorithm is to ensure that there is only one such value, even though there may be lots of rounds.

Most descriptions of Paxos call the acceptors 'voters'. This is unfortunate, because the acceptors do not make any decisions; they do whatever a proposer requests, unless they have already done something inconsistent with that. In fact, an acceptor can be coded by a memory that has a compare-and-swap operation, as we shall see later. Of course, the proposers and acceptors can run on the same machine, and even in the same process. This is usually the way it's coded, but the algorithm with separate proposers and acceptors is easier to explain.

It takes a total of 2½ round trips for a deciding round. If there's only one proposer that doesn't fail, Paxos reaches consensus in one round. If the proposer fails repeatedly, or several proposers

primary copy method to support highly available distributed systems, *Proc. 7th ACM Conference on Principles of Distributed Computing*, Aug. 1988. In this paper the consensus algorithm is intertwined with the replication algorithm. See also B. Lampson, The ABCDs of Paxos, at http://research.microsoft.com/lampson/65-ABCDPaxos/Abstract.html.

fight it out, it may take arbitrarily many rounds to reach consensus. This may seem bad, but actually it is a good thing, since we know from Fischer-Lynch-Paterson that we can't have an algorithm that is guaranteed to terminate.

The rounds are numbered (not necessarily consecutively) with numbers of type `N`, and the numbers determine a total ordering on the rounds. Each round has a single value, which starts out `nil` and then may change to one of the allowed values; we write v_n for the value of round `n`. In each round an acceptor starts out `neutral`, and it can only change to v_n or `no`. A v_n or `no` state can't change. Note that different rounds can have different values. A round is *dead* if a majority has state `no`, and *decides* if a majority has state v_n. If a round decides, that round's value is the outcome.

The state of Paxos that contributes to the abstraction function to `LateConsensus` is

```
MODULE Paxos[                                          % implements Consensus
               V,                                      % data Value to decide on
               P WITH {"<=": (P, P)->Bool},            % Proposer; <= a total order
               A WITH {majority : SET A->Bool} ]       % Acceptor; majorities must intersect

TYPE I       = Int
     N       = [i, p] WITH {"<=":=LEqN}                % round Number; <= is total
     Z       = (ENUM[no, neutral] + V)                 % one acceptor's state in one round
     S       = A -> N -> Z                             % Acceptors' states

VAR  s       : S             := {*->{*->neutral}}     % acceptor working States
     outcome : A -> (V+Null) := {*->nil}              % acceptor outcome values
     allowed : P -> SET V    := {*->{}}               % proposer states
```

% Agreement: `(outcome.rng - {nil}).size <= 1`

% Validity: `(outcome.rng - {nil}) <= (\/ : allowed.rng)`

Paxos ensures that a round has a single value by having at most one proposer process per round, and making the proposer's identity part of the round number. So `N = [i, p]`, and proposer `p` proposes `(i, p)` for `n`, where `i` is an `I` that `p` has not used before, for instance, a local clock. The proposer keeps v_n in a volatile variable; rather than resuming an old round after a crash, it just starts a new one.

To understand why the algorithm works, it's useful to have the notion of a *stable* predicate on the state, a predicate that once true, remains true henceforth. Since the non-`nil` value of a round can't change, "v_n = v" is stable. Since a state value in an acceptor once set can't change, s_n^a = v and are stable; hence "round `n` is dead" and "round `n` decides" are stable as well. Note that s_n^a = `neutral` is not stable, since it can change to `v` or to `no`. Stable predicates are important, since they are the only kind of information that can usefully be passed from one process to another in a distributed system. If you get a message from another process telling you that `p` is true, and `p` is stable, then you know that `p` is true now (assuming that the other process is trustworthy).

We would like to have the idea of a *safe* value at round `n`: `v` is safe at `n` if any previous round that decided, decided on `n`. Unfortunately, this is not stable as it stands, since a previous round might decide later. In order to make it stable, we need to prevent this. Here is a more complex stable predicate that does the job:

`v` is safe at `n` = `(ALL n' | n' <=` n `==> n' is dead \/` $v_{n'}$ `= v)`

In other words, if you look back at rounds before `n`, skipping dead rounds, you see v_n. If all preceding rounds are dead, any v_n makes `n` safe. For this to be well-defined, we need an ordering on

N's, and for it to be useful we need a total ordering. We get this as the lexicographic ordering on the N = [i, p] pairs. This means that we must assume a total ordering on the proposers P.

With these preliminaries, we can give the abstraction function from Paxos to LateConsensus. For allowed it is just the union of the proposers' allowed sets. For outcome it is the value of a deciding round.

```
ABSTRACTION FUNCTION
     LateConsensus.allowed = \/ : allowed.rng
     LateConsensus.outcome = {n | Decides(n) || Value(n)}.choose

FUNC Decides(n) -> Bool = RET {a | s_n^a IS V}.majority
FUNC Value(n) -> (V + Null) = IF VAR a, v | s_n^a = v => RET v [*] RET nil FI
```

For this to be an abstraction function, we need an invariant:

(I1) Every deciding round has the same value.

It's easy to see that this follows from a stronger invariant: If round n' decides, then any later round's value is the same or nil.

```
(I2)  (ALL n', n | n' <= n /\ n' deciding    ==> v_n = nil   \/ v_n = v_n')
```

This in turn follows easily from something with a weaker antecedent:

```
(I3)  (ALL n', n | n' <= n /\ n' is not dead     ==> v_n = nil   \/ v_n = v_n')
=     (ALL n', n | v_n = nil \/ (n' <= n         ==> n' is dead  \/ v_n = v_n')
=     (ALL n | v_n = nil \/ (ALL n' | n' <= n ==> n' is dead  \/ v_n = v_n'))
=     (ALL n | v_n = nil \/ v_n is safe)
```

For validity, we also need to know that every round's value is allowed:

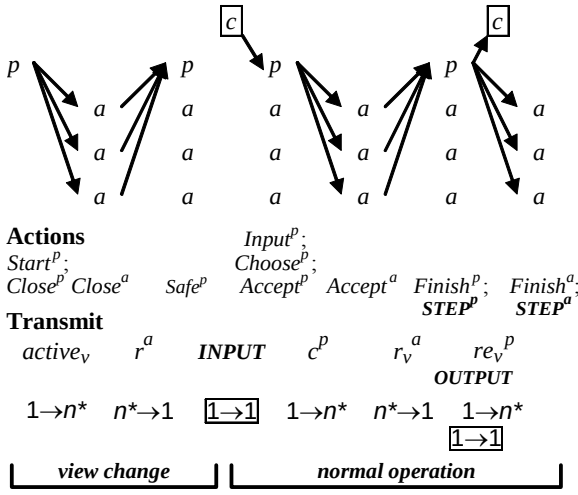
```
(I4)  (ALL n | v_n = nil \/ v_n IN (\/ : allowed.rng))
```

Initially all the v_n are nil so that (I3) and (I4) hold trivially. The Paxos algorithm maintains (I3) by choosing a safe value for a round. To accomplish this, the proposer chooses a new n and *queries* all the acceptors to learn their state in *all* rounds with numbers less than n. Before an acceptor responds, it *closes* a round earlier than n by setting any neutral state to no. Responses to this query from a majority of acceptors give the proposer enough information to find a safe value at round n, as follows:

> It looks back from n, skipping over rounds with no v state, since these must be dead (remember that the reported state is a v or no). When it comes to a round n' with $s_{n'}{}^a$ = v for some acceptor a, it takes v as safe. Since $v_{n'}$ is safe by (I3), and all the rounds between n' and n are dead, v is also safe.
>
> If all previous rounds are dead, any allowed value is safe.

Because 'safe' and 'dead' are stable properties, no state change can invalidate this choice.

Another way of looking at this is that because a deciding round is not dead and can never become dead, it forms a barrier that the proposer can't get past in choosing a safe value for a later round. Thus the deciding round forces any later safe value to be the same. A later round can propose differently from an earlier one only if the earlier one is dead.



Message flow in Paxos

An example may help to clarify the idea. Assume the allowed set is {x, y, w}. Given the following two sets of states in rounds 1 through 3, with three acceptors a, b, and c, the safe values for round 4 are as indicated. In the middle column the proposer can see a majority of acceptors in each round without seeing any v's, because all the rounds are dead. In the right-hand column there is a majority for w in round 2.

	State							
Value, acceptors	value	a	b	c	value	a	b	c
round 1	x	x	no	no	y	y	no	no
round 2	x	x	no	no	w	w	no	w
round 3	y	no	no	y	w	no	no	w
safe values for round 4	x, y, or w				w			

Note that only the latest v state from each acceptor is of interest, so only that state actually has to be transmitted.

Now in a second round trip the proposer *commands* everyone for round n. Each acceptor that is still neutral in round n (because it hasn't answered the query of a round later than n) *accepts* by changing its state to v_n in round n; in any case it reports its state to the proposer. If the proposer collects v_n reports from a majority of acceptors, then it knows that round n has succeeded, takes v_n as the agreed outcome of the algorithm, and sends this fact to all the processes in a final half round. Thus the entire process takes five messages or 2½ round trips.

Proposer p	**Message**	**Acceptor** a
Choose a new n_p, n		
Query a majority of acceptors for their status	query(n) → ← report(a, s^a)	**for all** n' < n, **if** $s_{n'}{}^a$ = neutral **then** $s_{n'}{}^a$:= no
Find a safe v at n. If all n' < n are dead, any v in $allowed_p$ is safe		
Command a majority of acceptors to accept v	command(n, v) → ← report(a, s^a)	**if** $s_n{}^a$ = neutral **then** $s_n{}^a$:= v
If a majority *accepts*, publish the outcome v	outcome(v) →	

How proposers and acceptors exchange messages in Paxos

When does a round succeed, that is, what action simulates the Decide action of the spec? It succeeds at the instant that some acceptor forms a majority by accepting its value, *even though* no acceptor or proposer knows at the time that this has happened. In fact, it's possible for the round to succeed without the proposer knowing this fact, if some acceptors fail after accepting but before getting their reports to the proposer, or if the proposer fails. In this case, some proposer will have to run another round, but it will have the same value as the invisibly deciding round.

When does Paxos terminate? If no proposer starts another round until after an existing one decides, then the algorithm definitely terminates as soon as the proposer succeeds in both querying and commanding a majority. It doesn't have to be the same majority for both, and the acceptors don't all have to be up at the same time. Therefore we want a single proposer, who runs one round at a time. If there are several proposers, the one running the biggest round will eventually succeed, but if new proposers keep starting bigger rounds none may ever succeed. This is fortunate, since we know from the Fischer-Lynch-Paterson result that there is no algorithm that is guaranteed to terminate.

It's easy to keep from having two proposers at once if there are no failures for a while, the processes have clocks, and the maximum time to send, receive, and process a message is known:

> Every potential proposer that is up broadcasts its name.
>
> You become the proposer one round-trip time after doing a broadcast unless you have received the broadcast of a bigger name.

The algorithm makes minimal demands on the properties of the network: lost, duplicated, or reordered messages are OK. Because nodes can fail and recover, a better network doesn't make things much simpler. We model the network as a broadcast medium from proposer to acceptors; in practice this is usually coded by individual messages to each acceptor. We describe continuous retransmission; in practice acceptors retransmit only in response to the proposer's retransmission.

A process acting as a proposer uses messages to communicate with the same process acting as an acceptor, so we describe the two roles of each process completely independently. In fact, the proposer need not be an acceptor at all.

Note the structure of the algorithm as a collection of independent atomic actions, each taking place at a single process. There are no non-atomic procedures except for the top-level scheduler, which simply chooses an enabled atomic action to run next. This structure makes it much easier to reason about the algorithm in the presence of concurrency and failures.

The next section gives the algorithm in detail. You can skip this, but be sure to read the following section on optimizations, which has important remarks about using Paxos in practice.

The Paxos algorithm: The details

We give a straightforward version of the algorithm in detail, and then describe encodings that reduce the information stored and transmitted to small fixed-size amounts. The first set of types and variables is copied from the earlier declarations.

```
MODULE Paxos[                                          % implements Consensus
    V,                                                 % data Value to decide on
    P WITH {"<=": (P, P)->Bool                         % Proposer; <= a total order
                  SUCHTHAT IsTotal(this)},
    A WITH {majority: SET A->Bool}                     % Acceptor
      SUCHTHAT IsMaj(this)} ]

TYPE I           =  Int
    N            =  [i, p] WITH {"<=":=LEqN}           % round Number; <= total
    Z            =  (ENUM[no, neutral] + V)            % one acceptor's state in one round
    S            =  A -> N -> Z                        % Acceptors' states

VAR outcome      :  A -> (V+Null) :={*->nil}           % the acceptors' state, in
    s            :  S             :={*->{*->neutral}}  % two parts.
    allowed      :  P -> SET V    :={*->{}}            % the proposers' state
    % The rest of the proposers' state is the variables of ProposerActions(p).
    % All volatile except for n, which we make stable for simplicity.

TYPE K           =  ENUM[query,command,outcome,report] % Kind of message
    M            =  [k,                                % Message; kind of message,
                     n,                                % about round n
                     x: (Null + S + V) ]               % acceptor state or outcome.
                                                       % S defined for just one a
    Phase        =  ENUM[idle, querying, commanding]   % of a proposer process

CONST n0         := N{i:=0, p:={p | true}.min}         % smallest N

% Actions for handling messages

APROC Send(k, n, x: ANY) = << UnreliableCh.Put({M{k, n, x}}) >>
APROC Receive(k) -> M = << VAR m | m := UnreliableCh.Get(); m.k = k => RET m >>

% External actions. Can happen at any proposer or acceptor.

APROC Allow(v)       = << VAR p | allowed(p) := allowed(p) \/ {v} >>
APROC Outcome() -> V = << VAR a | RET outcome(a) >>
```

```
THREAD ProposerActions(p) =
  VAR                                                  % proposer state (volatile except n)
    n           := N{i := 1, p := p},                  % last round started
    phase       := idle,                               % proposer's phase
    pS          := S{},                                % Proposer info about acceptor State
    v: (V+Null) := nil                                 % used iff phase = commanding
    |

DO << % Pick an enabled action and do it.
    % Crash. We don't care about the values of pS or v.
    phase := idle; allowed := {}; pS := {}; v := nil

[] % New round. Hope n becomes largest N in use so this round succeeds.
    phase = idle => VAR i | i > n.i => n.i := i; pS := {}; phase := querying

[] % Send query message.
    phase = querying => Send(query, n, nil)

[] % Receive report message for the current round.
    << phase = querying => VAR m := Receive(report) |
          m.n = n => pS + := m.x >>

[] % Start command. Note that Dead can't be true unless pS has a majority.
    << phase = querying =>
      IF  VAR n1 |   (ALL n' | n1 < n' /\ n' < n ==> Dead(pS, n'))
                  /\ Val(pS, n1) # nil => v := Val(pS, n1)
      [*] (ALL n'| n' < n ==> Dead(pS, n')) => VAR v' :IN allowed(p) | v := v'
      FI;
      pS := S{}; phase := commanding >>

[] % Send command message.
    phase = commanding => Send(command, n, v)

[] % Receive report message for the current round with non-neutral state.
    << phase = commanding => VAR m := Receive(report), s1 := m.x, a |
        m.n = n /\ s1!a /\ s1_n^a # neutral => pS_n^a:= s1_n^a >>

[] % Send outcome message if a majority has V state.
    Decides(pS, n) => Send(outcome, n, v)

[] % This round is dead if a majority has no state. Try another round.
    Dead(pS, n) => phase := idle

>> OD


THREAD AcceptorActions(a) = % State is in s^a and outcome^a, which are stable.

DO << % Pick an enabled action and do it.

   % Receive query message, change neutral state to no for all before m.n, and
   % send report. Note that this action is repeated each time a query message arrives.
    VAR m := Receive(query) |
        DO VAR n | n < m.n /\ s_n^a = neutral => s_n^a:= no OD;
        Send(report, m.n, s.restrict({a}))

[] % Receive command message, change neutral state to v_n, and send state message.
   % Note that this action is repeated each time a command message arrives.
```



```
    VAR m := Receive(command) |
        IF s_m.n^a = neutral => s_m.n^a:= m.x [*] SKIP FI;
        Send(report, m.n, s.restrict({a}).restrict({n}))}

[] % Receive outcome message.
    VAR m := Receive(outcome) | outcome^a:= m.x

>> OD
```

==========Useful functions for the proposer choosing a value==========

```
FUNC LEqN(n1, n2) -> Bool =                                % lexicographic ordering
    RET n1.i < n2.i \/ (n1.i = n2.i /\ n1.a <= n2.a)

FUNC Dead(s', n)       -> Bool = RET {a | s'!a /\ s'(a)(n) = no}.majority
FUNC Decides(s', n) -> Bool = RET {a | s'!a /\ s'(a)(n) IS V}.majority

FUNC Val(s', n) -> (V+Null) = IF VAR a, v | s'(a)(n) = v => RET v [*] RET nil FI
% The value of round n according to s': if anyone has v then v else nil.
```

==========Useful functions for the invariants==========

```
% We write x_1 for ProposerActions(p).x to make the formulas more readable.

FUNC IsTotal(le: (P, P) -> Bool) -> Bool =                 % Is le a total order?
    RET ( ALL p1, p2, p3 |   (le(p1, p2) \/ le(p2, p1))
                           /\ (le(p1, p2) /\ le(p2, p3) ==> le(p1, p3)) )

IsMaj(m: SET A->Bool) -> Bool =                            % any two must intersect
    RET (ALL aa1: SET A, aa2: SET A | (m(aa1) /\ m(aa2) ==> aa1 /\ aa2 ≠ {})

FUNC Allowed() -> SET V = RET \/ : allowed.rng

FUNC Decides(n) -> Bool = RET Decides(s, n)
FUNC Value(n) -> (V+Null)  = RET Val(s, n)                 % The value of round n

FUNC ValAnywhere(n) -> SET V =
% Any trace of the value of round n, in messages, in s, or in a proposer.
    RET    {m, a, s1 | m IN UnreliableCh.q /\ m.x = s1 /\ s1!a /\ s1(a)!n
                     | s1_n^a }
        \/ {Value(n)}
        \/ (phase_n.1 = commanding => {v_n.1} [*] {})

FUNC SafeVals(n) -> SET V =
% The safe v's at n.
    RET {v | ( ALL n' | n' < n ==> Dead(s, n') \/ v = Val(s, n') )}

FUNC GoodVals(n) -> SET V =
% The good values for round n: if there's no v yet, the ones that satisfy
% (I3) and validity. If there is a v, then v.
    RET ( Value(n) = nil => SafeVals(n) /\ Allowed() [*] {Value(n)} )
```

============================== Invariants ==============================

```
% Proofs are straightforward except as noted. Observe that s changes only when an acceptor receives query
% (when it may add no state) or when  an acceptor receives command (when it may add V state).

% (1) A proposer's n.p is always the proposer itself.
( ALL p | n_p.p = p )                                      % once p has taken an action
% (2) Ensure that there's no value yet for any round that a proposer might start.
( ALL p, n |    n.p = p /\ (n > n_p \/ (n = n_p /\ phase_p = querying))
          ==> Value(n) = nil )
```

```
% (3)  s always has a most one value per round, because a only changes s^a (when a receives query or command)
% from neutral, and either to no (query) or to agree with the proposer (command).
(ALL n | {a | s!a /\ s_n^a IS V || s_n^a}.size <= 1)

% (4)  All the S's in the channel or in any pS agree with s.
( ALL s1 :IN ({m | m IN UnreliableCh.q /\ m.x IS S || m.x} \/ {p || pS_p}) |
    (ALL a, n | s1!a /\ s1(a)!n /\ s1_n^a # neutral ==> s1_n^a = s_n^a))

% (5) Every round value is allowed
( ALL n | Value(n) IN (Allowed()\/ {nil}) )

% (6)  If anyone thinks v is the value of a round, it is a good round value.
( ALL n | ValAnywhere(n) <= v IN GoodVals(n)) )
% (7) A round has only one non-nil value.
( ALL n | ValAnywhere(n).size <= 1 )

% Major invariant (I3).
( ALL n | Value(n) IN ({nil} \/ SafeVals(n)) )
% Easy consequence (I2)
( ALL n1, n2 | n1 < n2 /\ Decides(n1) ==> Value(n2) IN {nil, Value(n1)} )
% Further easy consequence (I1)
( ALL a | outcome(a) # nil ==> (EXISTS n | Decides(n) /\ outcome(a) = Value(n))

END Paxos
```

Optimizations

It's possible to reduce the size of the proposer and acceptor state and the messages transmitted to a few bytes, and in many cases to reduce the latency and the number of messages sent by combining rounds of the algorithm.

Reducing state and message sizes

It's not necessary to store or transmit the complete acceptor state s^a. Instead, everything can be encoded in a small fixed number of N's, A's, and V's, as follows.

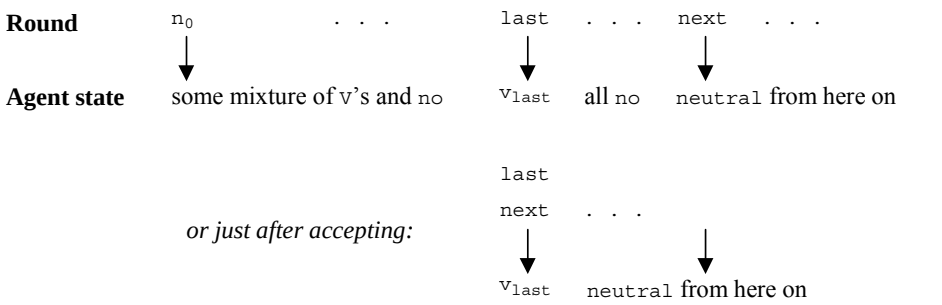
The relevant part of S in an acceptor or in a `report` message consists of v_{last} in some round `last`, plus `no` in all rounds strictly between `last` and some later round `next`, and `neutral` in any round after `last` and at or after `next`. Hence S can be encoded simply as $(v_{last}, last, next)$. The part of S before `last` is just history and is not needed for the algorithm to run, because we know that v_{last} is safe. Making this precise, we have

```
VAR y: [z, last: N, next: N] := {no; n0; n0}
```

and s^a is just

```
(\ n | (n <= y.last => y.z [*] n < y.next => no [*] neutral))
```

Note that this encoding loses the details of the state for rounds before `last`, but the algorithm doesn't need this information Here is a picture of this coding scheme.



In a proposer, there are two cases for pS.

- If `phase = querying`, pS consists of the s^a's, for rounds less than n, transmitted by a set of acceptors a. Hence it can be encoded as a set of 'last state' tuples $(a, last_a, v)$. From this we care only about the number of a's (assuming that `A.majority` just counts acceptors), the biggest `last`, and its corresponding v. So the whole thing can be coded as (count of A's, $last_{max}$, v).
- If `phase = commanding`, pS consists of a set of v_n or `no` in round n, so it can be encoded as the set of acceptors responding. We only care about a majority, so we only need to count the number of acceptors responding.

The proposers need not be the same processes as the acceptors. A proposer doesn't really need any stable state, though in the algorithm as given it has n. Instead, it can poll for the `next`'s from a majority after a failure and choose an n with a bigger `n.i`. This will yield an n that's larger than any n from this proposer that has appeared in a `command` message so far (because n can't get into a `command` message without having once been the value of `next` in a majority of acceptors), and this is all we need to keep s good. In fact, if a proposer ever sees a report with a `next` bigger than its own n, it should either stop being a proposer or immediately start another round with a new, larger n, because the current round is unlikely to succeed.

Combining rounds

In the most important applications of Paxos, we can combine the first round trip (`query/report`) with something else. For a commit algorithm, we can combine the first round-trip with the prepare message and its response; see handout 27 on distributed transactions.

The most important application is a state machine that needs to decide on a sequence of actions. We can number the actions $a_0, a_1, \ldots, a_k$, run a separate instance of the algorithm for each action, and combine the `query/report` messages for all the actions. Note that these action numbers, which we are calling K's, are *not* the same as the Paxos round numbers, the N's; each action has its own instance of Paxos and therefore its own set of round numbers. In this application, the proposer is usually called the *primary*. The following trick allows us to do the first round trip only once after a new primary starts up: interpret the `report` messages for action k as applying not just to consensus on a_k, but to consensus on a_j for *all* `j >= k`.

As long as the same process continues to be proposer, it can keep track of the current K in its private state. A new proposer needs to learn the first unused K. If it tries to get consensus using a number that's too small, it will discover that there's already an outcome for that action. If it uses a number k that's too big, however, it can get consensus. This is tricky, since it leads to a gap in

the action numbers. Hence you can't apply the decided action k, since you don't know the state at k because you don't know all of the preceding actions that are applied to make that state. So a new proposer must find the first unused K for its first action a. A clumsy way to do this is to start at k = 0 and try to get consensus on successive a_k's until you get consensus on a.

You might think that this is silly. Why not just poll the acceptors for the largest K for which one of them knows the outcome? This is a good start, but it's possible that consensus was reached on the last a_k (that is, there's a majority for it among the acceptors) but the outcome was not published before the proposer crashed. Or it was not published to a majority of the acceptors, and all the ones that saw it have also failed. So after finding that k is the largest K with an outcome, the proposer may still discover that the consensus on a_{k+1} is not for its action a, but for some earlier action whose deciding outcome was never broadcast. If proposers follow the rule of not starting consensus on k+1 until a majority knows the outcome for k, then this can happen at most once. It may be convenient for a new proposer to start by getting consensus on a SKIP action in order to get this complication out of the way before trying to do real work.

Further optimizations are possible in distributing the actions already decided, and handout 28 on primary copy replication describes some of them.

Note that since a state machine is completely general, one of the things it can do is to change the set of acceptors. So acceptors can be added or dropped by getting consensus among the existing acceptors. This means that no special algorithm is needed to change the set of acceptors. Of course this must be done with care, since once you have gotten consensus on a new set of acceptors you have to get a majority of *them* in order to make any further changes.

Leases

In a synchronous system, if you want to avoid running a full-blown consensus algorithm for every action that reads the state, you can instead run consensus to issue a *lease* on some component of the state. The lease guarantees that the state component won't change (unless you have an exclusive lease and change it yourself) until some expiration time, a point in real time. Thus a lease is just like a lock, except that it times out. Provided you have a clock that has a known maximum difference from real time, you can be confident that the value of a leased state component hasn't changed (that is, that you still hold the lock). To keep control of the state component (that is, to keep holding the lock), you can renew the lease before it expires. If you can't talk to all the processes that have the lease, you have to wait for it to expire before changing the leased state component. So there is a tradeoff between the cost of renewing a lease and the time you have to wait for it to expire after a (possible) failure.

There are several variations:

- If you issue the lease to some known set of processes, you can revoke it provided they all acknowledge the revocation.
- If you know the maximum transmission time for a message, you can get by with clocks that have known differences in running rate rather than known differences in absolute time.
- Since a lease is a kind of lock, it can have different lock modes, for instance, 'read' and 'write'.

The most common application is to give some set of processes the right to cache some part of the state, for instance the contents of a cache line or of a file, without having to worry about the pos-

sibility that it might change. If you don't have a lease, you have to do an expensive state machine action to get a current result; otherwise you might get an old result from some replica that isn't up to date. (Of course, you could settle for a result as of action k, rather than a current one. Then you would need neither a lease nor a state machine action, but the client has to interpret the k that it gets back along with the result it wanted. Usually this is too complicated for the client.)

If the only reason for running consensus is to issue a lease, you don't need stable state in the acceptors. If an acceptor fails, it can recover with empty state after waiting long enough that any previous lease has expired. This means that the consensus algorithm can't reliably tell you the owner of such a lease, but you don't care because it has expired anyway. Schemes for electing a proposer usually depend on this observation to avoid disk writes.

You might think that an exclusive lease allows the process that owns it to change the leased state as well, as with 'owner' access to a cache line or ownership of a multi-ported disk. This is not true in general, however, because the state is replicated, and at least a majority of the replicas have to be changed. There's no reliable way to do this without running consensus.

In spite of this, leases are not completely irrelevant to updates. With a lease you can use a simple read-write memory as an acceptor for consensus, rather than a fancier one that can do compare-and-swap, since the lease allows you do the necessary read-modify-write operation atomically under the lease's mutual exclusion. For this to work, you have to be able to bound the time that the write takes, so you can be sure that it completes before the lease expires. Actually, the requirement is weaker: a read must see the atomic effect of any write that is started earlier than the read. This ensures that if the write is started before the lease expires, a reader that starts after the lease expires will see the write.

This observation is of great practical importance, since it lets you run a replicated state machine where the acceptors are 'dual-ported' disk drives that can be accessed from more than one machine. One of the machines becomes the master by taking out a lease, and it can then write state changes to the disks.

Compare-and-swap acceptors

An alternative to using simple memory and leases is to use memory that implements a compare-and-swap or conditional store operation. The spec for compare-and-swap is

```
APROC CAS(a, old: V, new: V) -> V =
    << IF m(a) = old => m(a) := new; RET old [*] RET m(a) FI >>
```

Many machines, including the IBM 370 and the DEC Alpha, have such an operation (for the Alpha, you have to program it with Load Locked and Store Conditional, but this is easy and cheap).

To use a CAS memory as an acceptor, we have to code the state so that we can do the query and command actions as single CAS operations. Recall that the coded state of an acceptor is y = $(v_{last}, last, next)$, representing the value v_{last} for round last, no for all rounds strictly between last and next, and neutral for all rounds starting at next. A query for round n changes the state to $(v_{last}, last, n)$ provided next <= n. A command for round n changes the state to (v_n, n, n) provided next = n. So we need a representation that allows us to atomically test the current value of next and change the state in one of these ways. This is possible if an N fits in a single memory cell that CAS can read and update atomically. We can store the rest of the triple in a data structure on the side that is indexed by next. If each possible proposer

has its own piece of this data structure, they won't interfere with each other when they are updating it.

Since this is hard concurrency, the details of such a representation are tricky.

Complex updates

The actions of a state machine can be arbitrarily complex. For example, they can be complete transactions. In a replicated state machine the replicas must decide on the sequence of actions, and then each replica must do each action atomically. In handouts 19 and 20, on sequential and concurrent transactions, we will see how to make arbitrary actions atomic using redo recovery and locking. So in a general state machine each acceptor will write a redo log, in which each committed transaction corresponds to one action of the state machine. The acceptors must reach consensus on the complete sequence of actions that makes up the transaction. In practice, this means that each acceptor logs all the updates, and then they reach consensus on committing the transaction. When an acceptor recovers from a failure, it runs redo recovery in the usual way. Then it has to find out from other acceptors about any actions that they agreed on while it was down.

Of course, if several proposers are trying to run transactions at the same time, you have to make sure that the log entries don't get mixed up. Usually this is done by funneling everything through a single master called the primary; this master also acts as the proposer for consensus.

Another way of doing this is to use a single master with passive acceptors that just implement simple memory; usually these are disk drives that record redundant copies of the log. The previous section on leases explains how to run Paxos with such passive acceptors. When a master fails, the new one has to sort out the consensus on the most recent transaction as part of recovery.

Batching

Another way to avoid paying for consensus each time the state machine does an action is to batch several actions, possibly from different clients, into one super-action. They you get consensus on the super-action, and each replica can apply the individual actions of the super-action. You still have to pay to send the information that describes all the actions to each replica, but all the per-message overhead, plus the cost of the acknowledgements, is paid only once.

19. Sequential Transactions with Caching

There are many situations in which we want to make a 'big' action atomic, either with respect to concurrent execution of other actions (everyone else sees that the big action has either not started or run to completion) or with respect to failures (after a crash the big action either has not started or has run to completion).

Some examples:
Debit/credit: $x := x + \Delta$; $y := y - \Delta$
Reserve airline seat
Rename file
Allocate file and update free space information
Schedule meeting: room and six participants
Prepare report on one million accounts

Why is atomicity important? There are two main reasons:

1. *Stability*: A large persistent state is hard to fix it up if it gets corrupted. This can happen because of a system failure, or because an application fails in the middle of updating the state (presumably because of a bug). Manual fixup is impractical, and ad-hoc automatic fixup is hard to code correctly. Atomicity is a valuable automatic fixup mechanism.

2. *Consistency*: We want the state to change one big action at a time, so that between changes it is always 'consistent', that is, it always satisfies the system's invariant and always reflects exactly the effects of all the big actions that have been applied so far. This has several advantages:

   - When the server storing the state crashes, it's easy for the client to recover.
   - When the client crashes, the state remains consistent.
   - Concurrent clients always see a state that satisfies the invariant of the system. It's much easier to code the client correctly if you can count on this invariant.

The simplest way to use the atomicity of transactions is to start each transaction with no volatile state. Then there is no need for an invariant that relates the volatile state to the stable state between atomic actions. Since these invariants are the trickiest part of easy concurrency, getting rid of them is a major simplification.

Overview

In this handout we treat failures without concurrency; handout 20 treats concurrency without failures. A grand unification is possible and is left as an exercise, since the two constructions are more or less orthogonal.

We can classify failures into four levels. We show how to recover from the first three.

Transaction abort: not really a failure, in the sense that no work is lost except by request.

Crash: the volatile state is lost, but the only effect is to abort all uncommitted transactions.

Media failure: the stable state is lost, but it is recovered from the permanent log, so that the effect is the same as a crash.

Catastrophe or disaster: the stable state and the permanent log are both lost, leaving the system in an undefined state.

We begin by repeating the `SequentialTr` spec and the `LogRecovery` code from handout 7 on file systems, with some refinements. Then we give much more efficient code that allows for caching data; this is usually necessary for decent performance. Unfortunately, it complicates matters considerably. We give a rather abstract version of the caching code, and then sketch the concrete specialization that is in common use. Finally we discuss some pragmatic issues.

The spec

An `A` is an encoded action, that is, a transition from one state to another that also returns a result value. Note that an `A` is a function, that is, a deterministic transition.

```
MODULE NaiveSequentialTr [
    V,                                          % Value of an action
    S WITH { s0: ()->S }                        % State, s0 initially
    ] EXPORT Do, Commit, Crash =

TYPE A          =  S->(V, S)                    % Action

VAR  ss         := S.s0()                       % stable state
     vs         := S.s0()                       % volatile state

APROC Do(a) -> V = << VAR v | (v, vs) := a(vs); RET v >>
APROC Commit() = << ss := vs >>
APROC Crash () = << vs := ss >>                 % 'aborts' the transaction

END NaiveSequentialTr
```

Here is a simple example, with variables *X* and *Y* as the stable state, and *x* and *y* the volatile state.

Action	*X*	*Y*	*x*	*y*
	5	5	5	5
Do(*x* := *x* – 1);				
	5	5	4	5
Do(*y* := *y* + 1)				
	5	5	4	6
Commit				
	4	6	4	6
Crash before commit				
	5	5	5	5

If we want to take account of the possibility that the server (specified by this module) may fail separately from the client, then the client needs a way to detect this. Otherwise a server failure and restart after the decrement of *x* in the example could result in *X* = 5, *Y* = 6, because the client will continue with the decrement of *y* and the commit. Alternatively, if the client fails at that point, restarts, and repeats its actions, the result would be *X* = 3, *Y* = 6. To avoid these problems,

we introduce a new `Begin` action to mark the start of a transaction as `Commit` marks the end. We use another state variable `ph` (for phase) that keeps track of whether there is an uncommitted transaction in progress. A transaction in progress is aborted whenever there is a crash, or if another `Begin` action is invoked before it commits. We also introduce an `Abort` action so the client can choose to abandon a transaction explicitly.

This exported interface is slightly redundant, since `Abort = Begin; Commit`, but it's the standard way to do things. Note that `Crash = Abort` also; this is not redundant, since the client can't call `Crash`.

Note that if there's a crash after `Commit` sets `ss := vs` but before it returns to the client, there's no straightforward way for the client to know whether the transaction committed or aborted since `ph` has been set back to `idle`. This is deliberate; we don't want the spec to require that the code remember anything about the transaction indefinitely. If the client wants to know whether the transaction committed after a crash, it must put something into the state that implies the commit. For example, a funds transfer usually makes an entry in a ledger (though this is not shownin our examples).

```
MODULE SequentialTr [
    V,                                          % Value of an action
    S WITH { s0: ()->S }                        % State; s0 initially
    ] EXPORT Begin, Do, Commit, Abort, Crash =

TYPE A          =  S->(V, S)                    % Action

VAR  ss         := S.s0()                       % Stable State
     vs         := S.s0()                       % Volatile State
     ph         : ENUM[idle, run] := idle       % PHase (volatile)

EXCEPTION crashed

APROC Begin() = << Abort(); ph := run >>        % aborts any current trans.

APROC Do(a) -> V RAISES {crashed} = <<
    IF ph = run => VAR v | (v, vs) := a(vs); RET v [*] RAISE crashed FI >>

APROC Commit() RAISES {crashed} =
    << IF ph = run => ss := vs; ph := idle [*] RAISE crashed FI >>

APROC Abort () = << vs := ss; ph := idle >>     % same as Crash
APROC Crash () = << vs := ss; ph := idle >>     % 'aborts' the transaction

END SequentialTr
```

Here is the previous example extended with the `ph` variable.

Action	*X*	*Y*	*x*	*y*	ph
	5	5	5	5	idle
Begin();					
	5	5	5	5	run
Do(*x* := *x* - 1);					
	5	5	4	5	run
Do(*y* := *y* + 1)					
	5	5	4	6	run
Commit					
	4	6	4	6	idle
Crash before commit					
	5	5	5	5	idle

Uncached code

Next we give the simple uncached code based on logging; it is basically the same as the `LogRecovery` module of handout 7 on file systems, with the addition of `ph`. Note that `ss` is not the same as the `ss` of `SequentialTr`; the abstraction function gives the relation between them.

This code may seem impractical, since it makes no provision for caching the volatile state `vs`. We will study how to do this caching in general later in the handout. Here we point out that a scheme very similar to this one is used in Lightweight Recoverable Virtual Memory[1], with copy-on-write used to keep track of the differences between `vs` and `ss`.

```
MODULE LogRecovery [                                    % implements SequentialTr
    V,                                                  % Value of an action
    S0 WITH { s0: ()->S0 }                              % State
    ] EXPORT Begin, Do, Commit, Abort, Crash =

TYPE A          =  S->(V, S)                            % Action
    U           =  S -> S                               % atomic Update
    L           =  SEQ U                                % Log
    S           =  S0 WITH { "+":=DoLog }               % State with useful ops
    Ph          =  ENUM[idle, run]                      % PHase

VAR ss          := S.s0()                               % stable state
    vs          := S.s0()                               % volatile state
    sl          := L{}                                  % stable log
    vl          := L{}                                  % volatile log
    ph          := idle                                 % phase (volatile)
    rec         := false                                % recovering

EXCEPTION crashed

% ABSTRACTION to SequentialTr
    SequentialTr.ss = ss + sl
    SequentialTr.vs = (~ rec => vs [*] rec => ss + sl)
    SequentialTr.ph = ph
```

[1] M. Satyanarayanan et al., Lightweight recoverable virtual memory. *ACM Transactions on Computer Systems* **12**, 1 (Feb. 1994), pp 33-57.


```
% INVARIANT
    ~ rec ==> vs = ss + sl + vl
    (EXISTS l | l <= vl /\ ss + sl = ss + l)
    % Applying sl to ss is equivalent to applying a prefix l of vl.  That is, the
    % state after a crash will be the volatile state minus some updates at the end.

APROC Begin() = << vs := ss; sl := {}; vl := {}; ph := run >>

APROC Do(a) -> V RAISES {crashed} = <<
    IF  ph = run => VAR v, l | (v, vs + l) = a(vs) =>
        vs := vs + l; vl := vl + l; RET v
    [*] RAISE crashed
    FI >>

PROC Commit()RAISES {crashed} =
    IF ph = run => << sl := vl; ph := idle >> ; Redo() >> [*] RAISE crashed FI

PROC Abort() = << vs := ss + sl; vl := {}; ph := idle >>

PROC Crash() =
    << vs := ss; vl :={}; ph := idle; rec := true >>; % what the crash does
    vl := sl; Redo(); vs := ss; rec := false          % the recovery action

PROC Redo() =                                         % replay vl, then clear sl
% sl = vl before this is called
    DO vl # {} => << ss := ss + {vl.head} >>; << vl := vl.tail >> OD
    << sl := {} >>

FUNC DoLog(s, l) -> S =                               % s + l = DoLog(s, l)
    IF  l = {} => RET s                               % apply U's in l to s
    [*] RET DoLog((l.head)(s), l.tail))
    FI

END LogRecovery
```

Here is what this code does for the previous example, assuming for simplicity that `A = U`. You may wish to apply the abstraction function to the state at each point and check that each action simulates an action of the spec.

Action	*X*	*Y*	*x*	*y*	sl	vl	ph
	5	5	5	5	{}	{}	idle
Begin(); *Do*(*x* := *x* - 1); *Do*(*y* := *y* + 1)							
	5	5	4	6	{}	{*x*:=4; *y*:=6}	run
Commit							
	5	5	4	6	{*x*:=4; *y*:=6}	{*x*:=4; *y*:=6}	idle
Redo: *apply x*:=4							
	4	5	4	6	{*x*:=4; *y*:=6}	{*y*:=6}	idle
Redo: *apply y*:=6							
	4	6	4	6	{*x*:=4; *y*:=6}	{}	idle
Redo: *erase* sl							
	4	6	4	6	{}	{}	idle
Crash before commit							
	5	5	5	5	{}	{}	idle

Log idempotence

For this redo crash recovery to work, we need idempotence of logs: `s + l + l = s + l`, since a crash can happen during recovery. From this we get (remember that "<=" on sequences is "prefix")

```
l1 <= l ==> s + l1 + l = s + l
```

That is, `l` 'absorbs' any prefix of itself. From that it follows that repeatedly applying prefixes of `l`, followed by all of `l`, has the same effect as applying `l`; we care about this because each crash applies a prefix of `l` to `s`. For example, suppose `l = L{a;b;c;d;e}`. Then `L{a;b;c; a; a; a;b;c;d; a;b; a;b;c;d;e; a; a;b;c;d;e}` must have the same effect as `l` itself; here we have grouped the prefixes together for clarity. See handout 7 for more detail.

We can get log idempotence if the `U`'s commute and are idempotent, or if they are all writes like the assignments to *x* and *y* in the example, or writes of disk blocks. Often, however, we want to make more general updates atomic, for instance, inserting an item into a page of a B-tree. We can make general `U`'s log idempotent by attaching a unique ID to each one and recording it in `S`:

```
TYPE S          =  [ss, tags: SET UID]
     U          =  [uu: SS->SS, tag: UID] WITH { meaning:=Meaning }

FUNC Meaning(u, s)->S =
     IF  u.tag IN s.tags => RET s
     [*] RET S{ ss := (u.uu)(s.ss), tags := s.tags + {u.tag} }
     FI
```

If all the `u`'s in `l` have different tags, we get log idempotence. The way to think about this is that the modified updates have the meaning: if the tag isn't already in the state, do the original update, otherwise don't do it.

Most practical code for this makes each `U` operate on a single variable (that is, map one value to another without looking at any other part of `S`; in the usual application, a variable is one disk block). It assigns a version number `VN` to each `U` and keeps the `VN` of the most recently applied `U` with each block. Then you can tell whether a `U` has already been applied just by comparing its `VN` with the `VN` of the block. For example, if the version number of the update is `23`:

	Original	Idempotent
The disk block	`x: Int`	`x: Int` `vn: Int`

The update	`x := x + 1`	`IF vn = 22 => x := x + 1;`
		`vn := 23`
		`[*] SKIP FI`

Note: `vn = 22` implies that exactly updates `1, 2, ..., 22` have been applied.

Writing the log atomically

This code is still not practical, because it requires writing the entire log atomically in `Commit`, and the log might be bigger than the one disk block that the disk hardware writes atomically. There are various ways to get around this, but the standard one is to add a stable `sph` variable that can be `idle` or `commit`. We view `LogRecovery` as a spec for our new code, in which the `sl` of the spec is empty unless `sph = commit`. The module below includes only the parts that are different from `LogRecovery`. It changes `sl` only one update at a time. The action in the code that corresponds to `Commit` in the spec is setting `sph = commit`.


```
MODULE IncrementalLog                          % implements LogRecovery

...

VAR ...
    sph          : ENUM[idle, commit]       := idle     % stable phase
    vph          : ENUM[idle, run, commit]  := idle     % volatile phase

% ABSTRACTION to LogRecovery
    LogRecovery.sl = (sph = commit => sl  [*] {})
    LogRecovery.ph = (sph # commit => vph [*] idle)
    the identity elsewhere

APROC Begin() = << vs := ss; sl := {}; vl := {}; vph := run >>

APROC Do(a) -> V RAISES {crashed} = <<
    IF  vph = run …                            % the rest as before

PROC Commit() =
    IF vph = run =>
        % copy vl to sl a bit at a time
        VAR l := vl | DO l # {} => << sl := sl {l.head}; l := l.tail >> OD;
        << sph := commit; vph := commit >>;    % transaction commits here
        Redo()
    [*] RAISE crashed
    FI

PROC Crash() =
    << vs := ss; vl := {}; vph := idle >>;     % what the crash does
    vph := sph; vl := (vph = idle => {} [*] sl);   % the recovery
    Redo(); vs := ss                           % action

PROC Redo() =                                  % replay vl, then clear sl
    DO vl # {} => << ss := ss + {vl.head} >>; << vl := vl.tail >> OD
    DO sl # {} => << sl := sl.tail >> OD;
    << sph := idle; vph := idle >>

END IncrementalLog
```

And here is the example again.

Action	*X*	*Y*	*x*	*y*	sl	vl	sph	vph
Begin; *Do*; *Do*	5	5	4	6	{}	{*x*:=4; *y*:=6}	idle	run
Commit	5	5	4	6	{*x*:=4; *y*:=6}	{*x*:=4; *y*:=6}	commit	commit
Redo: *x*:=4; *y*:=6	4	6	4	6	{*x*:=4; *y*:=6}	{}	commit	commit
Redo: *cleanup*	4	6	4	6	{}	{}	idle	idle

We have described `sph` as a separate stable variable, but in practice each transaction is labeled with a unique transaction identifier, and `sph = commit` for a given transaction is represented by the presence of a *commit record* in the log that includes the transaction's identifier. Conversely, `sph = idle` is represented by the absence of a commit record or by the presence of a later "transaction end" record in the log. The advantages of this representation are that writing `sph` can be batched with all the other log writes, and that no storage management is needed for the `sph` variables of different transactions.

Note that you still have to be careful about the order of disk writes: all the log data must really be on the disk before `sph` is set to `commit`. This is a special case of the requirement called "write-ahead log" or 'WAL' in the database literature: everything needed to complete a transaction must be stable (that is, on the disk) before the transaction commits. For this sequential code, however, the complication caused by unordered disk writes is below the level of abstraction of our discussion.

Caching

We would like to have code for `SequentialTr` that can run fast. To this end it should:

1. Allow the volatile state `vs` to be cached so that the frequently used parts of it are fast to access, but not require a complete copy of the parts that are the same in the stable state.
2. Decouple `Commit` from actually applying the updates that have been written to the stable log; this is called *installing* the updates. Installing slows down `Commit` and therefore holds up the client, and it also does a lot of random disk writes that do not make good use of the disk. By waiting to write out changes until the main memory space is needed, we have a chance of accumulating many changes to a single disk block and paying only one disk write to install them all. We may also be able to group updates to adjacent regions of the disk.
3. Decouple crash recovery from installing updates. This is important once we have decoupled `Commit` from install, since a lot of updates can now pile up and recovery can take a long time. Also, we get it more or less for free.
4. Allow uncommitted updates to be written to the stable log, and even applied to the stable state. This saves a lot of bookkeeping to keep track of which parts of the cache go with uncommitted transactions, and it allows a transaction to make more updates than will fit in the cache.

Our new caching code has a stable state; as in `LogRecovery`, the committed state is the stable state plus the updates in the stable log. Unlike `LogRecovery`, the stable state may not include all the committed updates. `Commit` need only write the updates to the stable log, since this gets them into the abstract stable state `SequentialTR.ss`; a `Background` thread updates the concrete stable state `LogAndCache.ss`. We keep the volatile state up to date so that `Do` can return its result quickly. The price paid in performance for this scheme is that we have to reconstruct the volatile state from the stable state and the log after a crash, rather than reading it directly from the committed stable state, which no longer exists. So there's an incentive to limit the amount by which the background process runs behind.

Normally the volatile state consists of entries in the cache. Although the abstract code below does not make this explicit, the cache usually contains the most recent values of variables, that is, the values they have when all the updates have been done. Thus the stable state is updated simply by writing out variables from the cache. If the write operations write complete disk blocks, as is most often the case, it's convenient for the cached variables to be disk blocks also. If the variables are smaller, you have to read a disk block before writing it; this is called an 'installation read'. The advantage of smaller variables, of course, is that they take up less space in the cache.

The cache together with the stable state represents the volatile state. The cache is usually called a 'buffer pool' in the database literature, where these techniques originated.

We want to install parts of the cache to the stable state independently of what is committed (for a processor cache, install is usually called 'flush', and for a file system cache it is usually called 'sync'). Otherwise we might run out of cache space if there are transactions that don't commit for a long time. Even if all transactions are short, a popular part of the cache might always be touched by a transaction that hasn't yet committed, so we couldn't install it and therefore couldn't truncate the log. Thus the stable state may run *ahead* of the committed state as well as behind. This means that the stable log must include "undo" operations that can be used to reverse the installed but uncommitted updates in case the transaction aborts instead of committing. In order to keep undoing simple when the abort is caused by a crash, we arrange things so that before applying an undo, we use the stable log to completely do the action that is being undone. Hence an undo is always applied to an "action consistent" state, and we don't have to worry about the interaction between an undo and the smaller atomic updates that together comprise the action. To implement this rule we need to add an action's updates and its undo to the log atomically.

To be sure that we can abort a transaction after installing some parts of the cache to the stable state, we have to follow the "write ahead log" or WAL rule, which says that before a cache entry can be installed, all the actions that affected that entry (and therefore all their undo's) must be in the stable log.

Although we don't want to be forced to keep the stable state up with the log, we do want to discard old log entries after they have been installed, whether or not the transaction has committed, so the log space can be reused. Of course, log entries for undo's can't be discarded until `Commit`.

Finally, we want to be able to keep discarded log entries forever in a "permanent log" so that we can recover the stable state in case it is corrupted by a media failure. The permanent log is usually kept on magnetic tape.

Here is a summary of our requirements:

> Cache that can be installed independently of locking or commits.
>
> Crash recovery (or 'redo') log that can be truncated.
>
> Separate undo log to simplify truncating the crash recovery log.
>
> Complete permanent log for media recovery.

The `LogAndCache` code below is a full description of a practical transaction system, except that it doesn't deal with concurrency (see handout 20) or with distributing the transaction among multiple `SequentialTr` modules (see handout 27). The strategy is to:

- Factor the state into independent components, for example, disk blocks.
- Factor the actions into log updates called `U`'s and cache updates called `W`'s. Each cache update not only is atomic but works on only one state component. Cache updates for different components commute. Log updates do not need either of these properties.
- Define an *undo* action for each action (not each update). The action followed by its undo leaves the state unchanged. An undo action is a full-fledged action, so it may itself involve multiple updates.

- Keep separate log and undo log, both stable and volatile.

  Log : sequence of updates

  UndoLog : sequence of undo actions (not updates)

The essential step is installing a cache update into the stable state. This is an internal action, so it must not change the abstract stable or volatile state. As we shall see, there are many ways to satisfy this requirement.

The code in `LogAndCache` is rather abstract. We give the usual concrete form in `BufferPool` below. Here `W` (a cached update) is just `s(ba) := d` (that is, set the contents of a block of the stable state to a new value). This is classical caching, and it may be helpful to bear it in mind as concrete code for these ideas, which is worked out in detail in `BufferPool`. Note that this kind of caching has another important property: we can get the current value of `s(ba)` from the cache. This property isn't essential for correctness, but it certainly makes it easier for `Do` to be fast.

```
MODULE LogAndCache [                                  % implements SequentialTr
    V,                                                % Value of an action
    S0 WITH {s0:=()->S0}                              % abstract State; s0 initially
    ] = EXPORT Begin, Do, Commit, Abort, Crash =

TYPE A          =  S->(V, S)                          % Action
    S           =  S0 WITH {"+":=DoLog,               % State with ops
                            "++":= DoCache,
                            "-" := UndoLog}
    Tag         =  ENUM[commit]
    U           =  S -> S                             % Update
    Un          =  (A + ENUM[cancel])                 % Undo
    W           =  S -> S                             % update in cache
    L           =  SEQ (SEQ U + Tag)                  % Log
    UL          =  SEQ Un                             % Undo Log
    C           =  SET W WITH {"++":=CombineCache}    % Cache
    Ph          =  ENUM[idle, run]                    % Phase

VAR ss          := S.s0()                             % Stable State

    sl          := L{}                                % Stable Log
    sul         := UL{}                               % Stable Undo Log

    c           :  C := {}                            % Cache (dirty part)
    vl          := L{}                                % Volatile Log
    vul         := UL{}                               % Volatile Undo Log

    vph         := idle                               % Volatile PHase

    pl          := L{}                                % Permanent Log

    undoing     := false                              % true during recovery; just for
                                                        the abstraction function
```

Note that there are two logs, called `L` and `UL` (for undo log). A `L` records groups of updates; the difference between an update `U` and an action `A` is that an action can be an arbitrary state change, while an update must interact properly with a cache update `W`. To achieve this, a single action must in general be broken down into several updates. All the updates from one action form one group in the log. The reason for grouping the updates of an action is that, as we have seen, we

have to add all the updates of an action, together with its undo, to the stable log atomically. There are various ways to represent a group, for example as a sequence of updates delimited by a special `mark` token that is interpreted much like a commit record for the action, but we omit these details.

A cache update `W` must be applied atomically to the stable state. For example, if the stable state is just raw disk pages, the only atomic operation is to write a single disk page, so an update must be small enough that it changes only one page.

`UL` records undo actions `Un` that reverse the effect of actions. These are actions, not updates; a `Un` is converted into a suitable sequence of `U`'s when it is applied. When we apply an undo, we treat it like any other action, except that it doesn't need an undo itself; this is because we only apply undo's to abort a transaction, and we never change our minds about aborting a transaction. We do need to ensure either that each undo is applied only once, or that undo actions have log idempotence. Since we don't require log idempotence for ordinary actions (only for updates), it's unpleasant to require it for undo's. Instead, we arrange to remove each undo action from the undo log atomically with applying it. We code this idea with a special `Un` called `cancel` that means: don't apply the next earlier `Un` in the log that hasn't already been canceled, and we write a `cancel` to `vul/sul` atomically as we write the updates of the undo action to `vl/sl`. For example, after `un1`, `un2`, and `un3` have been processed, `ul` might be

```
    un0 un1 un2 cancel un3 cancel cancel
= un0 un1 un2 cancel cancel
= un0 un1 cancel
= un0
```

Of course many other encodings are possible, but this one is simple and fits in well with the rest of the code.

Examples of actions:

```
f(x) := y                                  % simple overwriting
f(x) := f(x) + y                           % not idempotent
f := f{x -> }                              % delete
split B-tree node
```

This code has commit records in the log rather than a separate `sph` variable as in `IncrementalLog`. This makes it easier to have multiple transactions in the stable log. For brevity we omit the machinations with `sph`.

Here is a summary of the requirements we have derived on `U`, `Un`, and `W`:

- We can atomically add to `sl` both all the `U`'s of an action and the action's undo (`ForceOne` does this).
- Applying a `W` to `ss` is atomic (`Install` does this).
- Applying a `W` to `ss` doesn't change the abstract `ss` or `vs`. This is a key property that replaces the log idempotence of `LogRecovery`.
- A `W` looks only at a small part of `s` when it is applied, normally only one component (`DoCache` does this).
- Mapping `U` to `W` is cheap and requires looking only at a small part (normally only one component) of `ss`, at least most of the time (`Apply` does this).

- All W's in the cache commute. This ensures that we can install cache entries in any order (CombineCache assures this).

```
% ABSTRACTION to SequentialTr
    SequentialTr.ss = ss + sl - sul
    SequentialTr.vs = (~undoing => ss + sl + vl [*] ss + (sl+vl)-(sul+vul)
    SequentialTr.ph = (~undoing => vph idle)

% INVARIANTS
    [1] (ALL l1, l2 |     sl = l1 + {commit} + l2     % Stable undos cancel
                       /\ ~ commit IN l2              % uncommitted tail of sl
                      ==> ss + l1 = ss + sl - sul )
    [2] ss + sl = ss + sl + vl - vul                  % Volatile undos cancel vl
    [3] ~ undoing ==> ss + sl + vl = ss ++ c          % Cache is correct; this is vs
    [4] (ALL w1 :IN c, w2 :IN c |                     % All W's in c commute
           w1 * w2 = w2 * w1).
    [5] S.s0() + pl + sl - sul = ss                   % Permanent log is complete
    [6] (ALL w :IN c |  ss ++ {w} + sl                % Any cache entry can be installed
                      = ss        + sl
```

% External interface

```
PROC Begin() = << IF vph = run => Abort() [*] SKIP FI; vph := run >>

PROC Do(a) -> V RAISES {crashed} = <<
    IF   vph = run => VAR v | v := Apply(a, AToUn(a, ss ++ c)); RET v
    [*]  RAISE crashed
    FI >>

PROC Commit() RAISES {crashed} =
    IF   vph = run => ForceAll(); << sl := sl + {commit}; sul := {}; vph := idle
    [*]  RAISE crashed
    FI

PROC Abort () = undoing := true; Undo(); vph := idle

PROC Checkpoint() = VAR c' := c, w |                  % move sl + vl to pl
    DO c' # {} => w := Install(); c' := c' - {w} OD;  % until everything in c' is installed
    Truncate()

PROC Crash() =
    << vl := {}; vul := {}; c := {}; undoing := true >>;
    Redo(); Undo(); vph := idle
```

% Internal procedures invoked from Do and Commit

```
APROC Apply(a, un) -> V = <<                          % called by Do and Undo
    VAR v, l, vs := ss ++ c |
        (v, l) := AToL(a, vs);                        % find U's that do a
        vl := vl + l; vul := vul + {un};
        VAR c' | ss ++ c ++ c' = ss ++ c + l          % Find w's for action a
            => c := c ++ c';
        RET v >>

PROC ForceAll() = DO vl # {} => ForceOne() OD; RET    % more all of vl to sl

APROC ForceOne() = << VAR l1, l2 |                    % move one a from vl to sl
    sl := sl + {vl.head}; sul := sul + {vul.head};
    vl := vl.tail; vul := vul.tail >>
```


% Internal procedures invoked from Crash or Abort

```
PROC Redo() = VAR l := + : sl |                       % Restore c from sl after crash
    DO << l # {} => VAR vs := ss ++ c, w |            % Find w for each u in l
        vs ++ {w} = vs + L{{l.head}} => c := c ++ {w}; l := l.tail >>
    OD

PROC Undo() =                                         % Apply sul + vul to vs
    VAR ul := sul + vul, i := 0 |
        DO ul # {} => VAR un := ul.last |
            ul := ul.reml;
            IF un=cancel => i := i+1
               [*] i>0 => i := i-1
               [*] Apply(un, cancel) FI
        OD; undoing := false
    % Every entry in sul + vul has a cancel, and everything is undone in vs.
```

% Background actions to install updates from c to ss and to truncate sl

```
THREAD Background() =
    DO
        Install()
    []  ForceOne()                                    % Enough of these implement WAL
    []  Drop()
    []  Truncate()
    []  SKIP
    OD

APROC Install() -> W = << VAR w :IN c |               % Apply some w to ss; requires WAL
       ss # ss ++ {w}                                 % no point if w is already in ss
    /\ ss ++ {w} + sl = ss + sl =>                    % w is in sl, the WAL condition
        ss := ss ++ {w}; RET w >>

APROC Drop() = << VAR w :IN c | ss ++ {w} = ss => c := c - {w} >>

APROC Truncate() = << VAR l1, l2 |                    % Move some of sl to pl
    sl = l1 + l2 /\ ss + l2 = ss + sl => pl := pl + l1; sl := l2 >>
```

% Media recovery

The idea is to reconstruct the stable state ss from the permanent log pl by redoing all the updates, starting with a fixed initial ss. Details are left as an exercise.

% Miscellaneous functions

```
FUNC AToL(a, s) -> (V, L) = VAR v, l |                % all U's in one group
    l.size = 1 /\ (v, s + l) = a(s) => RET (v, l)

FUNC AToUn(a, s) -> Un = VAR un, v, s' |
    (v, s') = a(s) /\ (nil, s) = un(s') => RET un
```

The remaining functions are only used in guards and invariants.

```
FUNC DoLog(s, l) -> S =                               % s + l = DoLog(s, l)
    IF   l = {} => RET s                              % apply U's in l to s
    [*]  VAR us := l.head |
        RET DoLog((    us IS Tag \/ us = {} => s
                   [*] (us.head)(s), {us.tail} + l.tail))
    FI
```

```
FUNC DoCache(s, c) -> S =                                  % s ++ c = DoCache(s, c)
    DO VAR w :IN c | s := w(s), c := c - {w} OD; RET s

FUNC UndoLog(s, ul) -> S =                                 % s - l = UndoLog(s, l)
    IF   ul = {} => RET s
    []   ul.last # cancel => RET UndoLog((u.last)(s), ul.reml)
    []   VAR ul1, un, ul2 | un # cancel /\ ul = ul1 + {un; cancel} + ul2 =>
         RET UndoLog(s, ul1 + ul2)
    FI
```

A cache is a set of commuting update functions. When we combine two caches `c1` and `c2`, as we do in `apply`, we want the total effect of `c1` and `c2`, and all the updates still have to commute and be atomic updates. The `CombineCache` below just states this requirement, without saying how to achieve it. Usually it's done by requiring updates that don't commute to compose into a single update that is still atomic. In the usual case updates are writes of single variables, which do have this property, since `u1 * u2 = u2` if both are writes of the same variable.

```
FUNC CombineCache(c1, c2) -> C =                           % c1++c2 = CombineCache(c1,c2)
    VAR c |   (* : c.seq) = (* : c1.seq) * (* : c2.seq)
          /\ (ALL w1 :IN c, w2 :IN c | w1 # w2 ==> w1 * w2 = w2 * w1) => RET c

END LogAndCache
```

We can summarize the ideas in `LogAndCache`:

- Writing stable state before committing a transaction requires undo. We need to write before committing because cache space for changed state is limited, while the size of a transaction may not be limited, and also to avoid starvation that keeps us from installing some cache entries that are always part of an uncommitted transaction.
- Every uncommitted log entry has a logged undo. The entry and the undo are made stable by a single atomic action (using some low-level coding trick that we leave as an exercise for the reader). We must log an action and its undo before installing a cache entry affected by it; this is write-ahead logging.
- Recovery is complete redo followed by undo of uncommitted transactions. Because of the complete redo, undo's are always from a clean state, and hence can be actions.
- An undo is executed like a regular action, that is, logged. The undo of the undo is a special `cancel` action.
- Writing a `W` to stable state doesn't change the abstract stable state. This means that redo recovery works. It's a strong constraint on the relation between logged `U`'s and cached `W`'s.

Actually, the write-ahead rule is stronger than what this code requires. The only requirement in the code for installing a `W` is that it does not affect the abstract stable state; this is what the crucial boxed part of the guard on `Install` says. Put another way, if `{w} + sl = sl` then `sl` *absorbs* `w` and it's OK to install `w`. If `w` is an update in `sl` and the updates are log idempotent this will definitely be true, and this is the usual implementation, but the necessary condition is only that `sl` absorbs `w`, and indeed only that it does so in the current state of `ss`.

This makes it clear why we need the flexibility to decompose an action into updates, rather than just logging the action itself: an arbitrary action that reads from the state is more likely to be affected by the state changes caused by installing a `W`. The best case, from this point of view, is an

update with no dependencies, that is, a "blind" write of a constant into a variable. If `sl` contains a blind write of `x`, then it will absorb any `W` that just writes `x`. By contrast, the action

```
VAR t := x | x := y; y := t
```

that swaps `x` and `y` does not absorb any write of `x` or `y` (except one that leaves the current value unchanged), so logging such an action might make it impossible to install its changes. Lomet and Tuttle discuss these issues in detail.[2]

Multi-level recovery

Although in our examples an update is usually a write of one disk block, the `LogAndCache` code works on top of any kind of atomic update, for example a B-tree or even another transactional system. The latter case codes each `W` as a transaction in terms of updates at a still lower level. Of course this idea can be applied recursively to yield a *n* level system. This is called 'multi-level recovery'.[3] It's possible to make a multi-level system more efficient by merging the logs from several levels into a single sequential structure.

Why would you want to complicate things in this way instead of just using a single-level transaction, which would have the same atomicity? There are at least two reasons:

- The lower level may already exist in a form that you can't change, and it may lack the necessary externally accessible locking or commit operations. A simple example is a file system, which typically provides atomic file renaming, on which you can build something more general. Or you might have several existing database systems, on top of which you want to build transactions that change more than one database. We show in handout 27 how to do this using two-phase commit. But if the existing systems don't implement two-phase commit, you can still build a multi-level system on them.
- Often you can get more concurrency by allowing lower level transactions to commit and release their locks. For example, a B-tree typically holds locks on whole disk blocks to maintain the integrity of the index data structures, but at the higher level only individual entries need to be locked.

Buffer pools

The standard code for the ideas in `LogAndCache` makes a `U` and a `W` read and write a single block of data. The `W` just gives the current value of the block, and the `U` maps one such block into another. Both `W`'s (that is, cache blocks) and `U`'s carry sequence numbers so that we can get the log idempotence property without restricting the kind of mapping that a `U` does, using the method described earlier; these are called 'log sequence numbers' or LSN's in the database literature.

The LSN's are also used to code the WAL guard in `Install` and the guard in `Truncate`. It's OK to install a `W` if the LSN of the last entry in `sl` is at least as big as the n of the `W`. It's OK to drop a `U` from the front of `sl` if every uninstalled `W` in the cache has a bigger LSN.

The simplest case is a block equal to a single disk block, for which we have an atomic write. Often a block is chosen to be several disk blocks, to reduce the size of indexes and improve the efficiency of disk transfers. In this case care must be taken that the write is still atomic; many commercial systems get this wrong.

[2] Lomet, D. and Tuttle, M. A Theory of Redo Recovery. *SIGMOD Conference*, San Diego, CA (June 2003) 397-406
[3] D. Lomet. MLR: A recovery method for multi-level systems. *Proc. SIGMOD Conf.*, May, 1992, pp 185-194.

The following module is incomplete.

```
MODULE BufferPool [                                  % implements LogAndCache
    V,                                               % Value of an action
    S0 WITH {s0:=()->S0},                            % abstract State
    Data
    ] EXPORT ... =

TYPE A          =  S->(V, S)                         % Action
    SN          =  Int                               % Sequence Number
    BA          =  Int                               % Block Address
    LB          =  [sn, data]                        % Logical Block

    S           =  BA -> LB                          % State
    U           =  [sn, ba, f: LB->LB]               % Update
    W           =  [ba, lb]                          % update in cache

    Un          =  A
    L           =  SEQ U
    UL          =  SEQ Un
    C           =  SET W

VAR ss          := S.s0()                            % Stable State

FUNC vs(ba) -> LB = VAR w IN c | w.ba = ba => RET w.lb [*] RET ss(ba)
% This is the standard abstraction function for a cache
```

The essential property is that updates to different blocks commute: `w.ba # u.ba ==> w` commutes with `u`, because `u` only looks at `u.ba`. Stated precisely:

```
(ALL s |    (ALL ba | ba # u.ba ==> u(s)(ba) = s(ba)
        /\ (ALL s' | s(u.ba) = s'(u.ba) ==> u(s)(u.ba) = u(s')(u.ba)) )
```

So the guard in `Install` testing whether `w` is already installed is just

```
    (EXISTS u | u IN vl /\ u.ba = w.a)
```

because in `Do` we get `w` as `W{ba:=u.ba, lb:=u(vs)(u.ba)}`.

```
END BufferPool
```

Transactions meet the real world

Various problems arise in using the transaction abstraction we have been discussing to code actual transactions such as ATM withdrawals or airline reservations. We mention the most important ones briefly.

The most serious problems arise when a transaction includes changes to state that is not completely under the computer's control. An ATM machine, for example, dispenses cash; once this has been done, there's no straightforward way for a program to take back the cash, or even to be certain that the cash was actually dispensed. So neither undo nor log idempotence may be possible. Changing the state of a disk block has neither of these problems.

So the first question is: *Did it get done*? The jargon for this is "testability". Carefully engineered systems do as much as possible to provide the computer with feedback about what happened in the real world, whether it's dispensing money, printing a check, or unlocking a door. This means having sensors that are independent of actuators and have a high probability of being able to detect whether or not an intended physical state transition occurred. It also means designing the

computer-device interface so that after a crash the computer can test whether the device received and performed a particular action; hence the name "testability".

The second question is: *Is undo possible*? The jargon for this is "compensation". Carefully engineered systems include methods for undoing at least the important effects of changes in the state of the world. This might involve executing some kind of compensating transaction, for instance, to reduce the size of the next order if too much is sent in the current one, or to issue a stop payment order for a check that was printed erroneously. Or it might require manual intervention, for instance, calling up the bank customer and asking what really happened in yesterday's ATM transaction. Usually compensation is not perfect.

Because compensation is complicated and imperfect, the first line of defense against the problems of undo is to minimize the probability that a transaction involving real-world state changes will have to abort. To do this, break it into two transactions. The first runs entirely inside the computer, and it makes all the internal state changes as well as posting instructions for the external state changes that are required. The second is as simple as possible; it just executes the posted external changes. Often the second transaction is thought of as a message system that is responsible for reliably delivering an action message to a physical device, and also for using the testability features to ensure that the action is taken exactly once.

The other major difficulty in transactions that interact with the world arises only with concurrent transactions. It has to do with input: if the transaction requires a round trip from the computer to a user and back it might take a long time, because users are slow and easily distracted. For example, a reservation transaction might accept a travel request, display flight availability, and ask the user to choose a flight. If the transaction is supposed to be atomic, seats on the displayed flights must remain available until the user makes her choice, and hence can't be sold to other customers. To avoid these problems, systems usually insist that a single transaction begin with user input, end with user output, and involve no other interactions with the user. So the reservation example would be broken into two transactions, one inquiring about flight availability and the other attempting to reserve a seat. Handout 20 on concurrent transactions discusses this issue in more detail.

Transactions as fairy dust

Atomicity in spite of faults is only one aspect of transactions. Atomicity in spite of concurrency is another aspect; it is the subject of the next handout. A third aspect is load balancing: when many transactions run against shared state stored in a database, there is a lot of freedom in allocating CPU, memory and communications resources to them.

A complete transaction processing system puts all three aspects together, and the result is something that is unique in computing: you can start with a collection of straightforward sequential programs that are written without any concern for fault-tolerance, concurrency, or scheduling, sprinkle transaction processing fairy dust on them, and automatically obtain a fault-tolerant, concurrent program that runs efficiently on a large collection of processors, memory, and disks. Nowhere else do we know how to spin straw into gold in this way.

20. Concurrent Transactions

We often (usually?) want more from a transaction mechanism than atomicity in the presence of failures: we also want atomicity in the presence of concurrency. As we saw in handout 14 on practical concurrency, the reasons for wanting to run transactions concurrently are slow input/output devices (usually disks) and the presence of multiple processors on which different transactions can run at the same time. The latter is especially important because it is a way of taking advantage of multiple processors that doesn't require any special programming. In a distributed system it is also important because separate nodes need autonomy.

Informally, if there are two transactions in progress concurrently (that is, the `Begin` of one happens between the `Begin` and the `Commit` of the other), we want all the observable effects to be as though all the actions of one transaction happen either before or after all the actions of the other. This is called *serializing* the two transactions; it is the same as making each transaction into an atomic action. This is good for the usual reason: clients can reason about each transaction separately as a sequential program. Clients only have to worry about concurrency in between transactions, and they can use the usual method for doing this: find invariants that each transaction establishes when it commits and can therefore assume when it begins. The simplest way to ensure that your program doesn't depend on anything except the invariants is to discard all state at the end of a transaction, and re-read whatever you need after starting the next transaction.

Here is the standard example. We are maintaining bank balances, with `deposit`, `withdraw`, and `balance` transactions. The first two involve reading the current balance, adding or subtracting something, making a test, and perhaps writing the new balance back. If the read and write are the largest atomic actions, then the sequence `read1`, `read2`, `write1`, `write2` will result in losing the effect of transaction 1. The third reads lots of balances and expects their total to be a constant. If its reads are interleaved with the writes of the other transactions, it may get the wrong total.

The other property we want is that if one transaction precedes another (that is, its `Commit` happens before the `Begin` of the other, so that their execution does not overlap) then it is serialized first. This is sometimes called *external consistency*; it's not just a picky detail that only a theoretician would worry about, because it's needed to ensure that when you put two transaction systems together you still get a serializable system.

A piece of jargon you will sometimes see is that transactions have the ACID properties: Atomic, Consistent, Isolated, and Durable. Here are the definitions given in Gray and Reuter:

> **Atomicity**. A transaction's changes to the state are atomic: either all happen or none happen. These changes include database changes, messages, and actions on transducers.
>
> **Consistency**. A transaction is a correct transformation of the state. The actions taken as a group do not violate any of the integrity constraints associated with the state. This requires that the transaction be a correct program.
>
> **Isolation**. Even though transactions execute concurrently, it appears to each transaction T that others executed either before T or after T, but not both.
>
> **Durability**. Once a transaction completes successfully (commits), its changes to the state survive failures.

The first three appear to be different ways of saying that all transactions are serializable. It's important to understand, however, that the transaction system by itself can't ensure consistency. Each transaction taken by itself must maintain the integrity constraints, that is, the invariant. Then the transaction system guarantees that all the transactions executing concurrently still maintain the invariant.

Many systems implement something weaker than serializability for committed transactions in order to allow more concurrency. The standard terminology for weaker degrees of isolation is degree 0 through degree 3, which is serializability. Gray and Reuter discuss the specs, code, advantages, and drawbacks of weaker isolation in detail (section 7.6, pages 397-419).

We give a spec for concurrent transactions. Coding this spec is called 'concurrency control', and we briefly discuss a number of coding techniques.

Spec

The spec is written in terms of the *histories* of the transactions: a history is a sequence of (action, result) pairs, called *events* below. The order of the events for a single transaction is fixed: it is the order in which the transaction did the actions. The spec says that there is a total ordering of all the committed transactions that has three properties:

> *Serializable*: Doing the actions in the total order that is consistent both with the total order of the transactions and with the order of actions in each transaction (starting from the initial state) yields the same result from each action, and the same final state, as the results and final state actually obtained.
>
> *Externally consistent*: The total order is consistent with the partial order established by the `Begin`'s and `Commit`'s.
>
> *Non-blocking*: it's always possible to abort a transaction. This is necessary because when there's a crash all the active transactions must abort.

This is all that most transaction specs say. It allows anything to happen for uncommitted transactions. Operationally, this means that an uncommitted transaction will have to abort if it has seen a result that isn't consistent with any ordering of it and the committed transactions. It also means that the programmer has to expect completely arbitrary results to come back from the actions. In theory this is OK, since a transaction that gets bad results will not be allowed to commit, and hence nothing that it does can affect the rest of the world. But in practice this is not very satisfactory, since programs that get crazy results may loop, crash, or otherwise behave badly in ways that are beyond the scope of the transaction system to control. So our spec imposes some constraints on how actions can behave even before they commit.

The spec works by keeping track of:

- The ordering requirements imposed by external consistency, in a relation `xc`.
- The histories of the transactions, in a map `y`.

It imposes an invariant on `xc` and `y` that is defined by the function `Invariant`. This function says that the committed transactions have to be serializable in a way consistent with `xc`, and that something must be true for the active transactions. As written, `Invariant` offers a choice of several "somethings"; the intuitive meaning of each one is described in a comment after its defini-

tion. The Do and Commit routines block if they can't find a way to satisfy the invariant. The invariant maintained by the system is Invariant(committed, active, xc, y).

It's unfortunate that this spec deals explicitly with the histories of the transactions. Normally our specs don't do this, but instead give a state machine that only generates allowable histories. I don't know any way to do this for the most general serializability spec.

The function Invariant defining the main invariant appears after the other routines of the spec.

```
MODULE ConcurrentTransactions [
    V,                                                  % Value
    S,                                                  % State of database
    T                                                   % Transaction ID
    ] EXPORT Begin, Do, Commit, Abort, Crash =

TYPE Result     =  [v, s]
    A           =  S -> Result                          % Action
    E           =  [a, v]                               % Event: Action and result Value
    H           =  SEQ E                                % History

    TS          =  SET T                                % Transaction Set
    XC          =  (T, T)->Bool                         % eXternal Consistency; the first
                                                        % transaction precedes the second
    TO          =  SEQ T SUCHTHAT to.size=to.set.size   % Total Order on T's
    Y           =  T -> H                               % histories of transactions

VAR s0          :  S                                    % current base state
    y           := Y{}                                  % current transaction histories
    xc          := XC{* -> false}                       % current required XC

    active      :  TS{}                                 % active transactions
    committed   :  TS{}                                 % committed transactions
    installed   :  TS{}                                 % installed transactions
    aborted     :  TS{}                                 % aborted transactions
```

The sets installed and aborted are only for the benefit of careless clients; they ensure that T's will not be reused and that Commit and Abort can be repeated without raising an exception.

Operations on histories and orderings

To define Serializable we need some machinery. A history h records a sequence of events, that is, actions and the values they return. Apply applies a history to a state to compute a new state; note that it's undefined (because the body fails) if the actions in the history don't give back the results in the history. Valid checks whether applying the histories of the transactions in a given total order can succeed, that is, yield the values that the histories record. Consistent checks that a total order is consistent with a partial order, using the closure method (see section 9 of the Spec reference manual) to get the transitive closure of the external consistency relation and the <<= method for non-contiguous sub-sequence. Then Serializable(ts, xc, y) is true if there is some total order to on the transactions in the set ts that is consistent with xc and that makes the histories in y valid.

```
FUNC Apply(h, s) -> S =
    % return the end of the sequence of states starting at s and generated by
    % h's actions, provided the actions yield h's values. Otherwise undefined.
    RET {e :IN h, s' := s BY (e.a(s').v = e.v => e.a(s').s)}.last
```



```
FUNC Valid(y0, to) -> BOOL = RET Apply!( + : (to * y0), s0)
```
% the histories in y0 in the order defined by to are valid starting at s0. Recall that f!x is true if f is defined at x.

```
FUNC Consistent(to, xc0) -> BOOL =
    RET xc0.closure.set <= (\ t1, t2 | TO{t1, t2} <<= to).set

FUNC Serializable(ts, xc0, y0) -> BOOL =                % is there a good TO of ts
    RET ( EXISTS to :IN ts.perms | Consistent(to, xc0) /\ Valid(y0, to) )
```

Interface procedures

A transaction is identified by a *transaction identifier* t, which is assigned by Begin and passed as an argument to all the other interface procedures. Do finds a result for the action a that satisfies the invariant; if this isn't possible the Do can't occur, that is, the transaction issuing it must abort or block. For instance, if concurrency control is coded with locks, the issuing transaction will wait for a lock. Similarly, Commit checks that the transaction is serializable with all the already committed transactions. Abort never blocks, although it may have to abort several transactions in order to preserve the invariant; this is called "cascading aborts" and is usually considered to be bad, for obvious reasons.

Note that Do and Commit block rather than failing if they can't maintain the invariant. They may be able to proceed later, after other transactions have committed. But some code can get stuck (for example, the optimistic schemes described later), and for these there must be a demon thread that aborts a stuck transaction.

```
APROC Begin() -> T =
% Choose a t and make it later in xc than every committed trans; can't block
    << VAR t | ~ t IN active \/ committed \/ installed \/ aborted =>
        y(t) := {}; active \/ := {t}; xc(t, t) := true;
        DO VAR t' :IN committed | ~ xc.closure(t', t) => xc(t', t) := true OD;
        RET t >>

APROC Do(t, a) -> V RAISES {badT} =
% Add (a,v) to history; may block unless NC
    << IF  ~ t IN active => RAISE badT
       [*] VAR v, y' |
             y'(t) + := {E{a, v};
             Invariant(committed, active, xc, y') => y := y'; RET v >>

APROC Commit(t) RAISES {badT} = <<                      % may block unless AC (for invariant)
    IF  t IN committed \/ installed => SKIP             % repeating Commit is OK
    []  ~ t IN active \/ committed \/ installed => RAISE badT >>
    []  t IN active /\ Invariant(committed \/ {t}, active - {t}, xc, y) =>
        committed \/ := {t}; active - := {t} >>

APROC Abort(t) RAISES {badT} = <<                       % doesn't block (need this for crashes)
    IF  t IN aborted => SKIP                            % repeating Abort is OK
    []  t IN active  =>
        % Abort t, and as few others as possible to maintain the invariant.
        % s is the possible sets of T's to abort; choose one of the smallest ones.
        VAR s      := {ts |   {t} <= ts /\ ts <= active
                              /\ Invariant(committed, active - ts, xc, y)},
            n      := {ts :IN s || ts.size}.min,
            aborts := {ts :IN s | ts.size = n}.choose |
                aborted \/ := aborts; active - := aborts;
                y := y{t->}
    [*] RAISE badT
    FI >>
```

Installation daemon

This is not really part of the spec, but it is included to show how the data structures can be cleaned up.

```
THREAD Install() = DO                                   % install a committed transaction in s0
    << VAR t |
         t IN committed
         % only if there's no other transaction that should be earlier
       /\ ( ALL t' :IN committed \/ active | xc(t , t') ) =>
              s0 := Apply(y(t), s0);
              committed - := {t}; installed \/ := {t}
              % remove t from y and xc; this isn't necessary, but it's tidy
              y := y{t -> };
              DO VAR t' | xc(t , t') => xc := xc{(t , t') -> } OD;
    >>
    [*] SKIP OD
```

Function defining the main invariant

```
FUNC Invariant(com: TS, act: TS, xc0, y0) -> BOOL = VAR current := com + act |
        Serializable(com, xc0, y0)
    /\  % constraints on active transactions: choose ONE
```

AC
```
      (ALL t :IN act |                      Serializable(com + {t}, xc0, y0) )
```

CC
```
                                            Serializable(com + act, xc0, y0)
```

EO
```
      (ALL t :IN act | (EXISTS ts |
          com       <=ts /\ ts<=current /\  Serializable(ts  + {t}, xc0, y0) )
```

OD
```
      (ALL t :IN act | (EXISTS ts |
          AtBegin(t)<=ts /\ ts<=current /\  Serializable(ts  + {t}, xc0, y0) )
```

OC1
```
      (ALL t :IN act, h :IN Prefixes(y0(t)) | (EXISTS to, h1, h2 |
           to.set = AtBegin(t) /\ Consistent(to, xc0) /\ Valid(y0, to)
        /\ IsInterleaving(h1, {t' | t' IN current-AtBegin(t)-{t} || y0(t')})
        /\ h2 <<= h1      % subsequence
        /\ h.last.a(Apply(+ : (to * y0) + h2 + h.reml, s0) = h.last.v ))
```

OC2
```
      (ALL t :IN act, h :IN Prefixes(y0(t)) | (EXISTS to, h1, h2, h3 |
           to.set = AtBegin(t) /\ Consistent(to, xc0) /\ Valid(y0, to)
        /\ IsInterleaving(h1, {t' | t' IN current-AtBegin(t)-{t} || y0(t')})
        /\ h2 <<= h1      % subsequence
        /\ IsInterleaving(h3, {h2; h.reml})
        /\ h.last.a(Apply(+ : (to * y0) + h3, s0) = h.last.v ))
```

NC
```
      true
```

```
FUNC Prefixes(h) -> SET H = RET {h' | h' <= h /\ h' # {}}

FUNC AtBegin(t) -> TS = RET {t' | xc.closure(t', t)}
% The transactions that are committed when t begins.

FUNC IsInterleaving(h, s: SET H) -> BOOL =
% h is an interleaving of the histories in s. This is true if there's amultiset il that partitions h.dom, and
% each element of il extracts one of the histories in s from h
    RET (EXISTS il: SEQ SEQ Int |
         (+ : il) == h.dom.seq /\ {z :IN il || z * h} == s.seq )
```


A set of transactions is serializable if there is a serialization for all of them. All versions of the invariant require the committed transactions to be serializable; hence a transaction can only commit if it is serializable with all the already committed transactions. There are different ideas about the uncommitted ones. Some ideas use `AtBegin(t)`: the transactions that committed before `t` started.

AC All Committable: every uncommitted transaction can commit now and `AC` still holds for the rest (implies that any subset of the uncommitted transactions can commit, since abort is always possible). Strict two-phase locking, which doesn't release any locks until commit, ensures `AC`.

CC Complete Commit: it's possible for all the transactions to commit (i.e., there's at least one that can commit and `CC` still holds for the rest). Two-phase locking, which doesn't acquire any locks after releasing one, ensures `CC`. `AC ==> CC`.

EO Equal Opportunity: every uncommitted transaction has some friends such that it can commit if they do. `CC ==> EO`.

OD Orphan Detection: every uncommitted transaction is serializable with its `AtBegin` plus some other transactions (a variation not given here restricts it to the `AtBegin` plus some other committed transactions). It may not be able to commit because it may not be serializable with all the committed transactions; a transaction with this property is called an 'orphan'. Orphans can arise after a failure in a distributed system when a procedure keeps running even though its caller has failed, restarted, and released its locks. The orphan procedure may do things based on the old values of the now unlocked data. `EO ==> OD`.

OC Optimistic Concurrency: uncommitted transactions can see some subset of what has happened. There's no guarantee that any of them can commit; this means that the code must check at commit. Here are two versions; `OC1` is stronger.

- `OC1`: Each sees `AtBegin` + some other stuff + its stuff; this roughly corresponds to having a private workspace for each uncommitted transaction. `OD ==> OC1`.
- `OC2`: Each sees `AtBegin` + some other stuff including its stuff; this roughly corresponds to a shared workspace for uncommitted transactions. `OC1 ==> OC2`

NC No Constraints: uncommitted transactions can see arbitrary values. Again, there's no guarantee that any of them can commit. `OC2 ==> NC`.

Note that each of these implies all the lower ones.

Code

In the remainder of the handout, we discuss various ways to code these specs. These are all ways to code the guards in `Do` and `Commit`, stopping a transaction either from doing an action which will keep it from committing, or from committing if it isn't serializable with other committed transactions.

Two-phase locking

The most common way to code this spec[1] is to ensure that a transaction can always commit (AC) by

> acquiring locks on data in such a way that the outstanding actions of active transactions always commute, and then
>
> doing each action of transaction `t` in a state consisting of the state of all the committed transactions plus the actions of `t`.

This ensures that we can always serialize `t` as the next committed transaction, since we can commute all its actions over those of any other active transaction. We proved a theorem to this effect in handout 17, the "big atomic actions" theorem. With this scheme there is at least one time where a transaction holds all its locks, and any such time can be taken as the time when the transaction executes atomically. If all the locks are held until commit (strict two-phase locking), *the serialization order is the commit order* (more precisely, the commit order is a legal serialization order).

To achieve this we need to associate a set of locks with each action in such a way that any two actions that don't commute have conflicting locks. For example, if the actions are just reads and writes, we can have a read lock and a write lock for each datum, with the rule that read locks don't conflict with each other, but a write lock conflicts with either. This works because two reads commute, while a read and a write do not. Note that the locks are on the actions, not on the updates into which the actions are decomposed to code logging and recovery.

Once acquired, `t`'s locks must be held until `t` commits. Otherwise another transaction could see data modified by `t`; then if `t` aborts rather than committing, the other transaction would also have to abort. Thus we would not be maintaining the invariant that every transaction can always commit, because the premature release of locks means that all the actions of active transactions may not commute. Holding the locks until commit is called *strict two-phase locking*.

A variation is to release locks before commit, but not to acquire any locks after you have released one. This is called *two-phase locking*, because there is a phase of acquiring locks, followed by a phase of releasing locks. Two-phase locking implements the CC spec, in the sense that it guarantees that there's always an order in which all the transactions can commit. Unlike strict two-phase locking, however, simple two-phase locking needs additional machinery, to decide which transactions can commit without waiting—it's the ones that have not touched data written by an uncommitted transaction.

One drawback of locking is that there can be deadlocks, as we saw in handout 14. It's possible to detect deadlocks by looking for cycles in the graph of threads and locks with arcs for the relations "thread `a` waiting for lock `b`" and "lock `c` held by thread `d`". This is usually not done for mutexes, but it often is done by the lock manager of a database or transaction processing system, at least for threads and locks on a single machine. It requires global information about the graph, so it is expensive to code across a distributed system. The alternative is timeout: assume that if a thread waits too long for a lock it is deadlocked. Timeout is the poor man's deadlock detection; most systems use it. A transaction system needs to have an automatic way to handle deadlock

[1] In Jim Gray's words, "People who do it for money use locks." This is not strictly true, but it's close.

because the clients are not supposed to worry about concurrency, and that means they are not supposed to worry about avoiding deadlock.

To get a lot of concurrency, it is necessary to have fine-granularity locks that protect only small amounts of data, say records or tuples. This introduces two problems:

> There might be a great many of these locks.
>
> Usually records are grouped into sets, and an operation like "return all the records with `hairColor = blue`" needs a lock that conflicts with inserting or deleting *any* such record.

Both problems are usually solved by organizing locks into a tree or DAG and enforcing the rule that a lock on a node conflicts with locks on every descendant of that node. When there are too many locks, *escalate* to fewer locks with coarser granularity. This can get complicated; see Gray and Reuter[2] for details.

We now make the locking scheme more precise, omitting the complications of escalation. Each lock needs some sort of name; we use strings, which might have the form `"Read(addr)"`, where `addr` is the name of a variable. Each transaction `t` has a set of locks `locks(t)`, and each action `a` needs a set of locks `protect(a)`. The `conflict` relation says when two locks conflict. It must have the property stated in invariant `I1`, that non-commuting actions have conflicting locks. Note that `conflict` need not be transitive.

Invariant `I2` says that a transaction has to hold a lock that protects each of its actions, and `I3` says that two active transactions don't hold conflicting locks. Putting these together, it's clear that all the committed transactions in commit order, followed by any interleaving of the active transactions, produces the same histories.

```
TYPE Lk          =  String
     Lks         =  SET Lk

CONST
     protect     :  A -> Lks
     conflict    :  (Lk, Lk) -> Bool

% I1: (ALL a1, a2 | a1 * a2 # a2 * a1 ==> conflict(protect(a1), protect(a2)))

VAR  locks       :  T -> Lks

% I2: (ALL t :IN active, e :IN y(t) | protect(e.a) <= locks(t))

% I3: (ALL t1 :IN active, t2 :IN active | t1 # t2 ==>
       (ALL lk1 :IN locks(t1), lk2 :IN locks(t2) | ~ conflict(lk1, lk2)))
```

To maintain `I2` the code needs a partial inverse of the `locks` function that answers the question: does anyone hold a lock that conflicts with `lk`.

Multi-version time stamps

It's possible to give very direct code for the idea that the transactions take place serially, each one at a different instant—we make each one happen at a single instant of logical time. Define a logical time and keep with each datum a history of its value at every instant in time. This can be represented as a sequence of pairs (*time*, *value*), with the meaning that the datum has the given value from the given time until the time of the next pair. Now we can code AC by picking a time

[2] Gray and Reuter, *Transaction Processing: Concepts and Techniques*, Morgan Kaufmann, 1993, pp 406-421.

for each transaction (usually the time of its `Begin`, but any time between `Begin` and `Commit` will satisfy the external consistency requirement), and making every read obtain the value of the datum at the transaction's time and every write set the value at that time.

More precisely, a read at *t* gets the value at the next earlier definition, call it *t'*, and leaves a note that the value of that datum can't change between *t'* and *t* unless the transaction aborts. To maintain AC the read must block if *t'* isn't committed. If the read doesn't block, then the transaction is said to read 'dirty data', and it can't commit unless the one at *t'* does. This version implements CC instead of AC. A write at *t* is impossible if some other transaction has already read a different value at *t*. This is the equivalent of deadlock, because the transaction cannot proceed. Or, in Jim Gray's words, reads are writes (because they add to the history) and waits are aborts (because waiting for a write lock turns into aborting since the value at that time is already fixed).[3] These translations are not improvements, and they explain why multi-version time stamps have not become popular.

A drastically simplified form of multi-version time stamps handles the common case of a very large transaction `t` that reads lots of shared data but only writes private data. This case arises in running a batch transaction that needs a snapshot of an online database. The simplification is to keep just one extra version of each datum; it works because `t` does no writes. You turn on this feature when `t` starts, and the system starts to do copy-on-write for all the data. Once `t` is done (actually, there could be several), the copies can be discarded.

Optimistic concurrency control

> *It's easier to ask forgiveness than to beg permission.*
> Grace Hopper

Sometimes you can get better performance by allowing a transaction to proceed even though it might not be able to commit. The standard version of this *optimistic* strategy allows a transaction to read any data it likes, keeps track of all the data values it has read, and saves all its writes in local variables. When the transaction commits, the system atomically

checks that every datum read still has the value that was read, and

if this check succeeds, installs all the writes.

This obviously serializes the transaction at commit time, since the transaction behaves as if it did all its work at commit time. If any datum that was read has changed, the transaction aborts, and usually retries. This implements OC1 or OC2. The check can be made efficient by keeping a version number for each datum. Grouping the data and keeping a version number for each group is cheaper but may result in more aborts. Computer architects call optimistic concurrency control *speculation* or *speculative execution*.

The disadvantages of optimistic concurrency control are that uncommitted transactions can see inconsistent states, and that livelock is possible because two conflicting transactions can repeatedly restart and abort each other. With locks at least one transaction will always make progress as long as you choose the youngest one to abort when a deadlock occurs.

[3] Gray and Reuter, p 437.

OCC can avoid livelock by keeping a private write buffer for each transaction, so that a transaction only sees the writes of committed transactions plus its own writes. This ensures that at least one uncommitted transaction can commit whenever there's an uncommitted transaction that started after the last committed transaction `t`. A transaction that started before `t` might see both old and new values of variables written by `t`, and therefore be unable to commit. Of course a private write buffer for each transaction is more expensive than a shared write buffer for all of them. This is especially true because the shared buffer can use copy-on-write to capture the old state, so that reads are not slowed down at all.

The Hydra design for a single-chip multi-processor[4] uses an interesting version of OCC to allow speculative parallel execution of a sequential program. The idea is to run several sequential segments of a program in parallel as transactions (usually loop iterations or a procedure call and its continuation). The desired commit order is fixed by the original sequential ordering, and the earliest segment is guaranteed to commit. Each transaction has a private write buffer but can see writes done by earlier transactions; if it sees any values that are later overwritten then it has to abort and retry. Most of this work is done by the hardware of the on-chip caches and write buffers.

Field calls and escrow locks

There is a specialization of optimistic concurrency control called "field calls with escrow locking" that can perform much better under some very special circumstances that occur frequently in practice. Suppose you have an operation that does

```
<< IF pred(v) => v := f(v) [*] RAISE error >>
```

where `f` is total. A typical example is a debit operation, in which `v` is a balance, `pred(v)` is `v > 100`, and `f(v)` is `v - 100`. Then you can attach to `v` a 'pending list' of the `f`'s done by active transactions. To do this update, a transaction must acquire an 'escrow lock' on `v`; this lock conflicts if applying any subset of the `f`'s in the pending list makes the predicate false. In general this would be too complicated to test, but it is not hard if `f`'s are increment and decrement (`v + n` and `v - n`) and `pred`'s are single inequalities: just keep the largest and smallest values that `v` could attain if any subset of the active transactions commits. When a transaction commits, you apply all its pending updates. Since these field call updates don't actually obtain the value of `v`, but only test `pred`, they don't need read locks. An escrow lock conflicts with any ordinary read or write. For more details, see Gray and Reuter, pp 430-435.

This may seem like a lot of trouble, but if `v` is a variable that is touched by lots of transactions (such as a bank branch balance) it can increase concurrency dramatically, since in general none of the escrow locks will conflict.

Full escrow locking is a form of locking, not of optimism. A 'field call' (without escrow locking) is the same except that instead of treating the predicate as a lock, it checks atomically at commit time that all the predicates in the transaction are still true. This *is* optimistic. The original form of optimism is a special case in which every `pred` has the form `v = old value` and every `f(v)` is just `new value`.

[4] Hammond, Nayfeh, and Olukotun, A single-chip multiprocessor, *IEEE Computer*, Sept. 1997. Hammond, Willey, and Olukotun, Data speculation support for a chip multiprocessor, *Proc 8th ACM Conference on Architectural Support for Programming Languages and Operating Systems*, San Jose, California, Oct. 1998. See also http://www-hydra.stanford.edu/publications.shtml.

Nested transactions

It's possible to generalize the results given here to the case of *nested* transactions. The idea is that within a single transaction we can recursively embed the entire transaction machinery. This isn't interesting for handling crashes, since a crash will cause the top-level transaction to abort. It *is* interesting, however, for two things:

1. Making it easy to program with concurrency inside a transaction by relying on the atomicity (that is, serializability) of sub-transactions
2. Making it easy to handle errors by aborting unsuccessful sub-transactions.

With this scheme, each transaction can have sub-transactions within itself. The definition of correctness is that all the sub-transactions satisfy the concurrency control invariant. In particular, all committed sub-transactions are serializable. When sub-transactions have their own nested transactions, we get a tree. When a sub-transaction commits, all its actions are added to the history of its parent.

To code nested transactions using locking we need to know the conflict rules for the entire tree. They are simple: if two different transactions hold locks `lk1` and `lk2` and one is not the ancestor of the other, then `lk1` and `lk2` must not conflict. This ensures that all the actions of all the outstanding transactions commute except for ancestors and descendants. When a sub-transaction commits, its parent inherits all its locks.

Interaction with recovery

We do not discuss in detail how to put this code for concurrency control together with the code for recovery that we studied earlier. The basic idea, however, is simple enough: the two are almost completely orthogonal. All the concurrent transactions contribute their actions to the logs. Committing a transaction removes its undo's from the undo logs, thus ensuring that its actions survive a crash; the single-transaction version of recovery in handout 18 removes everything from the undo logs. Aborting a transaction applies its undo's to the state; the single-transaction version applies all the undo's.

Concurrency control simply stops certain actions (`Do` or `Commit`) from happening, and perhaps aborts some transactions that can't commit. This is clearest in the case of locking, which just prevents any undesired changes to the state. Multi-version time stamps use a more complicated representation of the state; the ordinary state is an abstraction given a particular ordering of the transactions. Optimistic concurrency control aborts some transactions when they try to commit. The trickiest thing to show is that the undo's that recovery does in `Abort` do the right thing.

Performance summary

Each of the coding schemes has some costs when everything is going well, and performs badly for some combinations of active transactions.

When there is no deadlock, locking just pays the costs of acquiring the locks and of checking for deadlock. Deadlocks lead to aborts, which waste the work done in the aborted transactions, although it's possible to choose the aborted transactions so that progress is guaranteed. If the locks

are too coarse either in granularity or in mode, many transactions will be waiting for locks, which increases latency and reduces concurrency.

When there is no conflict, optimistic concurrency control pays the cost of checking for competing changes, whether this is done by version numbers or by saving initial values of variables and checking them at `Commit`. If transactions conflict at `Commit`, they get aborted, which wastes the work they did, and it's possible to have livelock, that is, no progress, in the shared-write-buffer version; it's OK in the private-write-buffer version, since someone has to commit before anyone else can fail to do so.

Multi-version time stamps pay a high price for maintaining the multi-version state in the good case; in general reads as well as writes change it. Transaction conflicts lead to aborts much as in the optimistic scheme. This method is inferior to both of the others in general; it is practical, however, for the special case of copy-on-write snapshots for read-only transactions, especially large ones.

21. Distributed Systems

The rest of the course is about distributed computing systems. In the next four lectures we will characterize distributed systems and study how to specify and code communication among the components of a distributed system. Later lectures consider higher-level system issues: distributed transactions, replication, security, management, and caching.

The lectures on communication are organized bottom-up. Here is the plan:

1. Overview.
2. Links. Broadcast networks.
3. Switching networks.
4. Reliable messages.
5. Remote procedure call and network objects.

Overview

An underlying theme in computer systems as a whole, and especially in distributed systems, is the tradeoff between performance and complexity. Consider the problem of carrying railroad traffic across a mountain range.[1] The minimal system involves a single track through the mountains. This solves the problem, and no smaller system can do so. Furthermore, trains can travel from East to West at the full bandwidth of the track. But there is one major drawback: if it takes 10 hours for a train to traverse the single track, then it takes 10 hours to switch from E-W traffic to W-E traffic, and during this 10 hours the track is idle. The scheme for switching can be quite simple: the last E–W train tells the W-E train that it can go. There is a costly failure mode: the East end forgets that it sent a 'last' E-W train and sends another one; the result is either a collision or a lot of backing up.

The simplest way to solve both problems is to put in a second track. Now traffic can flow at full bandwidth in both directions, and the two-track system is even simpler than the single-track system, since we can dedicate one track to each direction and don't have to keep track of which way traffic is running. However, the second track is quite expensive. If it has to be retrofitted, it may be as expensive as the first one.

A much cheaper solution is to add sidings: short sections of double track, at which trains can pass each other. But now the signaling system must be much more complex to ensure that traffic between sidings flows in only one direction at a time, and that no siding fills up with trains.

What makes a system distributed?

> *One man's constant is another man's variable.*
>
> Alan Perlis

[1] This example is due to Mike Schroeder.

> *A distributed system is a system where I can't get my work done because a computer has failed that I've never even heard of.*
>
> Leslie Lamport

There is no universally accepted definition of a distributed system. It's like pornography: you recognize one when you see it. And like everything in computing, it's in the eye of the beholder. In the current primitive state of the art, Lamport's definition has a lot of truth.

Nonetheless, there are some telltale signs that help us to recognize a distributed system:

It has *concurrency*, usually because there are multiple general-purpose computing elements. Distributed systems are closely related to multiprocessors.

Communication costs are an important part of the total cost of solving a problem on the system, and hence you try to minimize them. This is not the same as saying that the cost of communication is an important part of the system cost. In fact, it is more nearly the opposite: a system in which communication is good enough that the programmer doesn't have to worry about it (perhaps because the system builder spent a lot of money on communication) is less like a distributed system. Distributed systems are closely related to telephone systems; indeed, the telephone system is by far the largest example of a distributed system, though its functionality is much simpler than that of most systems in which computers play a more prominent role.

It *tolerates partial failures*. If some parts break, the rest of the system keeps doing useful work. We usually don't think of a system as distributed if every failure causes the entire system to go down.

It is *scaleable*: you can add more components to increase capacity without making any qualitative changes in the system or its clients.

It is *heterogeneous*. This means that you can add components that implement the system's internal interfaces in different ways: different telephone switches, different computers sending and receiving E-mail, different NFS clients and servers, or whatever. It also means that components may be *autonomous*, that is, owned by different organizations and managed according to different policies. It doesn't mean that you can add arbitrary components with arbitrary interfaces, because then what you have is chaos, not a system. Hence the useful reminder: "There's no such thing as a heterogeneous system."

Layers

> *Any idea in computing is made better by being made recursive.*
>
> Brian Randell

> *There are three rules for writing a novel.*
> *Unfortunately, no one knows what they are.*
>
> Somerset Maugham

You can look at a computer system at many different scales. At each scale you see the same basic components: computing, storage, and communications. The bigger system is made up of smaller ones. Figure 1 illustrates this idea over about 10 orders of magnitude (we have seen it before, in the handout on performance).

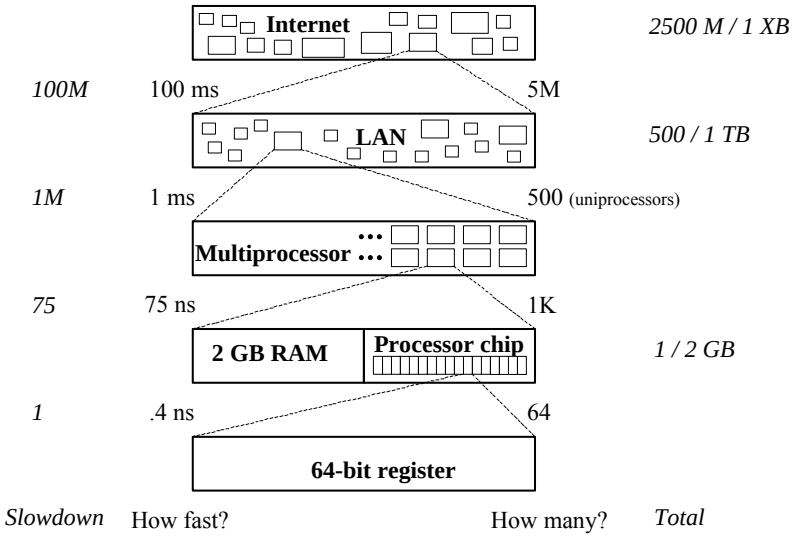


Fig. 1. Scales of interconnection. Relative speed and size are in italics.

But Figure 1 is misleading, because it doesn't suggest that different levels of the system may have quite different interfaces. When this happens, we call the level a *layer*. Here is an example of different interfaces that transport bits or messages from a sender to a receiver. Each layer is motivated by different functionality or performance than the one below it. This stack is ten layers deep. Note that in most cases the motivation for separate layers is either compatibility or the fact that a layer has other clients or other code.

	What	*Why*	
a)	a TCP reliable transport link	function:	reliable stream
b)	on an Internet packet link	function:	routing
c)	on the PPP header compression protocol	performance:	space
d)	on the HDLC data link protocol	function:	packet framing
e)	on a 14.4 Kbit/sec modem line	function:	byte stream
f)	on an analog voice-grade telephone line	function:	3 KHz low-latency signal
g)	on a 64 Kbit/sec digital line multiplexed	function:	bit stream
h)	on a T1 line multiplexed	performance:	aggregation
i)	on a T3 line multiplexed	performance:	aggregation
j)	on an OC-48 fiber.	performance:	aggregation

On top of TCP we can add four more layers, some of which have interfaces that are significantly different from simple transport.

	What	*Why*	

w)	mail folders	function:	organization
x)	on a mail spooler	function:	storage
y)	on SMTP mail transport	function:	routing
z)	on FTP file transport	function:	reliable char arrays

Now we have 14 layers with two kinds of routing, two kinds of reliable transport, three kinds of stream, and three kinds of aggregation. Each serves some purpose that isn't served by other, similar layers. Of course many other structures could underlie the filing of mail messages in folders.

Here is an entirely different example, code for a machine's load instruction:

	What	*Why*	
a)	load from cache	function:	data access
b)	miss to second level cache	performance:	space
c)	miss to RAM	performance:	space
d)	page fault to disk	performance:	space

Layer (d) could be replaced by a page fault to other machines on a LAN that are sharing the memory (function: sharing)[2], or layer (c) by access to a distributed cache over a multiprocessor's network (function: sharing). Layer (b) could be replaced by access to a PCI I/O bus (function: device access), which at layer (c) is bridged to an ISA bus (function: compatibility).

Another simple example is the layering of the various facsimile standards for transmitting images over the standard telephone voice channel and signaling. Recently, the same image encoding, though not of course the same analog encoding of the bits, has been layered on the internet or e-mail transmission protocols.

Addressing

Another way to classify communication systems is in terms of the kind of interface they provide:

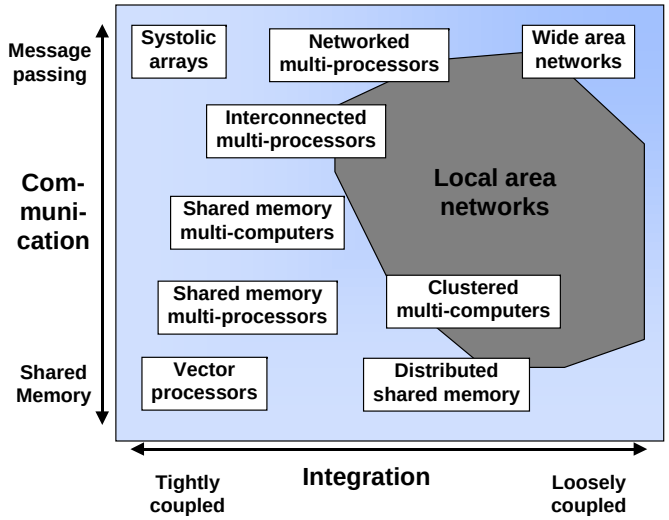
messages or storage,

the form of addresses,

the kind of data transported,

other properties of the transport.

[2] K. Li and P. Hudak: Memory coherence in shared virtual memory systems. *ACM Transactions on Computer Systems* 7, 321-359 (1989)



Here are a number of examples to bear in mind as we study communication. The first table is for messaging, the second for storage.

System	*Address*	*Sample address*	*Data value*	*Delivery*	
				Ordered	*Reliable*
J-machine[3]	source route	4 north, 2 east	32 bytes	yes	yes
IEEE 802 LAN	6 byte flat	`FF F3 6E 23 A1 92`	packet	no	no
IP	4 byte hierarchical	`16.12.3.134`	packet	no	no
TCP	IP + port	`16.12.3.134 / 3451`	byte stream	yes	yes
RPC	TCP + procedure	`16.12.3.134 / 3451 / Open`	arg. record	yes	yes
E-mail	host name + user	`blampson@microsoft.com`	`String`	no	yes

System	*Address*	*Sample address*	*Data value*
Main memory	32-bit flat	`04E72A39`	2^n bytes, $n \leq 4$
File system[4]	path name	`/udir/bwl/Mail/inbox/214`	0-4 Gbytes
World Wide Web	protocol + host name + path name	`http://research.microsoft.com/lampson/default.html`	typed, variable size

Layers in a communication system

The standard picture for a communication system is the OSI reference model, which shows peer-to-peer communication at each of seven layers (given here in the opposite order to the examples above):

[3] W. Dally: A universal parallel computer architecture. *New Generation Computing* **11**(1993), pp 227-249
[4] M. Satyanarayanan: Distributed file systems. In S. Mullender (ed.) *Distributed Systems*, Addison-Wesley, 1993, pp 353-384

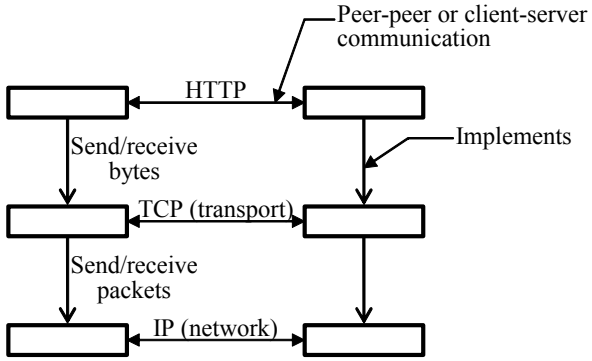


Fig. 2: Protocol stacks for peer-to-peer communication

physical (volts and photons),
data link,
network,
transport,
session,
presentation, and
application.

This model is often, and somewhat pejoratively, called the 'seven-layer cake'. The peer-to-peer aspect of the OSI model is not as useful as you might think, because peer-to-peer communication means that you are writing a concurrent program, something to be avoided if at all possible. At any layer peer-to-peer communication is usually replaced with client-server communication (also known as request-response or remote procedure call) as soon as possible.

The examples we have seen should make it clear that real systems cannot be analyzed so neatly. Still, it is convenient to use the first few layers as tags for important ideas, which we will study in this order:

Data link layer: framing and multiplexing.

Network layer: addressing and routing (or switching) of packets.

Transport layer: reliable messages.

Session layer: naming and encoding of network objects.

We are not concerned with volts and photons, and the presentation and application layers are very poorly defined. Presentation is supposed to deal with how things look on the screen, but it's unclear, for example, which of the following it includes: the X display protocol, the Macintosh PICT format and the PostScript language for representing graphical objects, or the Microsoft RTF format for editable documents. In any event, all of these topics are beyond the scope of this course.

Figure 2 illustrates the structure of communication and code for a fragment of the Internet.

Principles[5]

There are a few important ideas that show up again and again at the different levels of distributed systems: recursion, addresses, end-to-end reliability, broadcast vs. point-to-point, real time, and fault-tolerance.

Recursion

The 14-layer example of coding E-mail gives many examples of encapsulating a message and transmitting it over a lower-level channel. It also shows that it can be reasonable to code a channel using the same kind of channel several levels lower.

Another name for encapsulation is 'multiplexing'.

Addresses

Multi-party communication requires addresses, which can be flat or hierarchical. A flat address has no structure: the only meaningful operation (other than communication) is equality. A hierarchical address, sometimes called a path name, is a sequence of flat addresses or simple names, and if one address is a prefix of another, then in some sense the party with the shorter address contains, or is the parent of, the party with the longer one. Usually there is an operation to enumerate the children of an address. Flat addresses are usually fixed size and hierarchical ones variable, but there are exceptions. An address may be hierarchical in the code but flat at the interface, for instance an Internet address or a URL in the World Wide Web. The examples of addressing that we saw earlier should clarify these points; for more examples see handout 12 on naming.

People often make a distinction between names and addresses. What it usually boils down to is that an address is a name that is interpreted at a lower level of abstraction.

End-to-end reliability

A simple way to obtain reliable communication is to rely on the end points for every aspect of reliability, and to depend on the lower level communication system only to deliver bits with some reasonable probability. The end points check the transmission for correctness, and retry if the check fails.[6]

For example, an end-to-end file transfer system reads the file, sends it, and writes it on the disk in the usual way. Then the sender computes a strong checksum of the file contents and sends that. The receiver reads the file copy from its disk, computes a checksum using the same function, and compares it with the sender's checksum. If they don't agree, the check fails and the transmission must be retried.

In such an end-to-end system, the total cost to send a message is $1 + rp$, where r = cost of retry (if the cost to send a simple message is 1) and p = probability of retry. This is just like fast path (see handout 10 on performance). Note, however, that the retry itself may involve further retries; if $p << 1$ we can ignore this complication. For good performance (near to 1) rp must be small. Since usually $r > 1$, we need a small probability of failure: $p << 1/r < 1$. This means that the link,

[5] My thanks to Alex Shvartsman for some of the figures in this section.
[6] J. Saltzer, D. Reed, and D. Clark: End-to-end arguments in system design. *ACM Transactions on Computer Systems* **2**, 277-288 (1984).

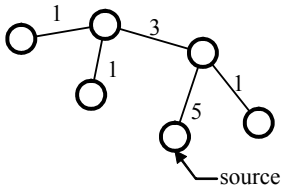


Fig. 3: The cost of doing broadcast with point-to-point communication

though it need not have any *guaranteed* properties, must transmit messages without error most of the time. To get this property, it may be necessary to do forward error correction on the link, or to do retry at a lower level where the cost of retry is less.

Note that p applies to the *entire* transmission that is retried. The TCP protocol, for example, retransmits a whole packet if it doesn't get a positive ack. If the packet travels over an ATM network, it is divided into small 'cells', and ATM may discard individual cells when it is overloaded. If it takes 100 cells to carry a packet, $p_{packet} = 100\ p_{cell}$. This is a big difference.

Of course r can be measured in different ways. Often the work that is done for a retry is about the same as the work that is done just to send, so if we count r as just the work it is about 1. However, the retry is often invoked by a timeout that may be long compared to the time to send. If latency is important, r should measure the time rather than the work done, and may thus be much greater than 1.

Broadcast vs. point-to-point transmission

It's usually much cheaper to broadcast the same information to n places than to send it individually to each of the n places. This is especially true when the physical communication medium is a broadcast medium. An extreme example is direct digital satellite broadcast, which can send a megabyte to everyone in the US for about $.05; compare this with about $.02 to send a megabyte to one place on a local ISDN telephone link. But even when the physical medium is point to point and switches are needed to connect n places, as is the case with telephony or ATM, it's still much cheaper to broadcast because the switches can be configured in a tree rooted at the source of the broadcast and the message needs to traverse each link only once, instead of once for each node that the link separates from the root. Figure 3 shows the number of times a message from the root would traverse each link if it were sent individually to each node; in a broadcast it traverses each link just once.

Historically, most LANs have done broadcast automatically, in the sense that every message reaches every node on the LAN, even if the underlying electrons or photons don't have this property; we will study broadcast networks in more detail later on. Switched LANs are increasingly popular, however, because they can dramatically increase the total bandwidth without changing the bandwidth of a single link, and they *don't* do broadcast automatically. Instead, the switches must organize themselves into a spanning tree that can deliver a message originating anywhere to every node.

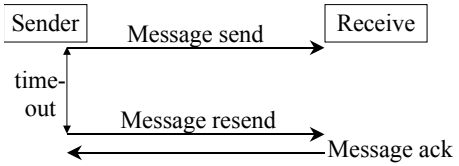
Broadcast is a special case of 'multicast', where messages go to a subset of the nodes. As nodes enter and leave a multicast group, the shape of the tree that spans all the nodes may change. Note that once the tree is constructed, any node can be the root and send to all the others. There are

clever algorithms for constructing and maintaining this tree that are fairly widely implemented in the Internet.[7]

Real time

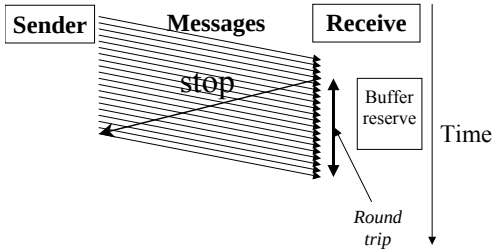
Although often ignored, real time plays an important role in distributed systems. It is used in three ways:

1. To decide when to retry a transmission if there is no response. This often happens when there is some kind of failure, for instance a lost Internet IP packet, as part of an end-to-end protocol. If the retransmission timeout is wrong, performance will suffer but the system will usually still work. When timeouts are used to control congestion, however, making them too short can cause the bandwidth to drop to 0.

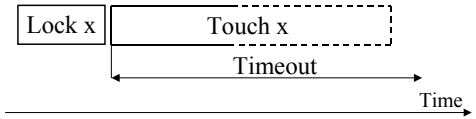


   This generalizes to any kind of fault-tolerance based on replication in time, or retry: the timeout tells you when to retry. More on this under fault-tolerance below.

2. To ensure the stability of a load control system based on feedback. This requires knowing the round trip time for a control signal to propagate. For instance, if a network provides a 'stop' signal when it can't absorb more data, it should have enough buffering to absorb the additional data that may be sent while the 'stop' signal makes its way back to the sender. If the 'stop' comes from the receiver then the receiver should have enough buffering to cover a sender-receiver-sender round trip. If the assumed round-trip time is too short, data will be lost; if it's too long, bandwidth will suffer.



3. To code "bounded waiting" locks, which can be released by another party after a timeout. Such locks are called 'leases'; they work by requiring the holder of the lock to either fail or release it before anyone else times out.[8]. If the lease timeout is too short the system won't work. This means that all the processes must have clocks that run at roughly the same rate. Furthermore, to make use of a lease to protect some operation such as a read or write, a process needs an upper bound on how the operation can last, so that it can check that it will hold the lease until the end of that time. Leases are used in many real systems, for example, to control ownership of a dual-ported disk between two processors, and to provide coherent file caching in distributed file systems. See handout 18 on consensus for more about leases.



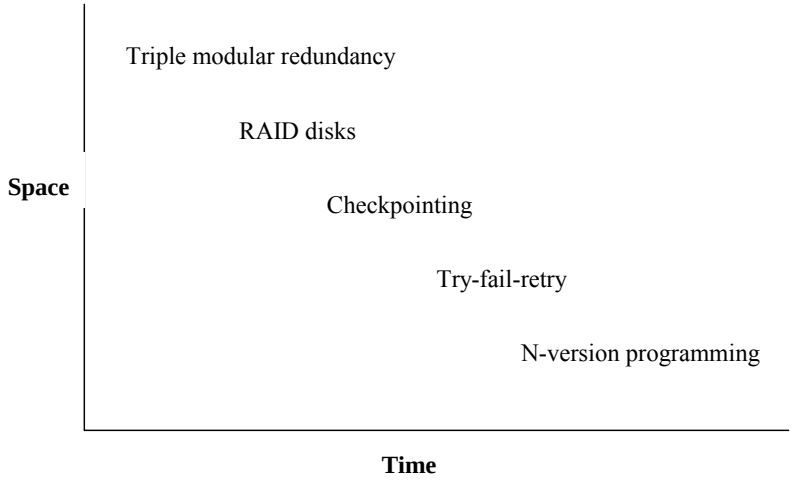
[7] S, Deering et al., An architecture for wide-area multicast routine, *ACM SigComm Computer Communication Review*, **24**, 4 (Oct. 1994), pp 126-135.
[8] C. Gray and D. Cheriton, Leases: An efficient fault-tolerant mechanism for distributed file cache consistency, *Proc. 12th Symposium on Operating Systems Principles*, Dec. 1989, pp 202-210.

Fault tolerance

Fault tolerance is always based on redundancy. The simplest strategy for fault-tolerance is to get the redundancy by replicating fairly large components or actions. Here are three ways to do it:

1. Duplicate components, detect errors, and ignore bad components (replicate in *space*).
2. Detect errors and retry (replicate in *time*, hoping the error is transient).
3. Checkpoint, detect errors, crash, reconfigure without the bad components, and restart from the checkpoint (a more general way to replicate in time)

There is a space-time tradeoff illustrated in the following picture.



Highly available systems use the first strategy. Others use the second and third, which are cheaper as long as errors are not too frequent, since they substitute duplication in time for duplication in space (or equipment). The second strategy works very well for communications, since there is no permanent state to restore, retry is just resend, and many errors are transient. The third strategy is difficult to program correctly without transactions, which are therefore an essential ingredient for complex fault tolerant systems.

Another way to look at the third approach is as *failover* to an alternate component and retry; this requires a failover mechanism, which for communications takes the simple form of changes in the routing database. An often-overlooked point is that unless the alternate component is only

used as a spare, it carries more load after the failure than it did before, and hence the performance of the system will decrease.

In general, fault tolerance requires timeouts, since otherwise you wait indefinitely for a response from a faulty component. Timeouts in turn require knowledge of how long things should take, as we saw in the previous discussion of real time. When this knowledge is precise, we call the system 'synchronous'; timeouts can be short and failure detection rapid, conditions that are usually met at low levels in a system. It's common to design a snoopy cache, for instance, on the assumption that every processor will respond in the same cycle so that the responses can be combined with an 'or' gate.[9] Higher up there is a need for compatibility with several implementations, and each lower level with caching adds uncertainty to the timing. It becomes more difficult to set timeouts appropriately; often this is the biggest problem in building a fault-tolerant system. Perhaps we should specify the real-time performance of systems more carefully, and give up the use of caches such as virtual memory that can cause large variations in response time.

All these methods have been used at every level from processor chips to distributed systems. In general, however, below the level of the LAN most systems are synchronous and not very fault-tolerant: any permanent failure causes a crash and restart. Above that level most systems make few assumptions about timing and are designed to keep working in spite of several failures. From this difference in requirements follow many differences in design.

In a system that cannot be completely reset, it is important to have *self-stabilization*: the system can get from an arbitrary state (which it might land in because of a failure) to a good state.[10]

In any fault-tolerant system the algorithms must be 'wait-free' or 'non-blocking', which means that the failure of one process (or of certain sets of processes, if the system is supposed to tolerate multiple failures) cannot keep the system from making progress.[11] Unfortunately, simple locking is not wait-free. Locking with leases is wait-free, however. We will study some other wait-free algorithms that don't depend on real time. We said a little about this subject in handout 14 on practical concurrency.[12] Note that the Paxos algorithm is wait-free; see handout 18 on consensus.

Performance of communication

Communication has the same basic performance measures as anything else: latency and bandwidth.

- *Latency*: how long a minimum communication takes. We can measure the latency in bytes by multiplying the latency time by the bandwidth; this gives the capacity penalty for each separate operation. There are standard methods for minimizing the effects of latency:

  Caching *reduces* latency when the cache hits.

  Prefetching *hides* latency by the distance between the prefetch and the use.

  Concurrency *tolerates* latency by giving something else to do while waiting.

[9] Hennessey and Patterson, section 8.3, pp 654-676.
[10] G. Varghese and M. Jayaram, The fault span of crash failures, *JACM*, to appear. Available here.
[11] These terms are not actually synonyms. In a wait-free system every process makes progress. In a non-blocking system some process is always making progress, but it's possible for a process to be starved indefinitely.
[12] M. Herlihy. Wait-free synchronization. *ACM Transactions on Programming Languages and Systems* **13**, 1 (Jan. 1991), pp 124-149.

- *Bandwidth*: how communication time grows with data size. Usually this is quoted for a two-party link. The "bisection bandwidth" is the minimum bandwidth across a set of links that partition the system into roughly equal-size parts if they are removed; it is a lower bound on the possible total rate of uniform communication. There are standard methods for minimizing the cost of bandwidth:

  Caching saves bandwidth when the cache hits.

  More generally, locality saves bandwidth when cost increases with distance.

  'Combining networks' save bandwidth to a hot spot by combining several operations into one, several loads or increments for example.

  Code shipping saves bandwidth by sending the code to the data.[13]

In addition, there are some other issues that are especially important for communication:

- *Connectivity*: how many parties you can talk to. Sometimes this is a function of latency, as in the telephone system, which allows you to talk to millions of parties but only one at a time.
- *Predictability*: how much latency and bandwidth vary with time. Variation in latency is called 'jitter'; variation in bandwidth is called 'burstiness'. The biggest difference between the computing and telecommunications cultures is that computer communication is basically unpredictable, while telecommunications service is traditionally highly predictable.
- *Availability*: the probability that an attempt to communicate will succeed.

Uniformity of performance at an interface is often as important as absolute performance, because dealing with non-uniformity complicates programming. Thus performance that depends on locality is troublesome, though often rewarding. Performance that depends on congestion is even worse, since congestion is usually much more difficult to predict than locality. By contrast, the Monarch multiprocessor[14] provides uniform, albeit slow, access to a shared memory from 64K processors, with a total bandwidth of 256 Gbytes/sec and a very simple programming model. Since all the processors make memory references synchronously, it can use a combining network to eliminate many hot spots.

Specs for communication

Regardless of the type of message being transported, all the communication systems we will study implement one of a few specs. All of them are based on the idea of sending and receiving messages through a channel. The channel has state that is derived from the messages that have been sent. Ideally the state is the sequence of messages that have been sent and not yet delivered, but for weaker specs the state is different. In addition, a message may be acknowledged. This is interesting if the spec allows messages to be lost, because the sender needs to know whether to retransmit. It may also be interesting if the spec does not guarantee prompt delivery and the sender needs to know that the message has been delivered.

None of the specs allows for messages to be corrupted in transit. This is because it's easy to convert a corrupted message into a lost message, by attaching a sufficiently good checksum to each

[13] Thanks to Dawson Engler for this observation.
[14] R. Rettberg et al.: The Monarch parallel processor hardware design. *IEEE Computer* 23, 18-30 (1990)

message and discarding any message with an incorrect checksum. It's important to realize that the definition of a 'sufficiently good' checksum depends on a model of what kind of errors can occur. To take an extreme example, if the errors are caused by a malicious adversary, then the checksum must involve some kind of secret, called a 'key'; such a checksum is called a 'message authentication code'. At the opposite extreme, if only single-bit errors are expected, (which is likely to be the case on a fiber optic link where the errors are caused by thermal noise) then a 32-bit CRC may be good; it is cheap to compute and it can detect three or fewer single-bit errors in a message of less than about 10 KB. In the middle is an unkeyed one-way function like MD5.[15]

These specs are for messages between a single sender and a single receiver. We allow for lots of sender-receiver pairs initially, and then suppress this detail in the interests of simplicity.

```
MODULE Channel[
    M,                                          % Message
    A ] =                                       % Address

TYPE Q          =  SEQ M                        % Queue: channel state
    SR          =  [s: A, r: A]                 % Sender - Receiver
    K           =  ENUM[ok, lost]               % acK

...

END Channel
```

Perfect channels

A perfect channel is just a FIFO queue. This one is unbounded. Note that `Get` blocks if the queue is empty.

```
VAR q           := (SR -> Q){* -> {}}           % all initially empty

APROC Put(sr, m)    = << q(sr) := q(sr) + {m} >>
APROC Get(sr) -> M = << VAR m | m = q(sr).head => q(sr) := q(sr).tail; RET m >>
```

Henceforth we suppress the `sr` argument and deal with only one channel, to reduce clutter in the specs.

Reliable channels

A reliable channel is like a perfect channel, but it can be down, in which case the channel is allowed to lose messages. Now it's interesting to have an acknowledgment. This spec gives the simplest kind of acknowledgment, for the last message transmitted. Note that `GetAck` blocks if `status` is `nil`; normally this is true iff `q` is non-empty. Also note that if the channel is down, `status` can become `lost` even when no message is lost.

```
VAR q           := {}
    status      :  (K + Null) := ok
    down        := false

APROC Put(m)        = << q := q + {m}, status := nil >>

APROC Get() -> M    = << VAR m | m = q.head =>
    q := q.tail; IF q = {} => status := ok [*] SKIP FI; RET m >>
```

[15] B. Schneier, *Applied Cryptography*, Wiley, 1994, p 329.


```
APROC GetAck() -> K = << VAR k | k = status => status := ok; RET k >>

APROC Crash()       = down := true
APROC Recover()     = down := false

THREAD Lose()       = DO                                    % internal action
    << down =>
        IF  VAR q1, q2, m | q = q1 + {m} + q2 =>
            q := q1 + q2; IF q2 = {} => status := lost [*] SKIP FI
        [*] status := lost
        FI >>
    [*] SKIP OD
```

Unreliable channels

An unreliable channel is allowed to lose, duplicate, or reorder messages at any time. This is an interesting spec because it makes the minimum assumptions about the channel. Hence anything built on this spec can work on the widest variety of channels. The reason that duplication is important is that the way to recover from lost packets is to retransmit them, and this can lead to duplication unless a lot of care is taken, as we shall see in handout 26 on reliable messages. A variation (not given here) bounds the number of times a message can be duplicated.

```
VAR q           := Q{}                              % as a multiset!

APROC  Put(m)     = << q := q \/ {m} >>
APROC  Get() -> M = << VAR m | m IN q => q := q - {m}; RET m >>

THREAD Lose()     = DO VAR m | << m IN q => q := q -  {m} >> [*] SKIP OD
THREAD Dup()      = DO VAR m | << m IN q => q := q \/ {m} >> [*] SKIP OD
```

An unreliable FIFO channel is a model of a point-to-point wire or of a broadcast LAN without bridging or switching. It preserves order and does not duplicate, but can lose messages at any time. This channel has `Put` and `Get` exactly like the ones from a perfect channel, and a `Lose` much like the unreliable channel's `Lose`.

```
VAR q           := Q{}                              % all initially empty

APROC  Put(m)     = << q := q + {m} >>
APROC  Get() -> M = << VAR m | m = q.head => q := q.tail; RET m >>

THREAD Lose()     =
    DO << VAR q1, q2, m | q = q1 + {m} + q2 => q := q1 + q2 >> [*] SKIP OD
```

These specs can also be written in an 'early-decision' style that decides everything about duplication and loss in the `Put`. As usual, the early decision spec is shorter. It takes a prophecy variable (handout 8) to show that the code with `Lose` and `Dup` implements the early decision spec for the unreliable FIFO channel, and for the unordered channel it isn't true, because the early decision spec cannot deliver an unbounded number of copies of `m`. Prophecy variables can work for infinite traces, but there are complicated technical details that are beyond the scope of this course.

Here is the early decision spec for the unreliable channel:

```
VAR  q            := Q{}                                        % as a multiset!

APROC  Put(m)      = << VAR i: Nat => q := q \/ {j :IN i.seq || m} >>
APROC  Get() -> M = << VAR m | m IN q => q := q - {m}; RET m >>
```

and here is the one for the unreliable FIFO channel

```
VAR  q            := Q{}                                        % all initially empty

APROC  Put(m)      = << q := q + {m} [] SKIP >>
APROC  Get() -> M = << VAR m | m = q.head => q := q.tail; RET m >>
```


22. Paper: Autonet: A High-Speed, Self-Configuring Local Area Network Using Point-to-Point Links

The attached paper on Autonet by Michael Schroeder et al. appeared as report 59, Systems Research Center, Digital Equipment Corp., April 1990. Essentially the same version was published in *IEEE Journal on Selected Areas in Communications* **9**, 8, (October 1991), pp1318-1335.

Read it as an adjunct to the lectures on distributed systems, links, and switching. It gives a fairly complete description of a working highly-available switched network providing daily service to about 100 hosts. The techniques used to obtain high reliability and fault-tolerance are characteristic of many distributed systems, not just of networks. The paper also makes clear the essential role of software in modern networks.

A second paper on Autonet describes the reconfiguration scheme in detail: Thomas Rodeheffer and Michael D. Schroeder. Automatic reconfiguration in Autonet. In *Proceedings of the 13th ACM Symposium on Operating System Principles*, pages 183-187, 1991.

.

23. Networks — Links and Switches[1]

This handout presents the basic ideas for transmitting digital data over links, and for connecting links with switches so that data can pass from lots of sources to lots of destinations. You may wish to read chapter 7 of Hennessy and Patterson for a somewhat different treatment, more focused on interconnecting the components of a multiprocessor computer.

Links

A link is an unreliable FIFO channel. As we mentioned in handout 21, it is an abstraction of a point-to-point wire or of a simple broadcast LAN. It is unreliable because noise or other physical problems can corrupt messages.

There are many kinds of physical links, with cost and performance that vary based on length, number of drops, and bandwidth. Here are some current examples. Bandwidth is in bytes/second[2], and the "+" signs mean that software latency must be added. The nature of the messages reflects the origins of the link. Computer people prefer variable-size packets, which are good for bursty traffic. Communications people have historically preferred bits or bytes, which are good for fixed-bandwidth voice traffic and minimize the latency and buffering added by collecting voice samples into a message.

A physical link can be unidirectional ('simplex') or bidirectional ('duplex'). A duplex link may operate in both directions at once ('full-duplex'), or in one direction at a time ('half-duplex'). A pair of simplex links in opposite directions forms a full-duplex link. So does a half-duplex link in which the time to reverse direction is negligible, but in this case the peak full-duplex bandwidth is half the half-duplex bandwidth. If most of the traffic is in one direction, however, the usable bandwidth of a half-duplex link may be nearly the same as that of a full-duplex link.

To increase the bandwidth of a link, run several copies of it in parallel. This goes by different names; 'space division multiplexing' and 'striping' are two of them. Common examples are:

Parallel busses, as in the first four lines of the table.

Switched networks: the telephone system and switched LANs.

Multiple disks, each holding part of a data block, that can transfer in parallel.

Cellular telephony, using spatial separation to reuse the same frequencies.

In the latter two cases the parallelism is being added to links that were originally designed to operate alone, so there must be physical switches to connect the parallel links.

Another use for multiple links is fault tolerance, discussed earlier.

[1] My thanks to Alex Shvartsman for some of the figures in this handout.
[2] Beware: communications people usually quote bits/sec, so network bandwidth tends to be quoted this way. All the numbers in the table are in bytes, however, except for the bus width in bits.

Medium	*Link*	*Bandwidth*		*Latency*		*Width*	*Message*
Pentium 4 chip	on-chip bus	1030	GB/s	.4	ns	64	word
PC board	Rambus bus	1.6	GB/s	75	ns	16	memory packet
	PCI I/O bus	533	MB/s	200	ns	32/64	DMA block
Wires	Fibre channel[3]	125	MB/s	200	ns	1	packet
	IEEE 1394[4]	50	MB/s	1	µs	1	packet
	USB 2	50	MB/s	1	µs	1	?
	SCSI	40	MB/s	500	ns	16	32
	Serial ATA	300	MB/s	200	ns	1	?
LAN	Gigabit Ethernet	125	MB/s	1 +	µs	1	packet, 64-1500 B
	Fast Ethernet[5]	12.5	MB/s	10 +	µs	1	packet, 64-1500 B
	Ethernet	1.25	MB/s	100 +	µs	1	packet, 64-1500 B
Wireless	802.11a	6	MB/s	100 +	µs	1	packet, < 1500 B
Fiber (Sonet)	OC-48	300	MB/s	5	µs/km	1	byte or 48 B cell
Coax cable	T3	6	MB/s	5	µs/km	1	byte
Copper pair	T1	0.2	MB/s	5	µs/km	1	byte
Copper pair	ISDN	16	KB/s	5	µs/km	1	byte
Broadcast	CAP 16	3	MB/s	3	µs/km	6 MHz	byte or cell

Flow control

Many links do not have a fixed bandwidth that is known to the sender, because the link is being shared (that is, there is multiplexing inside the link) or because the receiver can't always accept data. In particular, fixed bandwidth is bad when traffic is bursty, because it will be either too small or too large. If the sender doesn't know the link bandwidth or can't be trusted to stay below it, some kind of *flow control* is necessary to match the flow of traffic to the link's or the receiver's capacity. A link can provide this in two ways, by contention or by scheduling. In this case these general strategies take the form of *backoff* or *backpressure*.

Backoff

In backoff the link drops excess traffic and signals 'trouble' to the sender, either explicitly or by failing to return an acknowledgment. The sender responds by waiting for a while and then retransmitting. The sender increases the wait by some factor (say 2) after every trouble signal and decreases it with each trouble-free send. This is called 'exponential backoff'; when the increasing factor is 2, it is 'binary exponential backoff'. It is used in the Ethernet[6] and in TCP[7], and is analyzed in some detail in a later section.

[3] M. Sachs and A. Varman, Fibre channel and related standards. *IEEE Communications* **34**, 8 (Aug. 1996), pp 40-49.
[4] G. Hoffman and D. Moore, IEEE 1394: A ubiquitous bus. *Digest of Papers, IEEE COMPCON '95*, 1995, pp 334-338.
[5] M. Molle and G. Watson, 100Base-T/IEEE 802.12/Packet switching. *IEEE Communications* **34**, 8 (Aug. 1996), pp 63-73.
[6] R. Metcalfe and D. Boggs: Ethernet: Distributed packet switching for local computer networks. *Communications of the ACM* **19**, 395-404 (1976)
[7] V. Jacobsen: Congestion avoidance and control. *ACM SigComm Conference*, 1988, pp 314-329. C. Lefelhocg et al., Congestion control for best-effort service. *IEEE Network* **10**, 1 (Jan 1996), pp 10-19.

Exponential backoff works because it adjusts the rate of sending so that most packets get through. If every sender does this, then every sender's delay will jiggle around the value at which the network is just managing to carry all the traffic. This is because a wait that is too short will overload the network, some packets will be lost, and the sender will increase the wait. On the other hand, a wait that is too long will always succeed, and the sender will decrease it. Of course these statements are probabilistic: sometimes a conservative sender will lose a packet because someone else overloaded the network.

The precise details of how the wait should be lengthened (backed off) and shortened depend on the properties of the channel. If the 'trouble' signal comes back very quickly and the cost of trouble is small, senders can shorten their waits aggressively; this happens in the Ethernet, where collisions are detected in at most 64 byte times and abort the transmission immediately, so that senders can start with 0 wait for each new message. Under the opposite conditions, senders must shorten their waits cautiously; this happens in TCP, where the 'trouble' signal is only the lack of an acknowledgment, which can only be detected by timeout and which cannot abort the transmission immediately. The timeout should be roughly one round-trip time; the fact that in TCP it's often impossible to get a good estimate of the round-trip time is a serious complication.

An obvious problem with backoff is that it requires all the senders to cooperate. A sender who doesn't play by the rules can get an unfair share of the link resource, and in many cases two such senders can cause the total throughput of the entire link to become very small.

Backpressure

In backpressure the link tells the sender explicitly how much it can send without suffering losses. This can take the form of start and stop signals, or of 'credits' that allow a certain amount of additional traffic to be sent. The number of unused credits the sender has is called its 'window'. Let b be the bandwidth at which the sender can send when it has permission and r be the time for the link to respond to new traffic from the sender. A start–stop scheme can allow rb units of traffic between a start and a stop; a link that has to buffer this traffic will overrun and lose traffic if r is too large. A credit scheme needs rb credits when the link is idle to keep running at full bandwidth; a link will underrun and waste bandwidth if r is too large.[8]

Start–stop is used in the Autonet[9] (handout 22), and on RS-232 serial lines under the name XON-XOFF. The Ethernet, although it uses backoff to control acquiring the channel, also uses backpressure, in the form of carrier sense, to keep a sender from interrupting another sender that has already acquired the channel; this is called 'deference'. TCP uses credits to allow the receiver to control the flow. It also uses backoff to deal with congestion within the link itself (that is, in the underlying packet network). Having both mechanisms is confusing, and it's even more confusing (though clever) that the waits required by backoff are coded by fiddling with credits.

The failure modes of the two backpressure schemes are different. A lost 'stop' may cause lost data. A lost credit may reduce the bandwidth but doesn't cause data to be lost. On the other hand, 'start' and 'stop' are idempotent, so that a good state is restored just be repeating them. This is not true for credits of the form "send n more messages". There are several ways to get around this problem with credits:

[8] H. Kung and R. Morris, Credit-based flow control for ATM networks. *IEEE Network* **9**, 2 (Mar. 1995), pp 40-48.
[9] M. Schroeder et al., Autonet: A high-speed self-configuring local area network using point-to-point links. *IEEE Journal on Selected Areas in Communication* **9**, 8 (Oct. 1991), pp 1318-1335.

Number the messages, and send credits in the form "*n* messages after message *k*". Such a credit resets the sender's window completely. TCP uses this scheme, counting bytes rather than messages. On an unreliable channel, however, it only works if each message carries its own number, and this is extra overhead that is serious if the messages are small (for instance, ATM cells are only 53 bytes, and only 48 bytes of this are payload).

Stop sending messages and send a 'resync' request. When the receiver gets this it returns an absolute rather than an incremental credit. Once the sender gets this it resets its window and starts sending again. There are various schemes for avoiding a hiccup during the resync.

Know the round-trip time between sender and receiver, and keep track of *m*, the number of messages sent during the last round-trip time. The receiver sends an absolute credit *n*, and the sender sets its window to $n - m$, since there are *m* messages outstanding that the receiver didn't know about when it issued *n* credits. This works well for links with no buffering (for example, simple wires), because the round-trip time is constant. It works poorly if the link has internal buffering, because the round-trip time varies.

Another form of flow control that is similar to backpressure is called 'rate-based'. It assigns a maximum transmission bandwidth or 'rate' to each sender, undertakes to deliver traffic up to that bandwidth with high probability, and is free to discard excess traffic. The rate is measured by taking a moving average across some time window.[10]

Framing

The idea of framing (sometimes called 'acquiring sync') is to take a stream of X's and turn it into a stream of Y's. An X might be a bit and a Y a byte, or an X might be a byte and a Y a packet. This is a parsing problem. It occurs repeatedly in communications, at every level from analog signals through bit streams, byte streams, and streams of cells up to encoded procedure calls. We looked at this problem abstractly and in the absence of errors when we studied encoding and decoding in handout 7. For communication the parsing has to work even though physical problems such as noise can generate an arbitrary prefix of X's before a sequence of X's that correctly encodes some Y's.

If an X is big enough to hold a label, framing is easy: You just label each X with the Y it is part of, and the position it occupies in that Y. For example, to frame (or encapsulate) an IP packet on the Ethernet, just make the 'protocol type' field of the packet be 'IP', and if the packet is too big to fit in an Ethernet packet, break it up into 'fragments' and add a part number to each part. The receiver collects all the parts and puts them back together.[11] The jargon for the entire process is 'fragmentation/re-assembly'.

If X is small, say a bit or a byte, or even the measurement of a signal's voltage level, more cleverness is needed. There are many possibilities, all based on the idea of a 'sync' pattern that allows the receiver to recognize the start of a Y no matter what the previous sequence of X's has been.

[10] F. Bonomi and K. Fendick, The rate-based flow control framework for the available bit rate ATM service. *IEEE Network* **9**, 2 (Mar. 1995), pp 25-39.
[11] Actually fragmentation is usually done at the IP level itself, but the idea is the same.

Certain *values* of X can be reserved to mark the beginning or the end of a Y. In FDDI[12], for example, 4 bits of data are coded in 5 bits on the wire (this is called a 4/5 code). This is done because the wire doesn't work if there are too many 0's or too many 1's in a row, so it's not possible to simply send the data bytes. However, the wire's demands are weak enough that there are more than 16 allowable 5-bit combinations, and one of these is used as the sync mark for the start of a packet.[13] If a 'sync' appears in the middle of a packet, that is taken as an error, and the next legal symbol is the start of a new packet. A simpler version of this idea requires at least one transition on every bit (in 10 Mb Ethernet) or byte (in RS-232); the absence of a transition for a bit or byte time is a sync.

Certain *sequences* of X can be reserved to mark the beginning of a Y. If these sequences occur in the data, they must be 'escaped' or coded in some other way. A familiar example is C's literal strings, in which `'\'` is used as an escape, and to represent a `'\'` you must write `'\\'`. In HDLC an X is a bit, the rule is that more than *n* 0 bits is a sync for some small value of *n*, and the escape mechanism, called 'bit-stuffing', adds a 1 after each sequence of *n* data zeros when sending and removes it when receiving. In RS-232 an X is a high or low voltage level, sampled at say 10 times the bit rate, a Y is (usually) 8 data bits plus a 'start bit' which must be high and a 'stop bit' which must be low. Thus every Y begins with a low-high transition which determines the phase for the rest of the Y (this is called 'clock recovery'), and a sequence of 9 or more bit-times worth of low is a sync.

The sequences used for sync can be detected *probabilistically*. In telephony T-1 signaling there is a 'frame' of 193 bits, one sync bit and 192 data bits. The data bits can be arbitrary, but they are xored with a 'scrambling' sequence to make them pseudo-random. The encoding specifies a definite pattern (say "010101") for the sync bits of successive frames (which are not scrambled). The receiver decodes by guessing the start of a frame and checking a number of frames for the sync pattern. If it's not there, the receiver makes a different guess. After at most 193 tries it will have guessed right. This takes a lot longer than the previous schemes to acquire sync, but it uses a constant amount of extra bandwidth (unlike escape schemes), and much less than fixed sync schemes: 1/193 for T-1 instead of 1/5 for FDDI, 1/2 for Ethernet, or 1/10 for RS-232.

Multiplexing

Multiplexing is a way to share a link among multiple senders and receivers. It raises two issues:

Arbitration (for the sender)—when to send.

Addressing (for the receiver)—when to receive.

A 'multiplexer' implements arbitration; it combines traffic from several input links onto one output link. A 'demultiplexer' implements addressing; it separates traffic from one input link onto several output links. The multiplexed links are called 'sub-channels' of the one link, and each one has an address. Figure 1 shows various examples; the ovals are buffers.

[12] F. Ross: An overview of FDDI: The fiber distributed data interface. *IEEE Journal on Selected Areas in Communication* 7 (1989)
[13] Another symbol is used to encode a token, and several others are used for somewhat frivolous purposes.

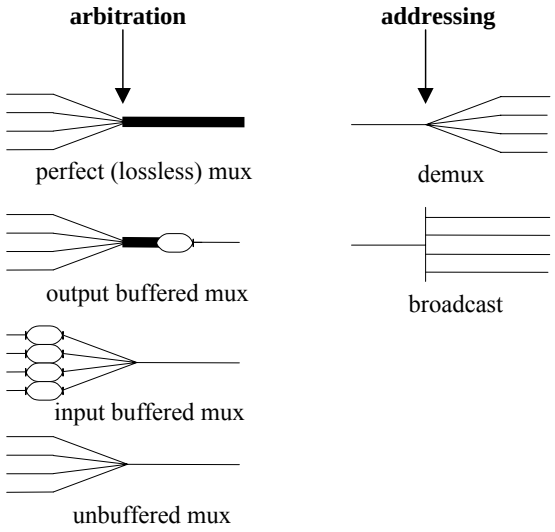


Fig. 1. Multiplexers and demultiplexers. Traffic flows from left to right. Fatter lines are faster channels.

There are three main reasons for multiplexers:

- Traffic may flow between one node and many on a single wire, for example when the one node is a busy server or the head end of a cable TV system.
- One wide wire may be cheaper than many narrow ones, because there is only one thing to install and maintain, or because there is only one connection at the other end. Of course the wide wire is more expensive than a single narrow one, and the multiplexers must also be paid for.
- Traffic aggregated from several links may be more predictable than traffic from a single one. This happens when traffic is bursty (varies in bandwidth) but uncorrelated on the input links. An extreme form of bursty traffic is either absent or present at full bandwidth. This is standard in telephony, where extensive measurements of line utilization have shown that it's very unlikely for more than 10% of the lines to be active at one time, at least for voice calls.

There are many techniques for multiplexing. In the analog domain:

- *Frequency division* (FDM) uses a separate frequency band for each sub-channel, taking advantage of the fact that e^{int} is a convenient basis set of orthogonal functions. The address is the frequency band of the sub-channel. FDM is used to subdivide the electromagnetic spectrum in free space, on cables, and on optical fibers. On fibers it's usually called 'wave division multiplexing', and they talk about wavelength rather than frequency, but of course it's the same thing.
- *Code division* (CDM, usually called CDMA for 'code division multiple access') uses a different coordinate system in which a basis vector is a time-dependent sequence of frequencies. This smears out the cross-talk between different sub-channels. The address is the 'code', the sequence of frequencies. CDM is used for military communications and in newer varieties of

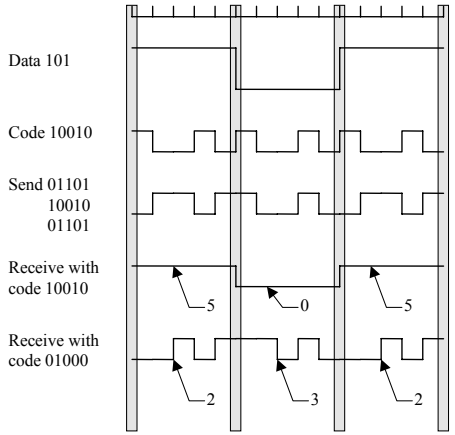


Fig 2: Simple code division multiplexing

cellular telephony. Figure 2 illustrates the simplest form of CDM, in which *n* senders share a digital channel. Bits on the channel have length 1, each sender's bits have length *n* (5 in the figure), and a sender has an *n*-bit 'code' (10010 in the figure) which it xor's with its current data bit. The receiver xor's the code in again and looks for either all zeros or all ones. If it sees something intermediate, that is interference from a sender with a different code. If the codes are sufficiently orthogonal (agree in few enough bits), the contributions of other senders will cancel out. Clearly longer code words work better.

In the digital domain time-division multiplexing (TDM) is the standard method. It comes in two flavors:

—*Fixed* TDM, in which *n* sub-channels are multiplexed by dividing the data sequence on the main channel into fixed-size slots (single bits, bytes, or whatever) and assigning every *n*th slot to the same sub-channel. Usually all the slots are the same size, but it's sufficient for the sequence of slot sizes to be fixed. The 1.5 Mbit/sec T1 line that we discussed earlier, for example, has 24 sub-channels and 'frames' of 193 bits. One bit marks the start of the frame, after which the first byte belongs to sub-channel 1, the second to sub-channel 2, and so forth. Slots are numbered from the start of the frame, and a sub-channel's slot number is its address. Note that this scheme requires framing to find the start of the frame (hence the name). But the addressing has no direct code (there is an "internal fragmentation" cost if the fixed channels are not fully utilized).

—*Variable* TDM, in which the data sequence on the main channel is divided into 'packets'. One packet carries data for one sub-channel, and the address of the sub-channel appears explicitly in the packet. If the packets are fixed size, they are often called 'cells', as in the Asynchronous Transfer Mode (ATM) networking standard. Fixed-size packets are used in other contexts, however, for instance to carry load and store messages on a programmed I/O bus. Variable sized packets (up to some maximum that either is fixed or depends on the link) are usual in computer networking, for example on the Ethernet, token ring, FDDI, or Internet, as well as for DMA bursts on I/O busses.

All these methods fix the division of bandwidth among sub-channels except for variable TDM, which is thus better suited to handle the burstiness of computer traffic. This is the only architectural difference among them. But there are other architectural differences among multiplexers,

resulting from the different ways of coding the basic function of *arbitrating* among the input channels. The fixed schemes do this in a fixed way that is determined which the sub-channels are assigned. This is illustrated at the top of figure 1, where the wide main channel has enough bandwidth to carry all the traffic the input channels can offer. Arbitration is still necessary when a sub-channel is assigned to an input channel; this operation is usually called 'circuit setup'.

With variable TDM there are many ways to arbitrate, but they fall into two main classes, which parallel the two methods of flow control described in the section on links above:

- *Collision* (parallel to backoff): an input channel simply sends its traffic, but has some way to tell whether the traffic was accepted. If not, it 'backs off' by waiting for a while, and then retries. The input channel can get an explicit and immediate collision signal, as on the Ethernet, it can get a delayed collision signal in the form of a 'negative acknowledgment', or it can infer a collision from the lack of an acknowledgment, as in TCP.
- *Scheduling* (parallel to backpressure): an input channel requests service and the multiplexer eventually grants it; I/O busses and token rings work this way. Granting can be centralized, as in many I/O busses, or distributed, as in a daisy-chained bus or a token ring like FDDI.

Flow control means buffering, as we saw earlier, and there are several ways to arrange buffering around a multiplexer, shown on the left side of figure 1. Having the buffers near the arbitration point is good because it reduces the round-trip time *r* and hence the size of the buffers. Output buffering is good because it allows arbitration to ignore contention for the output until the buffer fills up, but the buffer may cost more because it has to accept traffic at the total bandwidth of all the inputs. A switch implemented by a shared memory pays this cost automatically, and the shared memory acts as a shared buffer for all the outputs.

A multiplexer can be centralized, like a T1 multiplexer or a crosspoint in a crossbar switch, or it can be distributed along a bus. It seems natural to use scheduling with a centralized multiplexer and collision with a distributed one, but the examples of the Monarch memory switch[14] and the token ring described below show that the other combinations are also possible.

Multiplexers can be cascaded to increase the fan-in. This structure is usually combined with a converter. For example, 24 voice lines, each with a bandwidth of 64 Kb/s, are multiplexed to one 1.5 Mb/s T1 line, 30 of these are multiplexed to one 45 Mb/s T3 line, and 50 of these are multiplexed to one 2.4 Gb/s OC-48 fiber which carries 40,000 voice sub-channels. In the Vax 8800, 16 Unibuses are multiplexed to one BI bus, and 4 of these are multiplexed to one internal processor-memory bus.

Demultiplexing uses the same physical mechanisms as multiplexing, since one is not much use without the other. There is no arbitration, however; instead, there is *addressing*, since the input channel must select the proper output channel to receive each sub-channel. Again both centralized and distributed versions are possible, as the right side of figure 1 shows. A distributed implementation broadcasts the input channel to all the output channels, and an address decoder picks off the sub-channel as its data fly past. Either way it's easy to broadcast a sub-channel to any number of output channels.

[14] R. Rettberg et al.: The Monarch parallel processor hardware design. *IEEE Computer* 23, 18-30 (1990)

Broadcast networks

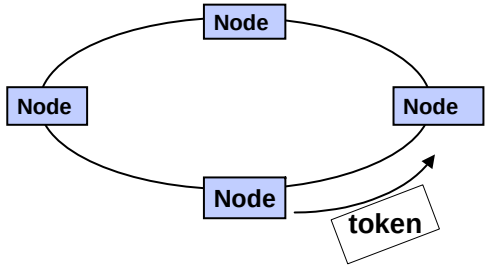
From the viewpoint of the preceding discussion of links, a broadcast network is a link that carries packets, roughly one at a time, and has lots of receivers, all of which see all the packets. Each packet carries a *destination address*, each receiver knows its own address, and a receiver's job is to pick out its packets. It's also possible to view a broadcast network as a special kind of switched network, taking the viewpoint of the next section.

Viewed as a link, a broadcast network has to solve the problems of arbitration and addressing. Addressing is simple, since all the receivers see all the packets. All that is needed is 'address filtering' in the receiver. If a receiver has more than one address the code for this may get tricky, but a simple, if costly, fallback position is for the receiver to accept all the packets, and rely on some higher-level mechanism to sort out which ones are really meant for it.

The tricky part is arbitration. A computer's I/O bus is an example of a broadcast network, and it is one in which each device requests service, and a central 'arbiter' *grants* bus access to one device at a time. In nearly all broadcast networks that are called networks, it is an article of religion that there is no central arbiter, because that would be a single point of failure, and another scheme would be required so that the distributed nodes could communicate with it[15]. Instead, the task is distributed among all the senders. As with link arbitration in general, there are two ways to do it: scheduling and contention.

Arbitration by scheduling: Token rings

Scheduling is deterministic, and the broadcast networks that use it are called 'token rings'. The idea is that each node is connected to two neighbors, and the resulting line is closed into a circle or ring by connecting the two ends. Bits travel around the ring in one direction. Except when it is sending or receiving its own packets, a node retransmits every bit it receives. A single 'token' circulates around the ring, and a node can send when the token arrives at the node. After sending one or more packets, the node regenerates the token so that the next node can send. When its packets have traveled all the way around the ring and returned, the node 'strips' them from the ring. This results in round-robin scheduling, although there are various ways to add priorities and semi-synchronous service.



[15] There are times when this religion is inappropriate. For instance, in a network based on cable TV there is a highly reliable place to put the central arbiter: at the head end (or, in a fiber-to-the-neighborhood system, in the fiber-to-coax converter. And by measuring the round-trip delays between the head end and each node, the head end can broadcast "node *n* can make its request now" messages with timing which ensures that a request will never collide with another request or with other traffic.

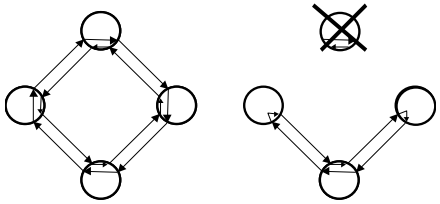


Fig. 3: A dual-attachment ring tolerates failure of one node

Rings are difficult to engineer because of the closure properties they need to have:

- *Clock synchronization*: each node transmits everything that it receives except for sync marks and its own packets. It's not possible to simply use the receive clock for transmitting because errors in decoding the clock will accumulate, so the node must generate its own clock. However, it must keep this clock very close to the clock of the preceding node on the ring to keep from having to add sync marks or buffer a lot of data.
- *Maintaining the single token*: with multiple tokens the broadcasting scheme fails. With no tokens, no one can send. So each node must monitor the ring. When it finds a bad state, it cooperates with other nodes to clear the ring and elect a 'leader' who regenerates the token. The strategy for election is that each node has a unique ID. A node starts an election by broadcasting its ID. When a node receives the ID of another node, it forwards it unless its own ID is larger, in which case it sends its own ID. When a node receives its own ID, it becomes the leader; this works because every other node has seen the leader's ID and determined that it is larger than its own. Compare this with the Paxos scheme for electing a leader (in handout 18).
- *Preserving the ring connectivity* in spite of failures. In a simple ring, the failure of a single node or link breaks the ring and stops the network from working at all. A 'dual-attachment' ring is actually two rings, which can run in parallel when there are no failures. If a node fails, splicing the two rings together as shown in figure 3 restores a single ring. Tolerating a single failure can be useful for a ring that runs in a controlled environment like a machine room, but is not of much value for a LAN where there is no reason to believe that only one node or link will fail. FDDI has dual attachment because it was originally designed as a machine room interconnect; today this feature adds complexity and confuses customers.
- A practical way to solve this problem is to connect all the nodes to a single 'hub' in a so-called 'star' configuration, as shown in figure 4. The hub detects when a node fails and cuts it out of the ring. If the hub fails, of course, the entire ring goes down, but the hub is a simple, special-purpose device installed in a wiring closet or machine room, so it's much less likely to fail than a node. The drawback of a hub is that it contains much of the hardware needed for the switches discussed in the next lecture, but doesn't provide any of the performance gains that switches do.

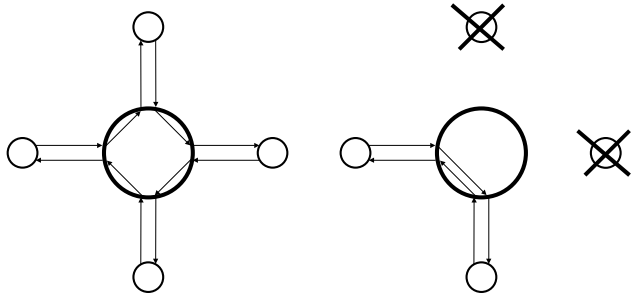


Fig. 4: A ring with a hub tolerates multiple failures

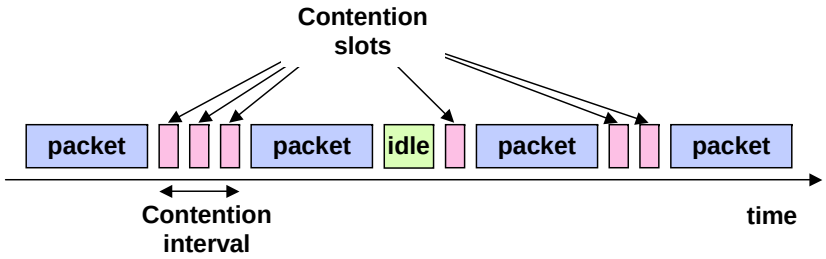
In spite of these problems, two token rings are in wide use (though much less wide than Ethernet, and rapidly declining): the IBM token ring and FDDI. In the case of the IBM token ring this happened because of IBM's marketing prowess; the salesmen persuaded bankers that they didn't want precious packets carrying dollars to collide on the Ethernet. In the case of FDDI it happened because most people were busy deploying Ethernet and developing Ethernet bridges and switches; the FDDI standard gained momentum before anyone noticed that it's not very good.

Arbitration by contention: Ethernet

Contention, using backoff, is probabilistic, as we saw when we discussed backoff on links. It wastes some bandwidth in unsuccessful transmissions. In the case of a broadcast LAN, bandwidth is wasted whenever two packets overlap at the receiver; this is called a 'collision'. How often does it happen?

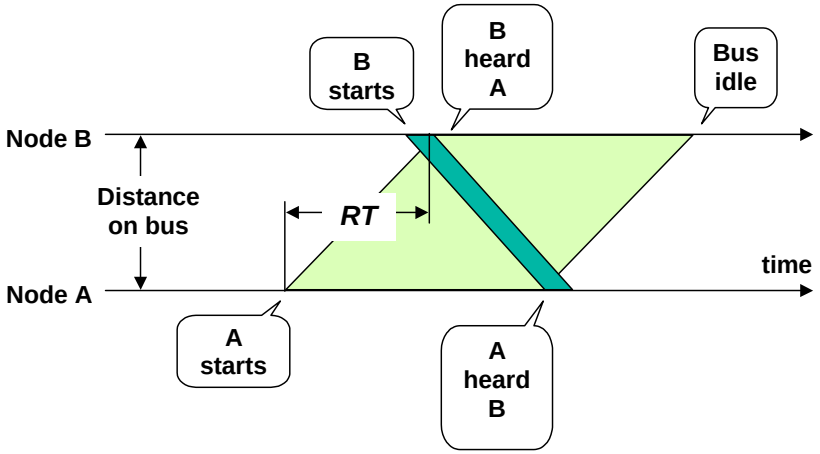
In a 'slotted Aloha' network a node can't tell that anyone else is sending; this model is appropriate for the radio transmission from feeble terminals to a central hub that was used in the original Aloha network. If everyone sends the same size packet (desirable in this situation because long packets are more likely to collide) and the senders are synchronized, we can think of time as a sequence of 'slots', each one packet long. In this situation exponential backoff gives an efficiency of $1/e = .37$ (see below).

If a node that isn't sending can tell when someone else is sending ('carrier sense'), then a potential sender can 'defer' to a current sender. This means that once a sender's signal has reached all the nodes without a collision, it has 'acquired' the medium and will be able to send the rest of its packet without further danger of collision. If a sending node can tell when someone else is sending ('collision detection') both can stop immediately and back off. Both carrier sense and collision detection are possible on a shared bus and are used in the Ethernet. They are also possible in a system with a head end that can hear all the nodes, even if the nodes can't hear each other: the head end sends a collision signal whenever it hears more than one sender.



The critical parameter for a 'CSMA/CD' (Carrier Sense Multiple Access/Collision Detection) network like the Ethernet is the round-trip time for a signal to get from one node to another and back; see the figure below. After a maximum round-trip time *RT* without a collision, a sender knows it has acquired the medium. For the Ethernet this time is about 50 μs = 64 bytes at the 10 Mbits/sec transmission time; this comes from a maximum diameter of 2 km = 10 μs (at 5 μs/km for signal propagation in cable), 10 μs for the time a receiver needs to read the 'preamble' of the packet and either synchronize with the clock or detect a collision, and 5 μs to pass through a maximum of two repeaters, which is 25 μs, times 2 for the round trip. A packet must be at least this long or the sender might finish sending it before detecting a collision, in which case it wouldn't know whether the transmission was successful.

The 100 Mbits/sec fast Ethernet has the same minimum packet size, and hence a maximum diameter of 5 μs, 10 times smaller. Gigabit Ethernet has a maximum diameter of .5 μs or 100 m. However, it normally operates in 'full-duplex' mode, in which a wire connects only two nodes and is used in only one direction, so that two wires are needed for each pair of nodes. With this arrangement only one node ever sends on a given wire, so there is no multiplexing and hence no need for arbitration. The CSMA/CD stuff is still in the standard because any change to the standard would mean a whole new standards process, during which lots of people would try to introduce their own crazy ideas. It's much faster and safer to leave in the unused features. In any case, the logic for CSMA/CD must be in the chips so that they can run at the slower speeds as well, in order to ensure that the network will still work no matter how it's wired up.



Here is how to calculate the throughput of Ethernet. If there are k nodes trying to send, p is the probability of one station sending, and r is the round trip time, then the probability that one of the nodes will succeed is $A = kp(1-p)^{k-1}$. This has a maximum at $p=1/k$, and the limit of the maximum for large k is $1/e = .37$. So if the packets are all of minimum length this is the efficiency. The expected number of tries is $1/A = e = 2.7$ at this maximum, including the successful transmission. The waste, also called the 'contention interval', is therefore $1.7r$. For packets of length l the efficiency is $l/(l + 1.7r) = 1/(1 + 1.7r/l) \sim 1 - 1.7r/l$ when $1.7r/l$ is small. The biggest packet allowed on the Ethernet is 1.5 Kbytes = 20 r, and this yields an efficiency of 91.5% for the maximum r. Most networks have a much smaller r than the maximum, and correspondingly higher efficiency.

But how do we get all the nodes to behave so that $p=1/k$? This is the magic of exponential backoff. A is quite sensitive to p, so if several nodes are estimating k too small they will fail and increase their estimate. With carrier sense and collision detect, it's OK to start the estimate at 0 each time as long as you increase it rapidly. An Ethernet node does this, doubling its estimate at each backoff by doubling its maximum backoff time, and making it smaller by resetting its backoff time to 0 after each successful transmission. Of course each node must chose its actual backoff time randomly in the interval [0 .. maximum backoff]. As long as all the nodes obey the rules, they share the medium fairly, with one exception: if there are very few nodes, say two, and one has lots of packets to send, it will tend to 'capture' the network because it always starts with 0 backoff, whereas the other nodes have experienced collisions and therefore has a higher backoff.

The TCP version of exponential backoff doesn't have the benefit of carrier sense or collision detection. On the other hand, routers have some buffering, so it's not necessary to avoid collisions completely. As a result, TCP has 'slow start'; it transmits slowly until it gets some acknowledgments, and then speeds up. When it starts losing packets, it slows down. Thus each sender's estimate of k oscillates around the true value (which of course is always changing as well).

All versions of backoff arbitration have the problem that a selfish sender that doesn't obey the rules can get more than its share. This isn't a problem for Ethernet because there are very few sources of interface chips, and each one has been careful engineered to behave correctly. For TCP there are similarly few sources of widely used code, but on the other hand the code can fairly easily be patched to misbehave. This doesn't have much benefit for clients in the Internet, however, since most traffic is from servers to clients. It might have some benefit for servers, but they are usually run by organizations that can be made to suffer if detected in misbehavior. So in both cases social mechanisms keep things working.

Since the Ethernet works by sharing a passive medium, a failing node can only cause trouble by 'babbling', transmitting more than the protocol allows. The most likely form of babbling is transmitting all the time, and Ethernet interfaces have a very simple way of detecting this and shutting off the transmitter.

Most Ethernet installations do not use a single wire with all the nodes attached to it. Although this configuration is possible, the hub arrangement shown in figure 5 is much more common (contrary to the expectations of the Ethernet's designers). An Ethernet hub just repeats an incoming signal to all the nodes. Hub wiring has three big advantages:

It's easier to run Ethernet wiring in parallel with telephone wiring, which runs to a hub.

The hub is a good place to put sensors that can measure traffic from each node and switches that can shut off faulty or suspicious nodes.

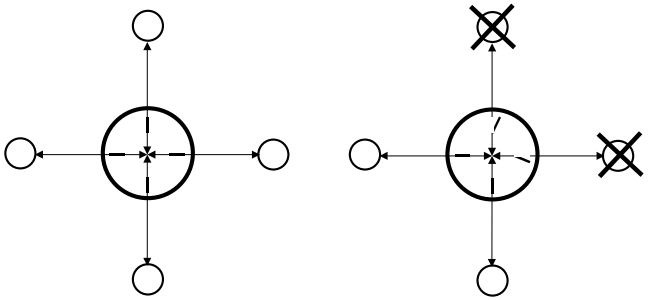


Fig. 5: An Ethernet with a hub can switch out failed nodes

Once wiring goes to a hub, it's easy to replace the simple repeating hub with a more complicated one that does some amount of switching and thus increases the total bandwidth. It's even possible to put in a multi-protocol hub that can detect what protocol each node is using and adjust itself accordingly. This arrangement is standard for fast Ethernet, which runs at 100 Mbits/sec instead of 10, but is otherwise very similar. A fast Ethernet hub automatically handles either speed on each of its ports.

A drawback is that the hub is a single point of failure. Since it is very simple, this is not a major problem. It would be possible to connect each network interface to two hubs, and switch to a backup if the main hub fails, but people have not found it necessary to do this. Instead, nodes that need very high availability of the network have two network interfaces connected to two different hubs.

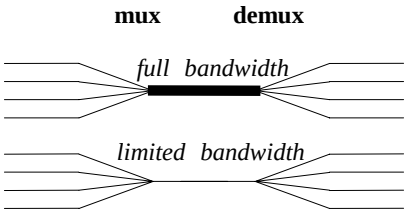
Switches

The modern trend in local area networks, however, is to abandon broadcast and replace hubs with switches. A switch has much more silicon than a hub, but silicon follows Moore's law and gets cheaper by 2x every 18 months. The cost of the wires, connectors, and packaging is the same, and there is much more aggregate bandwidth. Furthermore, a switch can have a number of slow ports and a few fast ones, which is exactly what you want to connect a local group of clients to a higher bandwidth 'backbone' network that has more global scope.

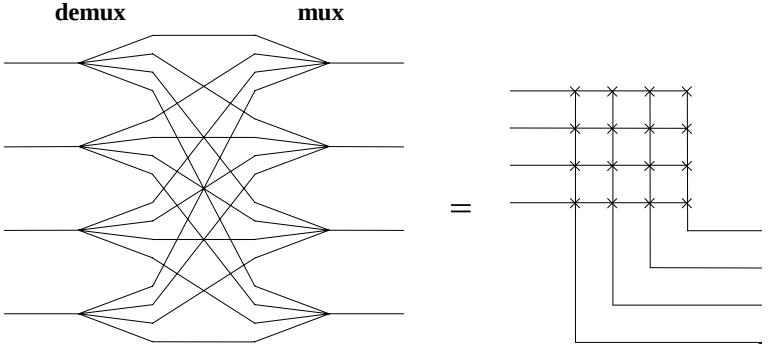
In the rest this handout we describe the different kinds of switches, and consider ways of connecting switches with links to form a larger link or switch.

A switch is a generalization of a multiplexer or demultiplexer. Instead of connecting one link to many, it connects many links to many. Figure 6(a) is the usual drawing for a switch, with the input links on the left and the output links on the right. We view the links as simplex, but usually they are paired to form full-duplex links so that every input link has a corresponding output link which sends data in the reverse direction. Often the input and output links are connected to the same nodes, so that the switch allows any node to send to any other.

(a) The usual representation of a switch



(b) Mux–demux code



(c) Demux–mux code, usually drawn as a crossbar

Fig. 6. Switches

A basic switch can be built out of multiplexers and demultiplexers in the two ways shown in figure 6(b) and 6(c). The latter is sometimes called a 'space-division' switch since there are separate multiplexers and demultiplexers for each link. Such a switch can accept traffic from every link provided each is connected to a different output link. With full-bandwidth multiplexers this restriction can be lifted, usually at a considerable cost. If it isn't, then the switch must arbitrate among the input links, generalizing the arbitration done by its component multiplexers, and if input traffic is not reordered the average switch bandwidth is limited to 58% of the maximum by 'head-of-line blocking'.[16]

Some examples reveal the range of current technology. The range in latencies for the LAN switches and IP routers is because they receive an entire packet before starting to send it on. For Email routers, latency is not usually considered important.

[16] M. Karol et al., Input versus output queuing on a space-division packet switch. *IEEE Transactions on Communications* **35**, 12 (Dec. 1987), pp 1347-1356.

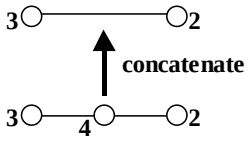


Fig. 7. Composing switches by concatenating.

Medium	*Link*	*Bandwidth*		*Latency*		*Links*
Pentium 4	register file	180	GB/s	.4	ns	6
Wires	Cray T3E	122	GB/s	1	µs	2K
LAN	Switched gigabit Ethernet	4	GB/s	5-100	µs	32
	Switched Ethernet	40	MB/s	100–1200	µs	32
IP router	many	1-6400	MB/s	50–5000	µs	16
Email router	SMTP	10-1000	KB/s	1-100	s	many
Copper pair	Central office	80	MB/s	125	µs	50K

Storage can serve as a switch of the kind shown in figure 6(b). The storage device is the common channel, and queues keep track of the addresses that input and output links should use. If the switching is coded in software, the queues are kept in the same storage, but sometimes they are maintained separately. Bridges and routers usually code their switches this way.

Pipelines

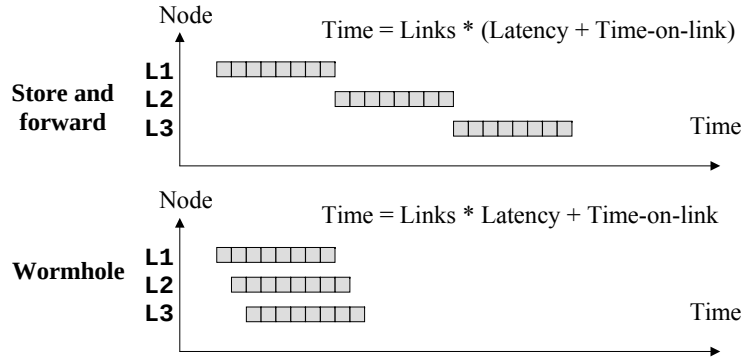
What can we make out of a collection of links and switches. The simplest thing to do is to concatenate two links using a connecting node, as in figure 7, making a longer link. This structure is sometimes called a 'pipeline'.

The only interesting thing about it is the rules for forwarding a single traffic unit:

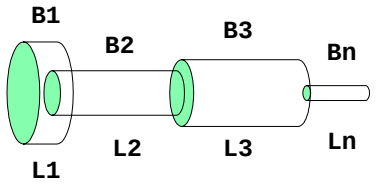
> Can the unit start to be forwarded before it is completely received ('wormholes' or 'cut-through')[17], and
>
> Can parts of two units be intermixed on the same link ('interleaving'), or must an entire unit be sent before the next one can start?

As we shall see, wormholes give better performance when the time to send a unit is not small, and often it is not because often a unit is an entire packet. Furthermore, wormholes mean that a switch need not buffer an entire packet.

[17] L. Ni and P. McKinley: A survey of wormhole routing techniques in direct networks. *IEEE Computer* 26, 62-76 (1993).



The latency of the composite link is the total delay of its component links (the time for a single bit to traverse the link) plus a term that reflects the time the unit spends entering links (or leaving them, which takes the same time). With no wormholes a unit doesn't start into link *i* until all of it has left link *i*-1, so this term is the sum of the times the unit spends entering each link (the size of the unit divided by the bandwidth of the link). With wormholes and interleaving, it is the time entering the slowest link, assuming that the granularity of interleaving is fine enough. With wormholes but without interleaving, each point where a link feeds a slower one adds the difference in the time a unit spends entering them; where a link feeds a faster one there is no added time because the faster link gobbles up the unit as fast as the slower one can deliver it.



`Latency = L1 + L2 + L3 + Ln`

This rule means that a sequence of links with increasing times is equivalent to the slowest, and a sequence with decreasing times to the fastest, so we can summarize the path as alternating slow and fast links $s_1\ f_1\ s_2\ f_2\ \ldots\ s_n\ f_n$ (where f_n could be null), and the entering time is the total time to enter slow links minus the total time to enter fast links. We summarize these facts:

Wormhole	*Interleaving*	*Time on links*
No	—	$\Sigma\ t_i$
Yes	No	$\Sigma\ ts_i - \Sigma\ tf_i = \Sigma\ (ts_i - tf_i)$
Yes	Yes	$\max\ t_i$

The moral is to use either wormholes or small units, and to watch out for alternating fast and slow links if you don't have interleaving. However, a unit shouldn't be too small on a variable TDM link because it must always carry the overhead of its address. Thus ATM cells, with 48 bytes of payload and 5 bytes of overhead, are about the smallest practical units (though the Cambridge slotted ring used cells with 2 bytes of payload). This is not an issue for fixed TDM, and indeed telephony uses 8 bit units.

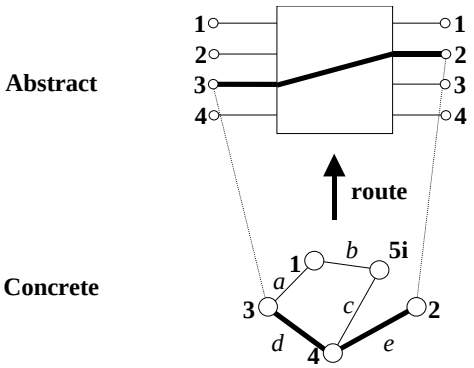


Fig. 8. Composing switches in a mesh.

There is no need to use wormholes for ATM cells, since the time to send 53 bytes is small in the intended applications. But Autonet, with packets that take milliseconds to transmit, uses wormholes, as do multiprocessors like the J-machine[18] which have short messages but care about every microsecond of latency and every byte of network buffering. The same considerations apply to pipelines.

Meshes

If we replace the connectors with switch nodes, we can assemble a mesh like the one in figure 8. The mesh can code the bigger switch above it; note that this switch has the same nodes on the input and output links. The heavy lines in both the mesh and the switch show the path from node 3 to node 2. The pattern of links between internal switches is called the 'topology' of the mesh. The figure is oversimplified in at least two ways: Any of the intermediate nodes might also be an end node, and the Internet has 300 million nodes rather than 4.

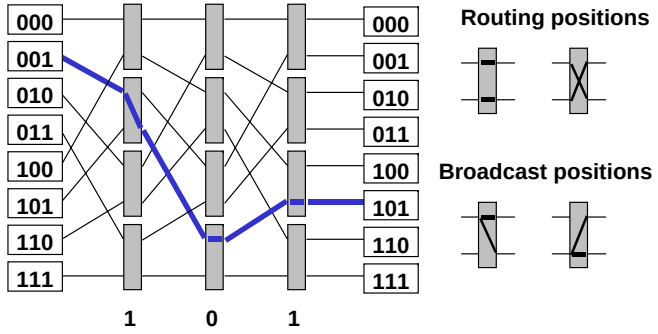
The new mechanism we need to make this work is *routing*, which converts an address into a 'path', a sequence of decisions about what output link to use at each switch. Routing is done with a map from addresses to output links at each switch. In addition the address may change along the path; this is coded with a second map, from input addresses to output addresses.

What spec does a mesh network satisfy? We saw earlier that a broadcast network provides unreliable FIFO delivery. In general, a mesh provides unreliable unordered delivery, because the routes can change, allowing one packet to overtake another, even if the links are FIFO. This is fine for IP on the Internet, which doesn't promise FIFO delivery. When switches are used to extend a broadcast LAN transparently, however, great care has to be taken in changing routes to preserve the FIFO property, even though it has very little value to most clients. This use of switching is called 'bridging'.

Addresses

There are three kinds of addresses. In order of increasing cost to code the maps, and increasing convenience to the end nodes, they are:

- *Source* addresses: the address is just the sequence of output links to use; each switch strips off the one it uses. In figure 8, the source addresses of node 2 from node 3 are (*d*, *e*) and (*a*, *b*, *c*, *e*).The IBM token ring and several multiprocessors (including the MIT J-machine and the Cosmic Cube[19]) use this. A variation distributes the source route across the path; the address (called a 'virtual circuit') is local to a link, and each switch knows how to map the addresses on its incoming links. ATM uses this variation. So does the 'shuffle-exchange' network shown below, in which 2^n inputs are switched to 2^n outputs with n levels of 2^{n-1} 2x2 switches. The switches in the ith level are controlled by the ith bit of the address.



- *Hierarchical* addresses: the address is hierarchical. Each switch corresponds to one node in the address tree and knows what links to use to get to its siblings, children, and parent. The Internet[20] and cascaded I/O busses use this.

- *Flat* addresses: the address is flat, and each switch knows what links to use for every address. Broadcast networks like Ethernet and FDDI use this; the code is easy since every receiver sees all the addresses and can just pick off those destined for it. Bridged LANs also use flat routing, falling back on broadcast when the bridges lack information about where an end-node address is. The mechanism for routing 800 telephone numbers is mainly flat.

Deadlock

Traffic traversing a composite link needs a sequence of resources (most often buffer space) to reach the end. Usually it acquires a resource while holding on to existing ones, since you need to get the next buffer before you can free the current one. This means that deadlock is possible. The left side of figure 9 shows the simplest case: two nodes with a single buffer pool in each, and links connecting them. If traffic must acquire a buffer at the destination before giving up its buffer at the source, it is possible for all the messages to deadlock waiting for each other to release their buffers.[21]

The simple rule for avoiding deadlock is well known (see handout 14): define a partial order on the resources, and require that a resource cannot be acquired unless it is greater in this order than all the resources already held. In our application it is usual to treat the links as resources and re-

[18] W. Dally: A universal parallel computer architecture. *New Generation Computing* 11, 227-249 (1993).
[19] C. Seitz: The cosmic cube. *Communications of the ACM* 28, 22-33 (1985)
[20] W. Stallings, IPV6: The new Internet protocol. IEEE Communications 34, 7 (Jul 1996), pp 96-109.
[21] Actually, this simple configuration can only deadlock if each node fills up with traffic going to the other node. This is very unlikely; usually some of the buffers will hold traffic for other nodes to the left or right, and this will drain out in time.

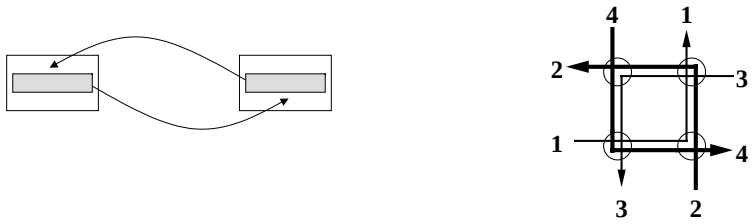


Fig. 9. Deadlock. The version on the left is simplest, but can't happen with more than 1 buffer/node

quire paths to be increasing in the link order. Of course the ordering relation must be big enough to ensure that a path exists from every sender to every receiver.

The right side of figure 9 shows what can happen even at one cell of a simple rectangular grid if this problem is ignored. The four paths use links as follows: 1—EN, 2—NW, 3—WS, 4—SE. There is no ordering that will allow all four paths, and if each path acquires its first link there is deadlock.

The standard order on a grid is: $l_1 < l_2$ iff they are head to tail, and either they point in the same direction, or l_1 goes east or west and l_2 goes north or south. So the rule is: "Go east or west first, then north or south." On a tree $l_1 < l_2$ iff they are head to tail, and either both go up toward the root, or l_2 goes down away from the root. The rule is thus "First up, then down." On a DAG impose a spanning tree and label all the other links up or down arbitrarily; the Autonet does this.

Note that this kind of rule for preventing deadlock may conflict with an attempt to optimize the use of resources by sending traffic on the least busy links.

Although figure 9 suggests that the resources being allocated are the links, this is a bit misleading. It is the buffers in the receiving nodes that are the physical resource in short supply. This means that it's possible to multiplex several 'virtual' links on a single physical link, by dedicating separate buffers to each virtual link. Now the virtual links are resources that can run out, but the physical links are not. The Autonet does not do this, but it could, and other mesh networks such as AN2[22] have done so, as do modern multiprocessor interconnects.

Topology

In the remainder of the handout, we study mechanisms for routing in more detail.[23] It's convenient to divide the problem into two parts: computing the topology of the network, and making routing decisions based on some topology. We begin with topology, in the context of a collection of links and nodes identified by index types `L` and `N`. A topology `T` specifies the nodes that each link connects. For this description it's not useful to distinguish routers from hosts or end-nodes, and indeed in most networks a node can play both roles.

These are simplex links, with a single sender and a single receiver. We have seen that a broadcast LAN can be viewed as a link with *n* senders and receivers. However, for our current purposes it is better to model it as a switch with 2*n* links to and from each attached node. Concretely, we can think of a link to the switch as the physical path from a node onto the LAN, and a link from the switch as the physical path the other way together with the address filtering mechanism.

[22] T. Anderson et al., High-speed switch scheduling for local area networks. *ACM Transactions on Computer Systems* **11**, 4 (Nov. 1993), pp 319-352.
[23] This is a complicated subject, and our treatment leaves out a lot. An excellent reference is R. Perlman, *Interconnections: Bridges and Routers* , Addison-Wesley, 1992. Chapter 4 on source routing bridges is best left unread.

Note that a path is not uniquely determined by a sequence of nodes (much less by endpoints), because there may be multiple links between two nodes. This is why we define a path as `SEQ L` rather than `SEQ N`. Note also that we are assuming a global name space `N` for the nodes; this is usually coded with some kind of UID such as a LAN address, or by manually assigned addresses like IP addresses. If the nodes don't have unique names, life becomes a lot more confusing.

We name links with local names that are relative to the sending node, rather than with global names. This reflects the fact that a link is usually addressed by an I/O device address. The link from a broadcast LAN node to another node connected to that LAN is named by the second node's LAN address.

```
MODULE Network[
    L                                                    % Link; local name
    N ]                                                  % Node; global name

TYPE Ns = SET N
    T  = N -> L -> N SUCHTHAT t.dom = {n | true}         % Topology; defined at each N
    P  = [n, r: SEQ L] WITH {"<=":=Prefix}               % Path starting at n
```

Here `t(n)(l)` is the node reached from node `n` on link `l`. For the network of figure 8,

```
t(3)(a) = 1
t(3)(d) = 4
t(1)(a) = 3
t(1)(b) = 5i
```

etc.

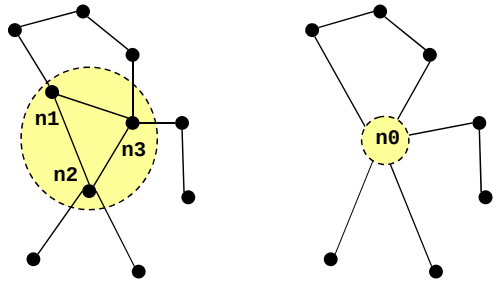
Note that a `T` is defined on every node, though there may not be any links from a node.

The `End` function computes the end node of a path. A `P` is actually a path if `End` is defined on it, that is, if each link actually exists. A path is acyclic if the number of distinct nodes on it is one more than the number of links. We can compute all the nodes on a path and all the paths between two nodes. All these notions only make sense in the context of a topology that says how the nodes and links are hooked up.

```
FUNC End(t, p) -> N = RET (p.r = {} => p.n [*] End(t, P{t(p.n)(p.r.head), p.r.tail

FUNC IsPath(t, p) -> Bool = RET End!(t, p)

FUNC Prefix(p1, p2) -> Bool = RET p1.n = p2.n /\ p1.r <= p2.r

FUNC Nodes(t, p) -> Ns = RET {p' | p' <= p || End(t, p'))

FUNC IsAcyclic(t, p) -> Bool = RET IsPath(t, p) /\ Nodes(t, p).size = p.r.size + 1

FUNC Paths(t, n1, n2)  -> SET p =
    RET {p | p.n = n1 /\ End(t, p) = n2 /\ IsAcyclic(t, p)}
```

Like anything else in computing, a network can be recursive. This means that a connected subnetwork can be viewed as a single node. We can collapse a topology to a smaller one in which a connected `ns` appears as a single representative node `n0`, by replacing all the links into `ns` with links to `n0` and discarding all the internal links. The outgoing links have to be named by pairs `[n, ll]`, since the naming scheme is local to a node; here we use `ll` for the 'lower-level' links

of the original T. Often collapsing is tied to hierarchical addressing, so that an entire subtree will be collapsed into a single node for the purposes of higher-level routing.



ns = {n1, n2, n3}
n0 IN ns

```
TYPE L = (L + [n, ll])

FUNC IsConnected(t, ns) -> Bool =
     RET (ALL n1 :IN ns, n2 :IN ns | (EXISTS p :IN Paths(t, n1, n2) ||
          Nodes(t, p) IN ns))

FUNC Collapse(t, ns, n0) -> T = n0 IN ns /\  IsConnected(t, ns) =>
     RET (\ n | (\ l |
          (    ~ n IN ns => (t(n)(l) IN ns => n0 [*] t(n)(l))
          [*] n = n0 /\ l IS [n, ll] /\ l.n IN n' /\ ~ t(l.n)(l.ll) IN ns =>
               t(l.n)(l.ll) ) ))
```

How does a network find out what its topology is? Aside from supplying it manually, there are two approaches. In both, each node learns which nodes are 'neighbors', that is, are connected to its links, by sending 'hello' messages down the links.

1. Run a global computation in which one node is chosen to learn the whole topology by becoming the root of a spanning tree. The root collects all the neighbor information and broadcasts what it has learned to all the nodes. The Autonet uses this method.

2. Run a distributed computation in which each node periodically tells its neighbors everything it knows about the topology. In time, any change in a node's neighbors will spread throughout the network. There are some subtleties about what a node should do when it gets conflicting information. The Internet uses this method, which is called 'link-state routing', and calls it OSPF.

In a LAN with many connected nodes, usually most are purely end-nodes, that is, do not do any switching of other people's packets. The end-nodes don't participate in the neighbor computation, since that would be an n^2 process. Instead, only the (few) routers on the LAN participate, and there is a separate scheme for the end-nodes. There are two mechanisms needed:

1. Routers need to know what end-nodes are on the LAN. This is just like finding out who is at the other end of the line, and it's done with hello messages, which for the LAN are broadcasts. Each end-node periodically broadcasts its IP address and LAN address, and the routers listen to these broadcasts and cache the results. The cache times out in a few broadcast intervals, so that obsolete information doesn't keep being used. Similarly, the routers broadcast the same information so that end-nodes can find out what routers are available.

Note that we treat the end-node—router and router—end-node cases separately, rather than doing a general node—node topology discovery, because the latter has n^2 cost and an end-nodes only needs to know about a few other end-nodes. When an end-node does need to know how to reach another end-node, it uses the ARP mechanism described below.

The Internet often doesn't do automatic topology discovery, however. Instead, information about the routers and end-nodes on a LAN is manually configured.

2. An end-node n1 needs to know which router can reach a node n2 that it wants to talk to; that is, n1 needs the value of sw(n1)(n2) defined below. To get it, n1 broadcasts n2 and expects to get back a LAN address. If node n2 is on the same LAN, it returns its LAN address. Otherwise a router that can reach n2 returns the router's LAN address. In the Internet this is done by the address resolution protocol (ARP). Of course n1 caches this result and times out the cache periodically.

The Autonet paper describes a variation on this, in which end-nodes use an ARP protocol to map Ethernet addresses into Autonet short addresses. This is a nice illustration of recursion in communication, because it turns the Autonet into a 'generic LAN' that is essentially an Ethernet, on top of which IP protocols will do another level of ARP to map IP addresses to Ethernet addresses.

Routing

For traffic to make it through the network, each switch must know which link to send it on. We begin by studying a simplified situation in which traffic is addressed by the N of its destination node. Later we consider the relationship between these globally unique addresses and real addresses.

A SW tells for each node how to map a destination node into a link[24] on which to send traffic; you can think of it as the dual of a topology, which for each node maps a link to a destination node. Then a route is a path that is chosen by sw.

```
TYPE SW  = N -> N -> L

PROC Route(t, sw, n1, n2) -> P = VAR p :IN Paths(t, n1, n2) |
     (ALL p' | p' <= p /\ p'.r # {} ==>
               p'.r.last = sw(End(t, p'{r := p'.r.reml})(n2)) => RET p
```

Here sw(n1)(n2) gives the link on which to reach n2 from n1. Note that if n1 = n2, the empty path is a possible result. There is nothing in this definition that says the route must be efficient. Of course, Route is not part of the code, but simply a spec.

We could generalize SW to N -> N -> SET L, and then

```
PROC Route(t, sw, n1, n2) -> SET P = RET {p :IN Paths(t, n1, n2) |
     (ALL p' | p' <= p /\ p'.r # {} ==>
               p'.r.last IN sw(End(t, p'{r := p'.r.reml})(n2))}
```

We want consistency between sw and t: the path sw chooses actually gets to the destination and is acyclic. Ideally, we want sw to choose a cheapest path. This is easy to arrange if everyone knows the topology and the Cost function. For concreteness, we give a popular cost function: the length of the path.

[24] or perhaps a set of links, though we omit this complication here.

```
FUNC IsConsistent(t, sw) -> Bool =
     RET ( ALL n1, n2 | Route(t, sw, n1, n2) IN Paths(t, n1, n2) )

FUNC IsBest(t, sw) -> Bool = VAR best := {p :IN Paths(t,n1,n2) || Cost(p)}.min |
     RET ( ALL n1, n2 | Cost(Route(t, sw, n1, n2)) = best )

FUNC Cost(p) -> Int = RET p.r.size                          % or your favorite
```

Don't lose sight of the fact that this is not code, but rather the spec for computing `sw` from `t`. Getting `t`, computing `sw`, and using it to route are three separate operations.

There might be more than one suitable link, in which case `L` is replaced by `SET L`, or by a function that gives the cost of each possible `L`. We work out the former:

```
TYPE SW  = N -> N -> SET L

PROC Routes(t, sw, n1, n2) -> SET P = RET { p :IN Paths(t, n1, n2) |
     (ALL p' | p' <= p /\ p'.r # {} ==>
               p'.r.last IN sw(End(t, p'{r := p'.r.reml})(n2)) }

FUNC IsConsistent(t, sw) -> Bool =
     RET ( ALL n1, n2 | Routes(t, sw, n1, n2) <= Paths(t, n1, n2) )

FUNC IsBest(t, sw) -> Bool = VAR best := {p :IN Paths(t,n1,n2) || Cost(p)}.min |
     RET ( ALL n1, n2 | (ALL p :IN Routes(t, sw, n1, n2) | Cost(p) = best) )
```

Addressing

In a broadcast network addressing is simple: since every node sees all the traffic, all that's needed is a way for each node to recognize its own addresses. In a mesh network the `sw` function in every router has to map each address to a link that leads there. The structure of the address can make it easy or hard for the router to do the switching, and for all the nodes to learn the topology. Not surprisingly, there are tradeoffs.

It's useful to classify addressing schemes as local (dependent on the source) or global (the same address works throughout the network), and as hierarchical or flat.

	Flat	*Hierarchical*
Local	—	Source routing Circuits = distributed source routing: route once, keep state in routers.
Global	LANs: router knows links to everywhere By broadcast By learning Fallback is broadcast, e.g. in bridges.	IP, OSI: router knows links to parent, children, and siblings.

Source routing is the simplest for the switches, since all work of planning the routes is unloaded on the sender and the resulting route is explicitly encoded in the address. The drawbacks are that the address is bigger and, more seriously, that changes to the topology of the network must be reflected in changes to the addresses.

Congestion control

As we have seen, we can view an entire mesh network as a single switch. Like any structure that involves multiplexing, it requires arbitration for its resources. This network-level arbitration is not the same as the link-level arbitration that is requires every time a unit is sent on a link. Instead, its purpose is to allocate the resources of the network as a whole. To see the need for network-level arbitration, consider what happens when some internal switch or link becomes overloaded. Once its buffers fill up, it will have to drop some traffic.

As with any kind of arbitration, there are two possibilities: scheduling, or contention and backoff. Scheduling can be done statically, by allocating a fixed bandwidth to a path or 'circuit' from a sender to a receiver. The telephone system works this way, and it does not allow traffic to flow unless it can commit all the necessary resources. A variation that is proposed for ATM networks is to allocate a maximum bandwidth for each path, but to overcommit the network resources and rely on traffic statistics to make it unlikely that the bluff will be called.

Alternatively, scheduling can be done dynamically by backpressure, as in the Autonet and AN2. We studied this method in connection with links, and the issues are the same in networks. One difference is that the round-trip time may be longer, so that more buffering is needed to support a given bandwidth. In addition, the round-trip time is usually much more variable, because traffic has to queue at each switch. Another difference is that because a circuit that is held up by backpressure may be tying up resources, deadlock is possible.

Contention and backoff are also similar in links and networks; indeed, one of the backoff links that we studied was TCP, which is normally coded on top of a network. When a link or switch is overloaded, it simply drops some traffic. The trouble signal is usually coded by timeout waiting for an ack. There have been a number of proposals for an explicit 'congested' signal, but it's difficult to ensure that this signal gets back to the sender reliably.

24. Network Objects

We have studied how to build up communications from physical signals to a reliable message channel defined by the `Channel` spec in handout 21 on distributed systems. This channel delivers bytes from a sender to a receiver in order and without loss or duplication as long as there are no failures; if there are failures it may lose some messages.

Usually, however, a user or an application program doesn't want reliable messages to and from a fixed party. Instead, they want access to a named object. A user wants to name the object with a World Wide Web URL (perhaps implicitly, by clicking on a hypertext link), and perhaps to pass some parameters that are supplied as fields of a form; the user expects to get back a result that can be displayed, and perhaps to change the state of the object, for instance, by recording a reservation or an order. A program may want the same thing, or it may want to call a procedure or invoke a method of an object.

In both cases, the object name should have universal scope; that is:

> It should be able to refer to an object on any computer that you can communicate with.
>
> It should refer to the same object if it is copied to any computer that you can communicate with.

As we learned when we studied naming, it's possible to encode method names and arguments into the name. For example, the URL

```
http://google.com/cgi-bin/query?&what=web&q=butler+lampson
```

could be written in Spec as `Google.Query("web", {"butler"; "lampson"})`. So we can write a general procedure call as a path name. To do this we need a way to encode and decode the arguments; this is usually called 'marshaling' and 'unmarshaling' in this context, but it's the same mechanism we discussed in handout 7.

So the big picture is clear. We have a global name space for all the objects we could possibly talk about, and we find a particular object by simply looking up its name, one component at a time. This summary is good as far as it goes, but it omits a few important things.

- *Roots*. The global name space has to be rooted somewhere. A Web URL is rooted in the Internet's Domain Name Space (DNS).
- *Heterogeneity*. There may be a variety of communication protocols used to reach an object, hardware architectures and operating systems implementing it, and programming languages using it. Although we can abstract the process of name lookup as we did in handout 12, by viewing the directory or context at each point as a function `N -> (D + V)`, there may be very different code for this lookup operation at different points. In a URL, for example, the host name is looked up in DNS, the next part of the name is looked up by the HTML server on that host, and the rest is passed to some program on the server.
- *Efficiency*. If we anticipate lots of references to objects, we will be concerned about efficiency. There are various tricks that we can use to make things run faster:

> Use specialized interfaces to look up a name. An important case of this is to pass a whole path name along to the lookup operation so that it can be swallowed in one gulp, rather than looking it up one simple name at a time.
>
> Cache the results of looking up prefixes of a name.
>
> Change the representation of an object name to make it efficient in a particular situation. This is called 'swizzling'. One example is to encode a name in a fixed size data structure. Another is to make it relative to a locally meaningful root, in particular, to make it a virtual address in the local address space.

- *Fault tolerance*. In general we need to deal with both volatile and stable (or persistent) objects. Volatile objects may disappear because of a crash, in which case there has to be a suitable error returned. Stable objects may be temporarily unreachable. Both kinds of objects may be replicated for availability, in which case we have to locate a suitable replica.
- *Location transparency*. Ideally, local and remote objects behave in exactly the same way. In fact, however, there are certainly performance differences, and methods of remote objects may fail because of communication failure or failure of the remote system.
- *Data types and encoding*. There may be restrictions on what types of values can be passed as parameters to methods, and the cost of encoding may vary greatly, depending on the encoding and on whether encoding is done by compiled code or by interpreting some description of the type.
- *Programming issues*. If the objects are typed, the type system must deal with evolution of the types, because in a big system it isn't practical to recompile everything whenever a type changes. If the objects are garbage collected, there must be a way to know when there are no longer any references to an object.

Another way of looking at this is that we want a system that is universal, that is, independent of the details of the code, in as many dimensions as possible.

Function	*Independent of*	*How*
Transport bytes	Communication protocol	Reliable messages
Transport meaningful values	Architecture and language	Encode and decode Stubs and pickles
Network references	Location, architecture, and language	Globally meaningful names
Request-response	Concurrency	Server: work queue Client: waiting calls
Evolution	Version of an interface	Subtyping
Fault tolerance	Failures	Replication and failover
Storage allocation	Failures, client programs	Garbage collection

There are lots of different kinds of network objects, and they address these issues in different ways and to different extents. We will look closely at two of them: Web URLs, and Modula-3

network objects. The former are intended for human consumption, the latter for programming, and indeed for fairly low level programming.

Web URLs

Consider again the URL

```
http://google.com/cgi-bin/query?&what=web&q=butler+lampson
```

It makes sense to view `http://google.com` as a network object, and an HTTP `Get` operation on this URL as the invocation of a `query` method on that object with parameters `(what:="web", q:="butler+lampson")`. The name space of URL objects is rooted in the Internet DNS; in this example the object is just the host named by the DNS name plus the port (which defaults to `80` as usual). There is additional multiplexing for the RPC server `cgi-bin`. This server finds the procedure to run by looking up `query` in a directory of scripts and running it.

HTTP is a request-response protocol. Internet TCP is the transport. This works in the most straightforward way: there is a new TCP connection for each HTTP operation (although a later version, HTTP 1.2, has provision for caching connections, which cuts the number of round trips and network packets by a factor of 3 when the response data is short). The number of instructions executed to do an invocation is not very important, because it takes a user action to cause an invocation.

In the invocation, all the names in the path name are strings, as are all the parameters. The data type of the response is always HTML. This, however, can contain other types. Initially GIF (for images) was the only widely supported type, but several others (for example, JPEG for images, Java and ActiveX for code) are now routinely supported. An arbitrary embedded type can be handled by dispatching a 'helper' program such as a Postscript viewer, a word processor, or a spreadsheet.

It's also possible to do a `Put` operation that takes an HTML value as a parameter. This is more convenient than coding everything into strings in a `Get`. Methods normally ignore parameters that they don't understand, and both methods and clients ignore the parts of HTML that they don't understand. These conventions provide a form of subtyping.

There is no explicit fault tolerance, though the Web inherits fault-tolerance for transport from IP and the ability to have multiple servers for an object from DNS. In addition, the user can retry a failed request. This behavior is consistent with the fact that the Web is used for casual browsing, so it doesn't really have to work. This usage pattern is likely to evolve into one that demands much higher reliability, and a lot of the code will have to change as well to support it.

Normally objects are persistent (that is, stored on the disk) and read-only, and there is no notion of preserving state from one operation to the next, so there is no need for storage allocation. There is a way to store server state in the client, using a data structure called a 'cookie'. Cookies are indexed by the URL of the server that made them, so that different servers don't step on each other's cookies. The user is responsible for getting rid of cookies when they are no longer needed, but since they are small, most people don't bother. Cookies are often used as pointers back to writeable state in the server, but there are no standard ways of doing this.

As everyone knows, the Web has been extremely successful. It owes much of its success to the fact that an operation is normally invoked by a human user and the response is read by the same user. When things go wrong, the user gives up, makes the best of it, or tries something else. It's

extremely difficult to write programs that use HTTP, because there are so many things that can happen besides the "normal" response. Another way of saying this is that the web doesn't have to work.

Modula-3 network objects

We now look at the Modula-3 network object system, which has an entirely different goal: to be used by programs. The things to be done are the same: name objects, encode parameters and responses, process request and response messages. However, most of the coding techniques are quite different. This system is described in the paper by Birrell et al. (handout 25). It addresses all of these issues in the table above except for fault-tolerance, and provides a framework for that as well. These network objects are closely integrated with Modula-3's strongly typed objects, which are similar to the typed objects of C++, Java, and other 'object-oriented' programming languages.

Before explaining network objects, it is appropriate to point out their pitfalls. The appeal of remote procedure call or network objects is that you can program a distributed computation exactly like a local one; the RPC mechanism abstracts out all the differences. In general, abstraction is a good thing, since it lets us ignore irrelevant detail. For example, a file system abstracts out many things about disk sizes, allocation, representation of indexes, etc. For the most part nothing important is lost in this abstraction; the most important aspect of disk performance, that sequential operations are much faster than random ones, maps fairly well to the same property within a single file.

Unfortunately, the same is often (perhaps even usually) not true for RPC. The most important aspects of distributed systems are non-negligible communication costs and partial failures. RPC abstracts away from these.

- It's just as easy to write and invoke a remote procedure for adding 1000 x1000 matrices as one for factoring an integer, even though the cost of transferring 24 Mbytes is many times the computation cost for the addition.
- When you write a remote call to update a bank balance on a machine in China, it's easy to ignore the possibility that the call may fail because the Chinese machine is down or has lost its connection to the Internet.

Many attempts to build distributed systems using RPC have come to grief by ignoring these realities. And unfortunately, they usually do so very late in their development, since a system will probably work just fine on small test cases. The moral is not that RPC is always a bad thing, but that you should approach it with caution.

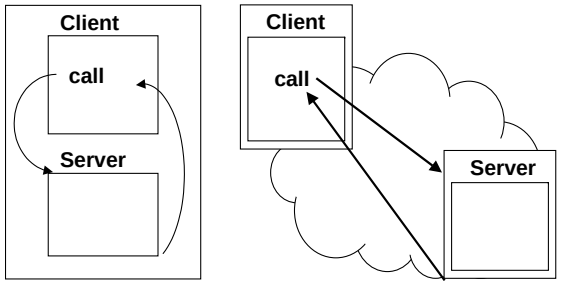
So much for pitfalls. Next, we ask: Why objects, rather than procedures? Because objects subsume the notions of procedure, interface, and reference/pointer. By an object we mean a collection of procedures that operate on some shared state; an object is just like a Spec module; indeed, its behavior can be defined by a Spec module. An essential property of an object is that there can be many codes for the same interface. This is often valuable in ordinary programming, but it's essential in a distributed system, because it's normal for different instances of the same kind of object to live on different machines. For example, two files may live on different file servers.

Although in principle every object can have its own procedures to implement its methods, normally there are lots of objects that share the same procedure code, each with its own state. A set of objects with the same code is often called a 'class'. The standard code for an object is a record

that holds its state along with a pointer to a record of procedures for the class. Indeed, Spec classes work this way.

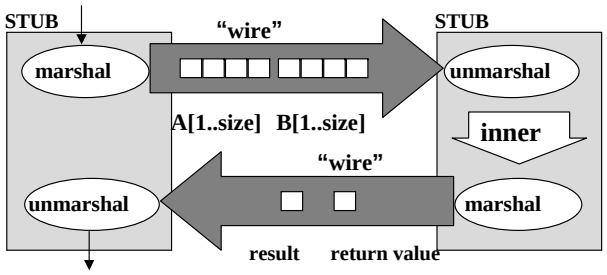
The basic idea

We begin with a spec for network objects. The idea is that you can invoke a method of an object transparently, regardless of whether it is local or remote.



If it's local, the code just invokes the method directly; if it's remote, the code sends the arguments in a message and waits for a reply that contains the result. A real system does this by supplying a 'surrogate' object for a remote object. The surrogate has the same methods as the local object, but the code of each method is a 'stub' or 'proxy' that sends the arguments to the remote object and waits for the reply. The source code for the surrogate class is generated by a 'stub generator' from the declaration of the real class, and then compiled in the ordinary way.

void inner(**long** size, **long** A[], **long** B[], **long** *result)



We can't do this in a general way in Spec. Instead, we change the call interface for methods to a single procedure `Call`. You give this procedure the object, the method, and the arguments (with type `Any`), and it gives back the result. This gives us clumsy syntax for invoking a method, `Call(o, "meth", args)` instead of `o.meth(args)`, and sacrifices static type checking, but it also gives us transparent invocation for local and remote objects.

An unrealistic form of this is very simple.

MODULE Object0 =

```
TYPE O           =  Method -> (PROC (Any) -> Any)      % Object
     Method      =  String                             % Method name

PROC Call(o, method, any) -> Any RAISES {failed} =
        RET o(method)(any)

END Object0
```


What's wrong with this is that it takes no account of what happens when there's a failure. The machine containing the remote object might fail, or the communication between that machine and the invoker might fail. Either way, it won't be possible to satisfy this spec. When there's a failure, we expect the caller to see a `failed` exception. But what about the method? It may not be invoked at all, or it may be invoked but no result returned to the caller, or it may still be running after the caller gets the exception. This third case can arise if the remote object gets the call message, but communication fails and the caller times out while the remote method is still running; such a call is called an 'orphan'. The following spec expresses all these possibilities; `Fork(p, a)` is a procedure that runs `p(a)` in a separate thread. We assume that the atomicity of the remote method when there's a failure (that is, how much of it gets executed) is already expressed in its definition, so we don't have to say anything about it here.

MODULE Object =

```
TYPE O           =  Method -> (PROC (Any) -> Any)      % Object
     Method      =  String                             % Method name

VAR  failure     :  Bool := false

PROC Call(o, method, any) -> Any RAISES {failed} =
        RET o(method)(any)
    [] failure =>
          BEGIN SKIP [] o(method)(any) [] Fork(o(method), any) END;
          RAISE failed

END Object
```

Now we examine basic code for this spec in terms of messages sent to and from the remote object. In the next two sections we will see how to optimize this code.

Our code is based on the idea of a `Space`, which you should think of as the global name of a process or address space. Each object and each thread is local to some space. An object's state is directly addressable there, and its methods can be directly invoked from a thread local to that space. We assume that we can send messages reliably between spaces using a channel `Ch` with the usual `Get` and `Put` procedures. Later on we discuss how to code this on top of standard networking.

For network objects to work transparently, we must be able to:

- Have a globally valid name for an object.
- Find its space from its global name.
- Convert between local and global names.

We go from global to local in order to find the object in its own space to invoke a method; this is sometimes called 'swizzling'. We go from local to global to send (a reference to) the object from its own space to another space; this is sometimes called 'unswizzling'.

Looking at the spec, the most obvious approach is to simply encode the value of an `O` to make it remote. But this requires encoding procedures, which is fraught with difficulty. The whole point

of a procedure is that it reads and changes state. Encoding a function, as long as it doesn't depend on global state, is just a matter of encoding its code, since given the code it can execute anywhere. Encoding a procedure is not so simple, since when it runs it has to read and change the same state regardless of where it is running. This means that the running procedure has to communicate with its state. It could do this with some low level remote read and write operations on state components such as bytes of memory. Systems that work this way are called 'distributed shared memory' systems,. The main challenge is to make the reads and writes efficient in spite of the fact that they involve network communication. We will study this problem in handout 30 when we discuss caching.

This is not what remote procedures or network objects are about, however. Instead of encoding the procedure code, we encode a *reference* to the object, and send it a message in order to invoke a method. This reference is a `Remote`; it is a path name that consists of the `Space` containing the object together with some local name for the object in that space. The local name could be just a `LocalObj`, the address of the object in the space. However, that would be rather fragile, since any mistake in the entire global system might result in treating an arbitrary memory address as the address of an object. It is prudent to name objects with meaningless object identifiers or `OId`'s, and add a level of indirection for exporting the name of a local object, `export: OId -> LocalObj`. We thus have `Remote = [space, oid]`.

We summarize the encoding and decoding of arguments and results in two procedures `Encode` and `Decode` that map between `Any` and `Data`, as described in handout 7. In the next section we discuss some of the details of this process.

```
MODULE NetObj =                                          % codes Object

TYPE O          =  (LocalObj + Remote)
     LocalObj   =  Object.O                              % A local object
     Remote     =  [space, oid]                          % Wire Rep for an object
     OId        =  Int                                   % Object Identifier
     Space      =  Int                                   % Address space

     Data       =  SEQ Byte
     CId        =  Int                                   % Call Identifier
     Req        =  [for: CId, remote, method, data]      % Request
     Resp       =  [for: CId,                 data]      % Response
     M          =  (Req + Resp)                          % Message

CONST r         :  Space := ...
     sendSR     := Ch.SR{s := r}

VAR  export     :  Space -> OId -> LocalObj              % One per space

PROC Call(o, method, any) -> Any RAISES {failed} =
     IF   o IS LocalObj => RET o(method)(any)
     [*]  VAR cid := NewCId(), to :=o.space |
          Ch.Put(sendSR{r := to}, Req{cid, o, method, Encode(any)});
          VAR m |
               IF   << (to, m) := Ch.Get(r); m IS Resp /\ m.for = cid => SKIP >>;
                    RET Decode(m.data)
               []   Timeout() => RAISE failed
               FI
     FI
```

After sending the request, `Call` waits for a response, which is identified by the right `CId` in the `for` field. If it hasn't arrived by the time `Timeout()` is true, `Call` gives up and raises `failed`.

Note the Spec hack: an atomic command that gets from the channel only the response to the current `cid`. Other threads, of course, might assign other `cid`'s and extract their responses from the same space-to-space channel. Code has to have this demultiplexing in some form, since the channel is between spaces and we are using it for all the requests and responses between those two spaces. In a real system the calling thread registers its `CId` and wait on a condition. The code that receives messages looks up the `CId` to find out which condition to signal and where to queue the response.

```
THREAD Server() =
     DO VAR m, from: Space, remote, result: Any |
          << (from, m) := Ch.Get(r); m IS Req => SKIP >>;
          remote := m.remote;
          IF remote.space = r => VAR local := export(r)(remote.oid) |
               result := local(m.method)(Decode(m.data));
               Ch.Put(sendSR{r := from}, Resp{m.for, Encode(result)})
          [*] ...                                        % not local object; error
          FI
     OD
```

Note that the server thread runs the method. Of course, this might take a while, but we can have as many of these server threads as we like. A real system has a single receiving thread, interrupt routine, or whatever that finds an idle server thread and gives it a newly arrived request to work on.

```
FUNC Encode(any ) -> Data = ...
FUNC Decode(data) -> Any  = ...

END NetObj
```

We have not discussed how to encode exceptions. As we saw when we studied the atomic semantics of Spec, an exception raised by a routine is just a funny kind of result value, so it can be coded along with the ordinary result. The caller checks for an exceptional result and raises the proper exception.

This module uses a channel `Ch` that sends messages between spaces. It is a slight variation on the perfect channel described in handout 20. This version delivers all the messages directed to a particular address, providing the source address of each one. We give the spec here for completeness.

```
MODULE PerfectSR[
     M,                                                  % Message
     A ] =                                               % Address

TYPE Q          =  SEQ M                                 % Queue: channel state
     SR         =  [s: A, r: A]                          % Sender - Receiver

VAR  q          := (SR -> Q){* -> {}}                    % all initially empty

APROC Put(sr, m)      = << q(sr) := q(sr) + {m} >>

APROC Get(r: A) -> (A, M) = << VAR sr, m | sr.r = r /\ m = q(sr).head =>
     q(sr) := q(sr).tail; RET (sr.s, m) >>

END PerfectSR

MODULE Ch = PerfectSR[NetObj.M, NetObj.Space]
```

Now we explain how types and references are handled, and then we discuss how the space-to-space channels are actually coded on top of a wide variety of existing communication mechanisms.

Types and references

Like the Modula 3 system described in handout 25, most RPC and network object systems have static type systems. That is, they know the types of the remote procedures and methods, and take advantage of this information to make encoding and decoding more efficient. In `NetObj` the argument and result are of type `Any`, which means that `Encode` must produce a self-describing `Data` result so that `Decode` has enough information to recreate the original value. If you know the procedure type, however, then you know the types of the argument and result, and `Decode` can be type specific and take advantage of this information. In particular, values can simply be encoded one after another, a 32-bit integer as 4 bytes, a record as the sequence of its component values, etc., just as in handout 7. The `Server` thread reads the object `remote` from the message and converts it to a local object, just as in `NetObj`. Then it calls the local object's `disp` method, which decodes the method, usually as an integer, and switches to method-specific code that decodes the arguments, calls the local object's method, and encodes the result.

This is not the whole story, however. A network object system must respect the object types, decoding an encoded object into an object of the same type (or perhaps of a supertype, as we shall see). This means that we need global as well as local names for object types. In fact, there are in general *two* local types for each global type `G`, one which is the type of local objects of type `G`, and another which is the type of remote objects of type `G`. For example, suppose there is a network object type `File`. A space that implements some files will have a local type `MyFile` for its code. It may also need a surrogate type `SrgFile`, which is the type of surrogate objects that are implemented by a remote space but have been passed to this one. Both `MyFile` and `SrgFile` are subtypes of `File`. As far as the runtime is concerned, these types come into existence in the usual way, because code that implements them is linked into the program. In Modula 3 the global name is the 'fingerprint' `FP` of the type, and the local name is the 'typecode' `TC` used by the local runtime for safe casts, garage collection, and other things. The stub code for the type registers the local-global mapping with the runtime in tables `FPtoTC: FP -> TC` and `TCtoFP: TC -> FP`.[1]

When a network object `remote` arrives and is decoded, there are three possibilities:

- It corresponds to a local object, because `remote.space = r`. The `export` table maps `remote.oid` to the corresponding `LocalObj`.
- It corresponds to an existing surrogate. The `surrogates` table keeps track of these in `surrogates: Space -> Remote -> LocalObj`. In handout 25 the `export` and `surrogates` tables are combined into a single `ObjTbl`.
- A new surrogate has to be created for it. For this to work we have to know the local surrogate type. If we pass along the global type with the object, we can map the global type to a local (surrogate) type, and then use the ordinary `New` to create the new surrogate.

Almost every object system, including Modula 3, allows a supertype (more general type) to be 'narrowed' to a subtype (more specific type). We have to know the smallest (most specific) type

[1] There's actually a kludge that maps the local typecode to the surrogate typecode, instead of mapping the fingerprint to both.

for a value in order to decide whether the narrowing is legal, that is, whether the desired type is a supertype of the most specific type. So the global type for the object must be its most specific type, rather than some more general one. If the object is coming from a space other than its owner, that space may not even have any local type that corresponds to the objects most specific type. Hence the global type must include the sequence of global supertypes, so that we can search for the most specific local type of the object.

It is expensive to keep track of the object's sequence of global types in every space that refers to it, and pass this sequence along every time the object is sent in a message. To make this cheaper, in Modula 3 a space calls back to the owning space to learn the global type sequence the first time it sees a remote object. This call is rather expensive, but it also serves the purpose of registering the space with the garbage collector (making the object 'dirty').

This takes care of decoding. To encode a network object, it must be in `export` so that it has an `OId`. If it isn't, it must be added with a newly assigned `OId`.

Where there are objects, there must be storage allocation. A robust system must reclaim storage using garbage collection. This is especially important in a distributed system, where clients may fail instead of releasing objects. The basic idea for distributed garbage collection is to keep track for each exported object of all the spaces that might have a reference to the object. A space is supposed to register itself when it acquires a reference, and unregister itself when it gives up the reference (presumably as the result of a local garbage collection). The owner needs some way to detect that a space has failed, so that it can remove that space from all its objects. The details are somewhat subtle and beyond the scope of this discussion.

Practical communication

This section is about optimizing the space-to-space communication provided by `PerfectSR`. We'd like the efficiency to be reasonably close to what you could get by assembling messages by hand and delivering them directly to the underlying channel. Furthermore, we want to be able to use a variety of transports, since it's hard to predict what transports will be available or which ones will be most efficient. There are several scales at which we may want to work:

- Bytes into or out of the channel.
- Data blocks into or out of the channel.
- Directly accessible channel buffers. Most channels will take bytes or blocks that you give them and buffer them up into suitable blocks (called packets) for transmission).
- Transmitting and receiving buffers.
- Setting up channels to spaces.
- Passing references to spaces.

At the lowest level, we need efficient access to a transport's mechanism for transmitting bytes or messages. This often takes the form of a 'connection' that transfers a sequence of bytes or messages reliably and efficiently, but is expensive to keep around. A connection is usually tied to a particular address space and, unlike an address, cannot be passed around freely. So our grand strategy is to map `Space -> Connection` whenever we do a call, and then send the message over the connection. Because this mapping is done frequently, it must be efficient. In the most

general case, however, when we have never talked to the space before, it's a lot of work to figure out what transports are available and set up a connection. Caching is therefore necessary.

The general mechanism we use is a sequence of mappings, `Space` to `SET Endpoint` to `Location` to `Connection`. The `Space` is globally unique, but has no other structure. It appears in every `Remote` and every surrogate object, so it must be optimized for space. An `Endpoint` is a transport-specific address; there is a set of them because a space may implement several transports. Because `Endpoint`'s are addresses, they are just bytes and can be passed freely in messages. A `Location` is an object; that is, it has methods that call the transport's code. Converting an `Endpoint` into a `Location` requires finding out whether the `Endpoint`'s transport is actually implemented here, and if it is, hooking up to the transport code. Finally, a `Location` object's `new` method yields a connection. The `Location` may cache idle connections or create new ones on demand, depending on the costs.

Consider the concrete example of TCP as the channel. An `Endpoint` is a DNS name or an IP address, a port number, and a UID for an address space that you can reach at that IP and port if it hasn't failed; this is just bits. The corresponding `Location` is an object whose `new` method generates a TCP connection to that space; it works either by giving you an existing TCP connection that it has cached, or by creating a new TCP connection to the space. A `Connection` is a TCP connection.

As we have seen, a `Space` is an abbreviation, translated by the `addrs` table. Thus `addrs: Space -> SET Endpoint`. We need to set up `addrs` for newly encountered `Space`'s, and we do this by callback to the source of the `Space`, maintaining the invariant: have `remote ==> addrs!(remote.space)`. This ensures that we can always invoke a method of the remote, and that we can pass on the space's `Endpoint`'s when we pass on the remote. The callback returns the set of `Endpoint`'s that can be used to reach the space.

An `Endpoint` should ideally be an object with a `location` method, but since we have to transport them between spaces, this would lead to an undesirable recursion. Instead, an `Endpoint` is just a string (or some binary record value), and a transport can recognize its own endpoints. Thus instead of invoking `endpoint.location`, we invoke `tr.location(endpoint)` for each transport `tr` that is available, until one succeeds and returns a `Location`. If a `Transport` doesn't recognize the `Endpoint`, it returns `nil` instead. If there's no `Transport` that recognizes the `Endpoint`, then the `Endpoint` is useless.

A `Connection` is a bi-directional channel that has the `SR` built in and has `M = Byte`; it connects a caller and a server thread (actually the thread is assigned dynamically when a request arrives, as we saw in `NetObject`). Because there's only one sender and one receiver, it's possible to stuff the parts of a message into the channel one at a time, and the caller does not have to identify itself but can take anything that comes back as the response. Thus the connection replaces `NetObj.CId`. The idea is that a TCP connection could be used directly as a `Connection`. You can make a `Connection` from a `Location`. The reason for having both is that a `Location` is just a small data structure, while a `Connection` may be much more expensive to maintain. A caller acquires a `Connection` for each call, and releases it when the call is done. The code can choose between creating and destroying connections on the one hand, and caching them on the other, based on the cost of creating one versus the cost of maintaining an idle one.

The byte stream code should provide multi-byte `Put` and `Get` operations for efficiency. It may also provide access to the underlying buffers for the stream, which might make encoding and decoding more efficient; this must be done in a transport-independent way. Transmitting and receiving the buffers is handled by the transport. We have already discussed how to obtain a connection to a given space.

Actually, of course, the channels are usually not perfect but only reliable; that is, they can lose messages if there is a crash. And even if there isn't a crash, there might be an indefinite delay before a message gets through. If you have a transactional queuing system the channels might be perfect; in other words, if the sender doesn't fail it will be able to queue a message. However, the response might be long delayed, and in practice there has to be a timeout after which a call raises the exception `CallFailed`. At this point the caller doesn't know for sure whether the call completed or not, though it's likely that it didn't. In fact, it's possible that the call might still be running as an 'orphan'.

For maximum efficiency you may want to use a specialized transport rather than a general one like TCP. Handout 11 describes one such transport and analyzes its efficiency in detail.

Bootstrapping

So far we have explained how to invoke a method on a remote object, and how to pass references to remote objects from one space to another. To get started, however, we have to obtain some remote objects. If we have a single remote directory object that maps names to objects, we can look up the names of lots of other objects there and obtain references to them. To get started, we can adopt the convention that each space has a special object with `OId 0` (zero) that is a directory. Given a `space`, we can forge `Remote{space, 0)` to get a reference to this object.

Actually we need not a `Space` but a `Location` that we can use to get a `Connection` for invoking a method. To get the `Location` we need an `Endpoint`, that is, a network address plus a well-known port number plus a standard unique identifier for the space. So given an address, say `www.microsoft.com`, we can construct a `Location` and invoke the lookup method of the standard directory object. If a server thread is listening on the well-known port at that address, this will work.

A directory object can act as a 'broker', choosing a suitable representative object for a given name. Several attempts have been made to invent general mechanisms for doing this, but usually they need to be application-specific. For example, you may want the closest printer to your workstation that has B-size paper. A generic broker won't handle this well.

25. Paper: Network Objects

The attached paper on network objects by Birrell, Nelson, Owicki, and Wobber is a fairly complete description of a working system. The main simplification is that it supports a single language, Modula 3, which is similar to Java. The paper explains most of the detail required to make the system reliable and efficient, and it gives the internal interfaces of the code.

6.826—Principles of Computer Systems 2006

.

26. Paper: Reliable Messages

The attached paper on reliable messages is Chapter 10 from the book *Distributed Systems: Architecture and Implementation*, edited by Sape Mullender, Addison-Wesley, 1993. It contains a careful and complete treatment of protocols for ensuring that a message is delivered at most once, and that if there are no serious failures it is delivered exactly once and its delivery is properly acknowledged.

6.826—Principles of Computer Systems 2006

.

27. Distributed Transactions

In this handout we study the problem of doing a transaction (that is, an atomic action) that involves actions at several different transaction systems, which we call the *resource managers* or RMs. The most obvious application is "distributed transactions": separate databases running on different computers. For example, we might want to transfer money from an account at Citibank to an account at Wells Fargo. Each bank runs its own transaction system, but we still want the entire transfer to be atomic. More generally, however, it is good to be able to build up a system recursively out of smaller parts, rather than designing the whole thing as a single unit. The different parts can have different code, and the big system can be built even though it wasn't thought of when the smaller ones were designed. For example, we might want to run a transaction that updates some data in a database system and some other data in a file system.

Specs

We have to solve two problems: composing the separate RMs so that they can do a joint action atomically, and dealing with partial failures. Composition doesn't require any changes in the spec of the RMs; two RMs that implement the `SequentialTr` spec in handout 19 can jointly commit a transaction if some third agent keeps track of the transaction and tells them both to commit. Partial failures do require changes in the resource manager spec. In addition, they require, or at least strongly suggest, changes in the client spec. We consider the latter first.

The client spec

In the code we have in mind, the client may be invoking `Do` actions at several RMs. If one of them fails, the transaction will eventually abort rather than committing. In the meantime, however, the client may be able to complete `Do` actions at other RMs, since we don't want each RM to have to verify that no other RM has failed before performing a `Do`. In fact, the client may itself be running on several machines, and may be invoking several `Do`'s concurrently. So the spec should say that the transaction can't commit after a failure, and can abort any time after a failure, but need not abort until the client tries to commit. Furthermore, after a failure some `Do` actions may report `crashed`, and others, including some later ones, may succeed.

We express this by adding another value `failing` to the phase. A crash sets the phase to `failing`, which enables an internal `CrashAbort` action that aborts the transaction. In the meantime a `Do` can either succeed or raise `crashed`.

```
CLASS DistSeqTr [
    V,                                                  % Value of an action
    S WITH { s0: ()->S }                                % State
    ] EXPORT Begin, Do, Commit, Abort, Crash =

TYPE A          =  S->(V, S)                            % Action

VAR  ss         := S.s0()                               % Stable State
     vs         := S.s0()                               % Volatile State
     ph         : ENUM[idle, run, failing] := idle      % PHase (volatile)
```



```
APROC Begin() = << Abort(); ph := run >>                % aborts any current trans.

APROC Do(a) -> V RAISES {crashed} = <<                  % non-deterministic if failing!
    IF  ph # idle => VAR v | (v, vs) := a(vs); RET v
    []  ph # run  => RAISE crashed
    FI >>

APROC Commit() RAISES {crashed} =
    << IF ph = run => ss := vs; ph := idle [*] Abort(); RAISE crashed FI >>

PROC Abort () = << vs := ss, ph := idle >>
PROC Crash () = << ph := failing >>

THREAD CrashAbort() = DO << ph = failing => Abort() >> OD

END DistSeqTr
```

In a real system `Begin` plays the role of `New` for this class; it starts a new transaction and returns its transaction identifier `t`, which is an argument to every other routine. Transactions can commit or abort independently (subject to the constraints of concurrency control). We omit this complication. Dealing with it requires representing each transaction's state change independently in the spec, rather than just letting them all update `vs`. If the concurrency spec is 'any can commit', for example, `Do(t)` sees `vs = ss + actions(t)`, and `Commit(t)` does `ss := ss + actions(t)`.

Partial failures

When several RMs are involved in a transaction, they must agree about whether the transaction commits. Thus each transaction commit requires consensus among the RMs.

The code that implements transactions usually keeps the state of a transaction in volatile storage, and only guarantees to make it stable at commit time. This is important for efficiency, since stable storage writes are expensive. To do this with several RMs requires a RM action to make a transaction's state stable without committing it; this action is traditionally called `Prepare`. We can invoke `Prepare` on each RM, and if they all succeed, we can commit the transaction. Without `Prepare` we might commit the transaction, only to learn that some RM has failed and lost the transaction state.

`Prepare` is a formalization of the so-called write-ahead logging in the old `LogRecovery` or `LogAndCache` code in handout 19. This code does a `Prepare` implicitly, by forcing the log to stable storage before writing the commit record. It doesn't need a separate `Prepare` action because it has direct and exclusive access to the state, so that the sequential flow of control in `Commit` ensures that the state is stable before the transaction commits. For the same reason, it doesn't need separate actions to clean up the stable state; the sequential flow of `Commit` and `Crash` takes care of everything.

Once a RM is prepared, it must maintain the transaction state until it finds out whether the transaction committed or aborted. We study a design in which a separate 'coordinator' module is responsible for keeping track of all the RMs and telling them to commit or abort. Real systems sometimes allow the RMs to query the coordinator instead of, or in addition to, being told what to do, but we omit this minor variation.

We first give the spec for a RM (not including the coordinator). Since we want to be able to compose RMs repeatedly, we give it as a modification (*not* an implementation) of the `DistSeqTr` client spec; this spec is intended to be called by the coordinator, not by the client (though, as we shall see, some of the procedures can be called directly by the client as an optimization). The

change from DistSeqTr is the addition of the stable 'prepared state' ps, and a separate Prepare action between the last Do and Commit. A transaction is prepared if ps # nil. Note that Crash has no effect on a prepared transaction. Abort works on any transaction, prepared or not.

```
TYPE T = (Coord + Null)                                  % Transaction id; see below for Coord

CLASS RMTr [
     V,                                                  % Value of an action
     S WITH { s0: ()->S },                               % State
     RM
     ] EXPORT Begin, Do, Commit, Abort, Prepare, Crash =

TYPE A           =  S->(V, S)                            % Action

VAR  ss          := S.s0()                               % Stable State
     ps          :  (S + Null) := nil                    % Prepared State (stable)
     vs          := S.s0()                               % Volatile State
     ph          : ENUM[idle, run, failing] := idle      % PHase (volatile)
     rm                                                  % the RM that contains self
     t           := nil                                  % "transaction id"; just for invariants

% INVARIANT ps # nil  ==>  ph = idle

APROC Begin(t') = << ph := run; t := t' >>

APROC Do(a) -> V RAISES {crashed} = <<                   % non-deterministic if ph=failing
     IF   ph # idle => VAR v | (v, vs) := a(vs); RET v
     []   ph # run => RAISE crashed
     FI >>

APROC Prepare() RAISES {crashed} =                       % called by coordinator
     << IF ph = run => ps := vs; ph := idle [*] RAISE crashed >>

APROC Commit() = <<                                      % succeeds only if prepared
     IF ps # nil => ss := ps; ps := nil [*] SKIP FI >>

PROC Abort () = << vs := ss, ph := idle; ps := nil >>
PROC Crash () = << IF ps = nil => ph := failing [*] SKIP >>

THREAD CrashAbort() = DO << ph = failing => Abort() >> OD

END RMTr
```

The idea of this spec is that its client is the coordinator, which implements DistSeqTr using one or more copies of RMTr. As we shall see in detail later, the coordinator

passes Do directly through to the appropriate RMTr,

does some bookkeeping for Begin, and

earns its keep with Commit by first calling Prepare on each RMTr and then, if all these are successful, calling Commit on each RMTr.

Optimizations discussed below allow the client to call Do, and perhaps Begin, directly. The RMTr spec requires its client to call Prepare exactly once before Commit. Note that because Do raises crashed if ph # idle, it raises crashed after Prepare. This reflects the fact that it's an error to do any actions after Prepare, because they wouldn't appear in ps. A real system might handle

these variations somewhat differently, for instance by raising tooLate for a Do while the RM is prepared, but the differences are inessential.

We don't give code for this spec, since it is very similar to LogRecovery or LogAndCache. Like the old Commit, Prepare forces the log to stable storage; then it writes a prepared record (coding ps # nil) so that recovery will know not to abort the transaction. Commit to a prepared transaction writes a commit record and then applies the log or discards the undo's. Recovery rebuilds the volatile list of prepared transactions from the prepared records so that a later Commit or Abort knows what to do. Recovery must also restore the concurrency control state for prepared transactions; usually this means re-acquiring their locks. This is similar to the fact that recovery in LogRecovery must re-acquire the locks for undo actions; in that case the transaction is sure to abort, while here it might also commit.

Committing a transaction

We have not yet explained how to code DistSeqTr using several copies of RMTr. The basic idea is simple. A coordinator keeps track of all the RMs that are involved in the transaction (they are often called 'workers', 'participants', or 'slaves' in this story). Normally the coordinator is also one of the RMs, but as with Paxos, it's easier to explain what's going on by keeping the two functions entirely separate. When the client tells the coordinator to commit, the coordinator tells all the RMs to prepare. This succeeds if all the Prepare's return normally. Then the coordinator records stably that the transaction committed, returns success to the client, and tells all the RMs to commit.

If some RM has failed, its Prepare will raise crashed. In this case the coordinator raises crashed to the client and tells all the RMs to abort. A RM that is not prepared and doesn't hear from the coordinator can abort on its own. A RM that is prepared cannot abort on its own, but must hear from the coordinator whether the transaction has committed or aborted. Note that telling the RMs to commit or abort can be done in background; the fate of the transaction is decided at the coordinator.

The abstraction function from the states of the coordinator and the RMs to the state of DistSeqTr is simple. We make RMTr a class, so that the type RMTr refers to an instance. We call it R for short. The RM states are thus defined by the RMTr class (which is an index to the RMTrs table of instances that is the state of the class (see section 7 of handout 4 for an explanation of how classes work in Spec).

The spec's vs is the combination of all the RM vs values, where 'combination' is some way of assembling the complete state from the pieces on the various RMs. Most often the state is a function from variables to values (as in the Spec semantics) and the domains of these functions are disjoint on the different RMs. That is, the state space is partitioned among the RMs. Then the combination is the overlay of all the RMs' vs functions. Similarly, the spec's ss is the combination of the RMs' ss unless ph = committed, in which case any RM with a non-nil ps substitutes that.

We need to maintain the invariant that any R that is prepared is in rs, so that it will hear from the coordinator what it should do.

```
CLASS Coord [

TYPE R           =  RMTr                                 % instance name on an RM
     Ph          =  ENUM[idle, commit]                   % PHase
```

```
CONST AtoRM      :  A -> RM := ...                    % the RM that handles action a

VAR  ph                                               % ph and rs are stable
     rs          :  SET R                             % the slave RMs
     finish      :  Bool                              % outcome is decided; volatile

ABSTRACTION
% assuming a partitioned state space, with S as a function from state component names to values; + combines these
     DistSeqTr.vs = + : rs.vs
     DistSeqTr.ss = (ph # commit => + : rs.ss
                      [*]  + : (rs * (\ r | (r.ps # nil => r.ps [*] r.ss))) )

INVARIANT
        r :IN rs ==> r.coord = self                   % slaves know the coordinators's id
     /\ {r | r.coord = self /\ r.ps # nil} <= rs      % r prepared => in rs

APROC Begin() = << ph := idle; rs := {}; finish := false >>

PROC Do(a) -> V RAISES {crashed} =
% abstractly r.begin=SKIP and rs is not part of the abstract state, so abstractly this is an APROC
     IF   ph = idle => VAR r := AtoR(a) |
          IF r = nil =>
               r := R.new(); r.rm := AtoRM(a); r.begin(self); rs + := r
          [*] SKIP FI;
          r.do(a)
     [*]  RAISE crashed FI

PROC Commit() RAISES {crashed} =
     IF ph = idle => VAR rs' |
          ForAllRs(R.prepare) EXCEPT crashed => Abort(); RAISE crashed;
          ph := commit; finish := true
     [*]  Abort(); RAISE crashed FI

PROC Abort() = finish := true

THREAD Finish() = finish =>                           % tell all RMs what happened
% It's OK to do this in background, after returning the transaction outcome to the client
     ForAllRs((ph = commit => R.commit [*] R.abort));
     ph := idle; rs := {}                             % clean up state; can be deferred

PROC Crash() = finish := true                         % rs and ph are stable

FUNC AtoR(a) -> (R + Null) = VAR r :IN rs | r.rm = AtoRM(a) => RET r [*] RET nil
% If we've seen the RM for this action before, return its r; otherwise return nil

PROC ForAllRs(p: PROC R->() RAISES {crashed}) RAISES crashed =
% Apply p to every RM in rs
     VAR rs' := rs | DO VAR r :IN rs' | p(r); rs' - := {r} OD

END Coord
```

We have written this in the way that Spec makes most convenient, with a class for `RMTr` and a class for `Coord`. The coordinator's identity (of type `Coord`) identifies the transaction, and each `RMTr` instance keeps track of its coordinator; the first invariant in `Coord` says this. In a real system, there is a single transaction identifier `t` that labels both coordinator and slaves, and you identify a slave by the pair `(rm, t)` where `rm` is the recourse manager that hosts the slave. We earlier defined `T` to be short for `Coord`:

This entire algorithm is called "two-phase commit"; do not confuse it with two-phase locking. The first phase is the prepares (the write-ahead logging), the second the commits. The coordinator can use any algorithm it likes to record the commit or abort decision. However, once some RM is prepared, losing the commit/abort decision will leave that RM permanently in limbo, uncertain whether to commit or abort. For this reason, a high-availability transaction system should use a high-availability way of recording the commit. This means storing it in several places and using a consensus algorithm to get these places to agree.

For example, you could use the Paxos algorithm. It's convenient (though not necessary) to use the RMs as the agents and the coordinator as leader. In this case the query/report phase of Paxos can be combined with the prepares, so no extra messages are required for that. There is still one round of command/report messages, which is more expensive than the minimum, non-fault-tolerant consensus algorithm, in which the coordinator just records its decision. But using Paxos, a RM is forced to block only if there is a network partition and it is on the minority side of the partition.

In the theory literature this form of consensus is called the 'atomic commitment' problem. We can state the validity condition for atomic commitment as follows: A crash of any unprepared RM does `Allow(abort)`, and when the coordinator has heard that every RM is prepared it does `Allow(commit)`. You might think that consensus is trivial since at most one value is allowed. Unfortunately, this is not true because in general you don't know which value it is.

Most real transaction systems do not use fault-tolerant consensus to commit, but instead just let the coordinator record the result. In fact, when people say 'two-phase commit' they usually mean this form of consensus. The reason for this sloppiness is that usually the RMs are not replicated, and one of the RMs is the coordinator. If the coordinator fails or you can't communicate with it, all the data it handles is inaccessible until it is restored from tape. So the fact that the outcome of a few transactions is also inaccessible doesn't seem important. Once RMs are replicated, however, it becomes important to replicate the commit result as well. Otherwise that will be the weakest point in the system.

Bookkeeping

The explanation above gives short shrift to the details of making the coordinator efficient. In particular, how does the coordinator keep track of the RMs efficiently. This problem has three aspects: enumerating RMs, noticing failed RMs, and cleaning up. The first two are caused by wanting to allow clients to talk directly to RMs for everything except commit and abort.

Enumerating RMs

The first is simply finding out who the RMs are, since for a single transaction the client may be spread out over many processes, and it isn't efficient to funnel every request to a RM through the coordinator as this code does. The standard way to handle this is to arrange all the client processes that are part of a single transaction in a tree, and require that each client process report to its parent the RMs that it or its children have talked to. Then the client at the root of the tree will know about all the RMs, and it can either act as coordinator itself or give the coordinator this information. The danger of running the coordinator on the client, of course, is that it might fail and leave the RMs hanging.

Noticing failed RMs

The second is noticing that an RM has failed during a transaction. In the `SequentialTr` or `DistSeqTr` specs this is simple: each transaction has a `Begin` that sets `ph := run`, and a failure

sets ph to some other value. In the code, however, since again there may be lots of client processes cooperating in a single transaction, a client doesn't know the first time it talks to a RM, so it doesn't know when to call Begin on that RM. One way to handle this is for each client process to send Begin to the coordinator, which then calls Begin exactly once on each RM; this is what Coord does. This costs extra messages, however, An alternative is to eliminate Begin and instead have both Do and Prepare report to the client whether the transaction is new at that RM, that is, whether ph = idle before the action; this is equivalent to having the RM tell the client that it did a Begin along with the Do or Prepare. If a RM fails, it will forget this information (unless it's prepared, in which case the information is stable), so that a later client action will get another 'new' report. The client processes can then roll up all this information. If any RM reports 'new' more than once, it must have crashed.

To make this precise, each client process in a transaction counts the number of 'new' reports it has gotten from each RM (here C names the client processes):

```
VAR news        : C -> R -> Int := {* -> 0}
```

We add to the RM state a history variable lost which is true if the RM has failed and lost some of the client's state. This is what the client needs to detect, so we maintain the invariant (here clientPrs(t) is the set of client processes for t):

```
( ALL r | r.t = t /\ r.lost ==>
        r.ph = idle /\ r.ps = nil)
    \/ (+ : {c :IN clientPrs(t) || news(c)(r)}) > 1 ) )
```

After all the RMs have prepared, they all have r.ps # nil, so if anything is lost is shows up in the news count. The second disjunct says that across all the client processes that are running t, r has reported 'new' more than once, and therefore must have crashed during the transaction. As with enumerating the RMs, we collect this information from all the client processes before committing.

A variation on this scheme has each RM maintain an 'incarnation id' or 'crash count' which is different each time it recovers, and report this id to each Do and Prepare. Then any RM that is prepared and has reported more than one id must have failed during the transaction. Again, the RM doesn't know this, but the coordinator does.

Cleaning up

The third aspect of bookkeeping is making sure that all the RMs find out whether the transaction committed or aborted. Actually, only the prepared RMs really need to find out, since a RM that isn't prepared can just abort the transaction if it is left in the lurch. But the timeout for this may be long, so it's usually best to inform all the RMs if it's convenient.

There's no problem if the coordinator doesn't crash, since it's cheap to maintain a volatile rs, although it's expensive to maintain a stable rs as Coord does. If rs is volatile, however, then the coordinator won't know who the RMs are after a crash. If the coordinator remembers the outcome of a transaction indefinitely this is OK; it can wait for the RMs to query it for the outcome after a crash. The price is that it can never foorget the outcome, since it has no way of knowing when it's heard from all the RMs. We say that a T with any prepared RMs is "pending", because some RM is waiting to know the outcome. If the coordinator is to know when it's no longer pending, so that it can forget the outcome, it needs to know (a superset of) all the prepared RMs and to hear that each of them is no longer prepared but has heard the outcome and taken the proper action.

So the choices appear to be either to record rs stably before Prepare, in which case Coord can forget the outcome after all the RMs know it, at the price of an ack message from each RM, or to remember the outcome forever, in which case there's no need for acks; Coord does the former. Both look expensive. We can make remembering the outcome cheap, however, if we can *compute* it as a function Presumed of the transaction id t for any transaction that is pending.

The simplest way to use these ideas is to make Presumed(t) = aborted always, and to make rs stable at Commit, rather than at the first Prepare, by writing it into the commit record. This makes aborts cheap: it avoids an extra log write before Prepare and any acks for aborts. However, it still requires each RM to acknowledge the Commit to the coordinator before the coordinator can forget the outcome. This costs one message from each RM to the coordinator, in addition to the unavoidable message from the coordinator to the RM announcing the commit. Thus presumed abort optimizes aborts, which is stupid since aborts are rare.

The only way to avoid the acks on commit is to make Presumed(t) = committed. This is not straightforward, however, because now Presumed is not constant. Between the first Prepare and the commit, Presumed(t) = aborted because the outcome after a crash is aborted and there's no stable record of t, but once the transaction is committed the outcome is committed. This is no problem for t.Commit, which makes t explicit by setting ph := commit (that is, by writing a commit record in the log), but it means that by the time we forget the outcome (by discarding the commit record in the log) so that t becomes presumed, Presumed(t) must have changed to committed.

This will cost a stable write, and to make it cheap we batch it, so that lots of transactions change from presumed abort to presumed commit at the same time. The constraint on the batch is that if t aborted, Presumed(t) can't change until t is no longer pending.

Now comes the tricky part: after a crash we don't know whether an aborted transaction is pending or not, since we don't have rs. This means that we can't change Presumed(t) to committed for any t that was active and uncommitted at the time of the crash. That set of t's has to remain presumed abort forever. Here is a picture that shows one way that Presumed can change over time:

PC | PA-live | future
PresumeCommitted
PC | PC | PA-live | future
Crash
PC | PC | PA+*ph=c* | PA-live | future
PresumeCommitted
PC | PC | PA+*ph=c* | PC | PA-live |future

Note that after the crash, we have a permanent section of presumed-abort transactions, in which there might be some committed transactions whose outcome also has to be remembered forever. We can avoid the latter by making rs stable as part of the Commit, which is cheap. We can avoid the permanent PA section entirely by making rs stable before Prepare, which is not cheap. The following table shows the various cost tradeoffs.

	Commit, no crash	*Commit, crash*	*Abort, no crash*	*Abort, crash*
`Coord`				
Stable `rs`	~~ph~~, ~~rs~~, acks	~~ph~~, ~~rs~~, acks	~~ph~~, ~~rs~~, acks	~~ph~~, ~~rs~~, acks
Presumed abort				
Volatile `rs` only	ph, ~~acks~~	ph, ~~acks~~	~~ph~~, ~~acks~~	~~ph~~, ~~acks~~
Stable `rs` on commit	~~ph~~, ~~rs~~, acks	~~ph~~, ~~rs~~, acks	~~ph~~, ~~acks~~	~~ph~~, ~~acks~~
Presumed commit				
Volatile `rs` only	~~ph~~, ~~acks~~	ph, ~~acks~~	~~ph~~, acks	ph, ~~acks~~
Stable `rs` on commit	~~ph~~, ~~rs~~, ~~acks~~	~~ph~~, ~~rs~~, acks	~~ph~~, acks	ph, ~~acks~~
Stable `rs`	~~ph~~, ~~rs~~, ~~acks~~	~~ph~~, ~~rs~~, acks	~~ph~~, ~~rs~~, acks	~~ph~~, ~~rs~~, acks

Legend: ~~abc~~ = never happens, ~~abc~~ = erased, abc = kept forever

For a more complete explanation of this efficient presumed commit, see the paper by Lampson and Lomet.[1]

Coordinating synchronization

Simply requiring serializability at each site in a distributed transaction system is not enough, since the different sites could choose different serialization orders. To ensure that a single global serialization order exists, we need stronger constraints on the individual sites. We can capture these constraints in a spec. As with the ordinary concurrency described in handout 20, there are many different specs we could give, each of which corresponds to a different class of mutually compatible concurrency control methods (but where two concurrency control methods from two different classes may be incompatible). Here we illustrate one possible spec, which is appropriate for systems that use strict two-phase locking and other compatible concurrency control methods.

Strict two-phase locking is one of many methods that serializes transactions in the order in which they commit. Our goal is to capture this constraint—that committed transactions are serializable in the order in which they commit—in a spec for individual sites in a distributed transaction system. This cannot be done directly, because commit decisions are made in a decentralized manner, so no single site knows the commit order. However, each site has some information about the global commit order. In particular, if a site hears that transaction `t1` has committed before it processes an operation for transaction `t2`, then `t2` must follow `t1` in the global commit order (assuming that `t2` eventually commits). Given a site's local knowledge, there is a set of global commit orders consistent with its local knowledge (one of which must be the actual commit order). Thus, if a site ensures serializability in all possible commit orders consistent with its local knowledge, it is necessarily ensuring serializability in the global commit order.

We can capture this idea more precisely in the following spec. (Rather than giving all the details, we sketch how to modify the spec of concurrent transactions given in handout 20.)

- Keep track of a partial order `precedes` on transactions, which should record that `t1 precedes t2` whenever the `Commit` procedure for `t1` happens before `Do` for `t2`. This can be done either by keeping a history variable with all external operations recorded (and defining `precedes` as a function on the history variable), or by explicitly updating `precedes` on each `Do(B)`, by adding all pairs `(tt, t2)` where `t1` is known to be committed.
- Change the constraint `Serializable` in the invariant in the spec to require serializability in all total orders consistent with `precedes`, rather that just some total order consistent with `xc`. Note that an order consistent with `precedes` is also externally consistent.

It is easy to show that the order in which transactions commit is one total order consistent with `precedes`; thus, if every site ensures serializability in every total order consistent with its local `precedes` order, it follows that the global commit order can be used as a global serialization order.

[1] B. Lampson and D Lomet, A new presumed commit optimization for two phase commit. *Proc. 19th VLDB Conference*, Dublin, 1993, pp 630-640.

28. Availability and Replication

This handout explains the basic issues in building highly available computer systems, and describes in some detail the specs and code for a replicated service with state.

What is availability?

A system is available if it delivers service promptly. Exactly what this means is something that has to be specified. For example, the spec might say that an ATM must deliver money from a local bank account to the user within 15 seconds, or that an airline reservation system must respond to user input within 1 second. The definition of availability is the fraction of offered load that gets prompt service; usually it's more convenient to measure the probability p that a request is not serviced promptly.

If requests come in at a certain rate, say 1/minute, with a memoryless distribution (that is, what happens to one request doesn't depend on other requests; a tossed coin is memoryless, for example), then p is also the probability that not all requests arriving in one minute get service. If this probability is small then the time between bad minutes is $1/p$ minutes. This is called the 'mean time to failure' or MTTF; sometimes 'mean time between failures' or MTBF is used instead. Changing the time scale of course doesn't change the MTTF: the probability of a bad hour is $60p$, so the time between bad hours is $1/60p$ hours = $1/p$ minutes. If p = .00001 then there are 5 bad minutes per year. Usually this is described as 99.999% availability, or '5-nines' availability.

The definition of 'available' is important. In a big system, for example, something is always broken, and usually we care about the service that one stream of customers sees rather than about whether the system is perfect, so we use the availability of one terminal to measure the MTTF. If you are writing or signing a contract, be sure that you understand the definition.

We focus on systems that fail and are repaired. In the simplest model, the system provides no service while it is failed. After it's repaired, it provides perfect service until it fails again. If MTTF is the mean time to failure and MTTR is the mean time to repair, then the availability is

$$p = \text{MTTR}/(\text{MTTF} + \text{MTTR})$$

If MTTR/MTTF is small, we have approximately

$$p = \text{MTTR}/\text{MTTF}$$

Thus the important parameter is the ratio of repair time to uptime. Note that doubling MTTF halves p, and so does halving the MTTR. The two factors are equally important. This simple point is often overlooked.

Redundancy

There are basically two ways to make a system available. One is to build it out of components that fail very seldom. This is good if you can do it, because it keeps the system simple. However, if there are n components and each fails independently with small probability p_c, then the system fails with probability $n\,p_c$. As n grows, this number grows too. Furthermore, it is often expensive to make highly reliable components.

The other way to make a system available is to use redundancy, so that the system can work even if some of its components have failed. There are two main patterns of redundancy: retry and replication.

Retry is redundancy in time: fail, repair, and try again. If failures are intermittent, repair doesn't require any action. In this case 1/MTTF is the probability of failure, and MTTR is the time required to detect the failure and try again. Often the failure detector is a timeout; then the MTTR is the timeout interval plus the retry time. Thus in retry, timeouts are critical to availability.

Replication is physical redundancy, or redundancy in space: have several copies, so that one can do the work even if another fails. The most common form of replication is 'primary-backup' or 'hot standby', in which the system normally uses the primary component, but 'fails over' to a backup if the primary fails. This is very much like retry: the MTTR is the failover time, which is the time to detect the failure plus the time to make the backup live. This is a completely general form of redundancy. Error correcting codes are a more specialized form. Two familiar examples are the Hamming codes used in RAM and the parity used in RAID disks.

These examples illustrate the application-dependent nature of specialized replication. A Hamming code needs $\log n$ check bits to protect $n - \log n$ data bits. A RAID code needs 1 check bit to protect any number of data bits. Why the difference? The RAID code is an 'erasure code'; it assumes that a data bit can have one of three values: 0, 1, and `error`. Parity means that the `xor` of all the bits is 0, so that any bit is equal to the `xor` of all the other bits. Thus any single `error` bit can be reconstructed from the others. This scheme is appropriate for disks, where there's already a very strong code detecting errors in a single sector. A Hamming code, on the other hand, needs many more check bits to detect which bit is bad as well as provide its correct value.

Another completely general form of replication is to have several replicas that operate in lockstep and interact with the rest of the world only between steps. At the end of each step, compare the outputs of the replicas. If there's a majority for some output value, that value is the output of the replicated system, and any replica that produced a different value is declared faulty and should be repaired. At least three replicas are needed for this to work; when there are exactly three it's called 'triple modular redundancy', TMR for short. A common variation that simplifies the handling of outputs is 'pair and spare', which uses four replicas arranged in two pairs. If the outputs of a pair disagree, it is declared faulty and the other pair's output is the system output.

A computer system has three major components: processing, storage, and communication. Here is how to apply redundancy to each of them.

- In communication, intermittent errors are common and retry is simply retransmitting a message. If messages can take different paths, even the total failure of a component often looks like an intermittent error because a retry will use different components. It's also possible to use error-correcting codes (called 'forward error correction' in this context), but usually the error rate is low enough that this isn't cost effective.
- In storage, retry is not so easy but error correcting codes still work well. ECC memory using Hamming codes, the elaborate codes used on disk drives, and RAID disks are all examples of this. Straightforward replication, usually called 'mirroring', is also popular.
- In processing, error correcting codes usually can't handle arbitrary state transitions. Retry is only possible if you have the old state, so it's usually coded in a transaction system. The replicated state machines that we studied in handout 18 are fully general, however, and can make any kind of processing highly available. Using these methods to replicate a processor at

the instruction set level is tricky but possible.[1] People also use lockstep replication at the instruction level, usually pair-and-spare, but such systems can't use standard components above the chip level, and it's very expensive to engineer them without single points of failure. As a result, they are expensive and not very successful.

War stories

Availability is a property of an entire system, hardware, software, and operations. There are lots of ways that things can go wrong. It's instructive to study some examples.

Ariane crash

The first flight of the European Space Agency's Ariane 5 rocket self-destructed 40 seconds into the flight. The sequence of events that led to this $400 million failure is instructive. In reverse temporal order, it is roughly as follows, as described in the report of the board of inquiry.[2]

1. The vehicle self-destructed because the solid fuel boosters started to separate from the main vehicle. This decision to self-destruct was part of the design and was carried out correctly.
2. The boosters separated because of high aerodynamic loads resulting from an angle of attack of more than 20 degrees.
3. This angle of attack was caused by full nozzle deflections of the solid boosters and the main engine.
4. The nozzle deflections were commanded by the on board computer (OBC) software on the basis of data transmitted by the active inertial reference system (SRI 2; the abbreviation is from the French for 'inertial reference system'). Part of the data for that time did not consist of proper flight data, but rather showed a diagnostic bit pattern of the computer of SRI 2, which was interpreted by the OBC as flight data.
5. SRI 2 did not send correct flight data because the unit had declared a failure due to a software exception.
6. The OBC could not switch to the back-up SRI (SRI 1) because that unit had already ceased to function during the previous data cycle (72-millisecond period) for the same reason as SRI 2.
7. Both units shut down because of uncaught internal software exceptions. In the event of any kind of exception, according to the system spec, the failure should be indicated on the data bus, the failure context should be stored in an EEPROM memory (which was recovered and read out), and, finally, the SRI processor should be shut down. This duly happened.
8. The internal SRI software exception was caused during execution of a data conversion from a 64-bit floating-point number to a 16-bit signed integer value. The value of the floating-point number was greater than what could be represented by a 16-bit signed integer. The result was an operand error. The data conversion instructions (in Ada code) were not protected from causing operand errors, although other conversions of comparable variables in the same place in the code were protected. It was a deliberate design decision not to protect this conversion, made because the protection is not free, and analysis had shown that overflow was impossible. In retrospect, of course, we know that the analysis was faulty; since it was not preserved, we don't know what was wrong with it.
9. The error occurred in a part of the software that controls only the alignment of the strap-down inertial platform. The results computed by this software module are meaningful only before liftoff. After liftoff, this function serves no purpose. The alignment function is operative for 50 seconds after initiation of the flight mode of the SRIs. This initiation happens 3 seconds before liftoff for Ariane 5. Consequently, when liftoff occurs, the function continues for approximately 40 seconds of flight. This time sequence is based on a requirement of Ariane 4 that is not shared by Ariane 5. It was left in to minimize changes to the well-tested Ariane 4 software, on the grounds that changes are likely to introduce bugs.
10. The operand error occurred because of an unexpected high value of an internal alignment function result, called BH (horizontal bias), which is related to the horizontal velocity sensed by the platform. This value is calculated as an indicator for alignment precision over time. The value of BH was much higher than expected because the early part of the trajectory of Ariane 5 differs from that of Ariane 4 and results in considerably higher horizontal velocity values. There is no evidence that any trajectory data were used to analyze the behavior of the unprotected variables, and it is even more important to note that it was jointly agreed not to include the Ariane 5 trajectory data in the SRI requirements and specifications.

It was the decision to shut down the processor that finally proved fatal. Restart is not feasible since attitude is too difficult to recalculate after a processor shutdown; therefore, the SRI becomes useless. The reason behind this drastic action lies in the custom within the Ariane program of addressing only random hardware failures. From this point of view, exception- or error-handling mechanisms are designed for random hardware failures, which can quite rationally be handled by a backup system. But a deterministic bug in software will happen in the backup system as well.

Maxc/Alto memory

The following extended saga of fault tolerance in computer RAM happened to my colleagues in the Computer Systems Laboratory of the Xerox Palo Alto Research Center. Many other people have had some of these experiences.

One of the lab's first projects (in 1971) was to build a time-sharing computer system named Maxc. Intel had just started to sell a 1024-bit semiconductor RAM chip[3], the Intel 1103, and it promised to be a cheap and reliable way to build the main memory. Of course, since it was new, we didn't know whether it would really work. However, we knew that for about 20% overhead we could use Hamming codes to implement single error correction and double error detection, so that the memory system would work even if individual chips hard a rather high failure rate. We did this, and the memory was solid as a rock. We never saw any failures, or even any double errors.

When the time came to design the Alto personal workstation in 1972, we used the same 1103 chips, and indeed the same memory boards. However, the Alto memory was much smaller (128

[1] Hypervisor-based fault tolerance, T. Bressoud and F. Schneider; *ACM Transactions on. Computing Systems* **14**, 1 (Feb. 1996), pp 80 – 107.
[2] This report is a model of clarity and conciseness. You can find it at http://www.esrin.esa.it/htdocs/tidc/Press/Press96/ariane5rep.html and a summary at http://www.siam.org/siamnews/general/ariane.htm.
[3] One million times smaller than the tate-of-the-art RAM chip of 2002.

KB instead of 3 MB) and had 16 bit words rather than the 40 bit words of Maxc. As a result, error correction would have added much more overhead, so we left it out; we did provide a parity bit for each word. For about 6 months the machines performed flawlessly, running a fairly vanilla minicomputer operating system that we had built, which provided a terminal on the screen that emulated a teletype.

It was only when we started to run the Bravo full-screen editor (the prototype for Microsoft Word) that we started to get parity errors. These errors were puzzling, because the chips were identical to those used without incident in Maxc. When we looked closely at the Maxc system, however, we discovered that although the ECC circuits had been designed to report both corrected errors and uncorrectable errors, the software logged only uncorrectable errors; corrected errors were being ignored. When logging of corrected errors was implemented, it turned out that the 1024-bit chips were actually failing quite often, and the error-correction circuitry was working hard to set things right.[4]

Investigation revealed that 1103's are pattern-sensitive: sometimes a bit will flip when the values of surrounding bits are just so. The reason we didn't see them on the Alto in the first 6 months is that you just don't get enough patterns on a single-user machine that isn't being very heavily used. Bravo put up lots of interesting stuff on the screen, which used about half the main memory to store values for its pixels, and thus Bravo made enough different patterns to tickle the chips. With some effort, we were able to write memory test programs that ran on the Alto, using lots of random test patterns, and also found errors. We never saw these errors in the routine testing that we did when the boards were manufactured.

Lesson: Fault-tolerant systems tend to become fault-intolerant, because faults that are tolerated don't get fixed. It's essential to monitor the faults and repair the faulty components even though the system is still working perfectly. Without monitoring, there's no way to know whether the system is operating with a large or a small safety margin.

When we built the Alto 2 two years later in 1975, we used 4k RAM chips, and because of the painful experience with the 1103, we did put in error correction. The machine worked flawlessly. Two years later, however, we discovered that in one-quarter of the memory, neither error correction nor parity was working at all. The chips were much better that 1103's, and in addition, many single-bit errors don't actually cause any observed failure of the software. On Alto 1 we knew about every single-bit error because of the parity. On Alto 2 in 1/4 of the memory we didn't know. Perhaps there were some failures that had no visible impact. Perhaps there were failures that crashed programs, but they were attributed to bugs in the software.

Lesson: To test a fault-tolerant system, you have to inject all the faults the system is supposed to tolerate. You also need to detect all faults, and you have to test the detection mechanism as well.

I believe this is why most PC manufacturers don't put parity on the memory: it isn't really needed because chips are pretty reliable, and if parity errors are reported the PC manufacturer gets blamed, whereas if random things happen Microsoft gets blamed.

[4] A couple of years later we had a similar problem with Maxc. In early January people noticed that the machine seemed to be slow. After a while, someone looked at the console log and discovered that over the holidays the memory had developed a permanent double (uncorrectable) error. The software found this error and reconfigured the memory without the bad region; this excluded one quarter of the memory from the running system, which considerably increased the amount of paging. Normally no one looked at the console log, so no one knew that this had happened.

Lesson: Beauty is in the eye of the beholder. The various parties involved in the decisions about how much failure detection and recovery to code do not always have the same interests.

Replication

In the remainder of this handout we present specs and code for a variety of replication techniques. We start with two specs of a "strongly consistent" replicated service, which looks almost like a single copy to its clients. The complication is that some client requests can fail; the second spec constrains the failure behavior more than the first. Then we give two codes, one based on primary copy and the other based on voting. Finally, we give a spec of a "loosely consistent" or "eventually consistent" service, which is much weaker but allows much cheaper highly available code.

Specs for consistent replication

A consistent service executes actions just like a non-replicated service: each action is executed at most once, and all clients see the same sequence of actions. However, the response to a client's request for an action can also be that the action "failed"; in this case, the client does not know whether or not the action was actually done. The client may be able to figure out whether or not it was done by executing more actions (for example, if the action leaves an unambiguous record in the state, such as a sequence number), but the `failed` response gives no information. The idea is that a `failed` response may be caused by failure of the replica doing the action, or of the communication channel between the client and the service.

The first spec places no constraints on the timing of failed actions. If a client requests an action and receives a `failed` response, the action may be performed at any later time. In addition, a `failed` response can be generated at any time.

The second spec still allows actions with `failed` responses to happen at any later time. However, it allows a `failed` response only if the system fails (or is recovering from a failure) during the execution of an action.

In practice, some constraints on when failed actions are performed would be desirable, but it seems hard to write a general spec of such constraints that applies to a wide range of code. For example, a client might like to be guaranteed that all actions, including failed actions, are done in the order in which the client requests them. Or, the client might like the same kind of ordering guarantee, but covering all clients rather than each individual one separately.

Here is the first spec, which allows `failed` responses at any time. It is modeled on the spec for sequential transactions in handouts 7 and 19.

```
MODULE Replication [
    V,                                          % Value
    S WITH { s0: () -> S }                      % State
    ] EXPORT Do =

TYPE VS          =  [v, s]
    A            =  S -> VS                     % Action

VAR  s           := S.s0()                      % State of service
    pending      :  SET A := {}                 % Failed actions to be done.
```

```
APROC Do(a) -> V RAISES {failed} = <<                % Do a or raise failed
        VAR vs := a(s) | s := vs.s; RET vs.v
    []  pending \/ := {a}; RAISE failed >>

THREAD DoPending() =                                % Do or drop a pending failed a
    DO << VAR a :IN pending |
        pending - := {a};
        BEGIN s := a(s).s [] SKIP END >>            % Do a or drop it
    [] SKIP OD

END Replication
```

Here is the second spec. Intuitively, we would like a failed response only if the service fails (by a crash or a network failure) sometime during the execution of the action, or if the action is requested while the system is recovering from a failure. The body of `Do` is a single atomic action which happens between the invocation and the return; if `down` is true during that interval, one possible outcome of the body is to raise `failed`. In the spec above, `Do` is an `APROC`; that is, there is a single atomic action in the module's interface. In the second spec below, `Do` is not an `APROC` but a `PROC`; that is, there are two atomic actions in the interface, one to invoke `Do` and one for its return.

Note that an action that has made it into `pending` can be executed at an arbitrary later time, perhaps when `down = false`.

MODULE Replication2 `[ V, S` as in `Replication ] EXPORT Do =`

```
TYPE VS          =  [v, s]
    A            =  S -> VS                         % Action

VAR  s           := S.s0()                          % State of service
    pending      :  SET A := {}                     % failed actions to be done.
    down         := false                           % true when system has failed
                                                    % and not finished recovering

PROC Do(a) -> V RAISES {failed} = <<                % Do a or raise failed
% Raise failed only if the system is down sometime during the execution. Note that this isn't an APROC
        VAR vs := a(s) | s := vs.s; RET vs.v
    []  down => pending \/ := {a}; RAISE failed >>

% Thread DoPending as in Replication

THREAD Fail() = DO << down := true >>; << down := false >> OD
% Happens whenever a node crashes or the network fails.

END Replication2
```

There are two general ways of coding a replicated service: primary copy (also known as master-slave, or primary-backup), and voting (also known as quorum consensus). Here we sketch the basic ideas of each.

Primary copy

The primary copy algorithm we give here is based on one invented by Liskov and Oki.[5] It codes a replicated state machine along the lines described in handout 18, using the Paxos consensus

[5] B. Liskov and B. Oki, Viewstamped replication: A new primary copy method to support highly available distributed systems, *Proc. 7th ACM Conference on Principles of Distributed Computing*, Aug. 1988.

algorithm to decide the sequence of state machine actions. When things are working well, the clients send action requests to the replica that is currently the primary; that replica uses Paxos to reach consensus among all the replicas about the index to assign to the requested action, and then responds to the client. We only assign an index `j` to an action if all prior indices have been assigned to actions, and no later ones.

For simplicity, we assume that every action is unique, and use the action to identify all the messages and outcomes associated with it. In practice, clients accomplish this by tagging each action with a unique ID and use the ID for this purpose.

MODULE PrimaryCopy `[`   % implements `Replication`
    `V, S` as in `Replication`
    `C,`   % Client names
    `R ] EXPORT Do =`   % Replica (server) names

```
TYPE VS          =  [v, s]
    A            =  S -> VS                         % Action
    X            =  ENUM[fail]                      % eXception result
    Data         =  (Null + V + X)                  % Data in message
    P            =  (R + C)                         % All process names
    M            =  [sp: P, rp: P, a, data]         % Message: sender, rcvr, action, data
    J            =  Nat                             % Action index: 1, 2, ...
```

There is a separate instance of consensus for each action index `J`. Its outcome records the agreed-upon `j`th action. We achieve this by making the `Consensus` module of handout 18 into a `CLASS` with `A` as `V`. The `actions` function maps from `J` to instances of the class. The processes in `R` run consensus. In a real system the primary would also be both the leader and an agent of the consensus algorithm, and its state would normally include the outcomes of all the already decided actions (or at least the recent ones) as well as the next available action index. This means that all the old outcomes will be available, so that `Outcome()` will never return `nil` for one of them. We assume this in what follows, and accordingly make `outcome` a function.

CLASS ReplCons `EXPORT allow, outcome =`

```
VAR  outcom   : (A + Null) := nil

APROC allow(a) = << outcome = nil => outcom := a [] SKIP >>
FUNC  outcome() -> (A + Null) = << RET outcom >>

END ReplCons
```

We abstract the communication as a set of messages in transit among all the clients and replicas. This could be coded by a set of the unreliable channels of handout 21, one in each direction for each client-replica pair; this is the way most real systems do it. Note that the channel can lose or duplicate both requests and responses. The channel connects the client's `Do` procedure with the replica. The `Do` procedure, which is the client side of the channel, deals with losses by retransmitting. If there's a failure, the result value may be lost; in this case `Do` raises `failed` as required by the `Replication` spec.

The client code keeps trying to get a replica to handle its request. The replica proceeds if it thinks it is the primary. If there's more than one primary, there will be contention for action indexes, so this is not desirable; hence if a replica loses the contention, it stops being primary. Since we are using Paxos, there should be only one primary at a time; in fact, the primary and the Paxos leader

should be the same. If a replica thinks it should become primary (presumably if the current primary seems to have failed and this replica is preferred for some reason), it does so.

For simplicity, we show each replica handling only one request at a time; in practice, of course, they could be batched. In spite of this, there can be lots of requests in progress at a time, since several replicas may be handling client request simultaneously if there is confusion about who is the primary.

We begin with code in which the replicas only keep track of the actions, that is, the results of consensus. This is not very practical, since it means that they have to recompute the current state from scratch for every request, but it is simple; we did the same thing when introducing transactions in handout 7. Later we consider the complications of keeping track of the current state.

```
VAR  actions     :  J -> ReplCons := InitActions()
     msgs        :  SEQ M  := {}                          % multiset of messages in transit
     mayBePrimary:  R -> Bool
     working     :  P -> (A + Null) := {}                 % history, for abstraction function

% ABSTRACTION FUNCTION:
     Replication.s = AllActions(LastJ())(S.s0()).s
     Replication.pending =   working.rng \/ {m :IN msgs | m.data = nil || m.a}
                           - Outcome.rng - {nil}

% INVARIANT: (ALL j :IN 1 .. LastJ() | Outcome(j) # nil)

% The client
VAR  primary : R                                          % remember the primary; guess initially
     done    : (Null + Data + X)                          % nil if still trying
PROC Do(a, c) -> V RAISES {failed} =                      % Do a from client c
     done := nil;
     working(c) := a;                                     % Just for the abstraction function
     DO done = nil => Send(c, primary, a, nil);           % Try the current primary
       << VAR r, a', data | (r, a', data) := Receive(c);
              IF   a' = a => done := data
              [*]  SKIP FI                                % Discard responses that aren't to a
       [] SKIP                                            % if timeout on response; send again
       [] VAR r | r # primary => primary := r             % if no response; guess another primary
       [] done := fail                                    % if too many retries
       >>
     OD;
     working(c) := nil;                                   % Just for the abstraction function
     IF done = fail => RAISE failed [*] RET done FI

% The server replicas
THREAD DoActions(r) =                                     % one for each replica
     DO maybePrimary(r) => VAR c, a, data |               % if I think I'm primary
          << (c,a,data):=Receive(r); working(r):=a >>;    % Receive request
          data := DoAction(r, a); Send(r,c,a, data);      % Do it and send response
          working(r) := nil                               % Just for the abstraction function
     OD

PROC DoAction(r, a) -> Data =
  DO VAR j |                                              % Keep trying until id is done.
     j := LastJ();                                        % Find last completed j
     IF a IN Outcome.rng => RET fail                      % Has a been done already? If so, failed
     [*]  j + := 1; actions(j).allow(a);                  % No. Try for consensus on a=action j
          Outcome(j) # nil =>                             % Wait for consensus
               IF   Outcome(j) = a => RET Value(j)        % If we got j, Return its result.
```



```
               [*]  maybePrimary(r) := false; RET fail   % Another action got j. I'm not primary
               FI
     FI
  OD

THREAD BecomePrimary(r) =                                 % one for each replica
     DO IShouldBePrimary(r) => maybePrimary(r) := true % IShouldBePrimary is up for grab
     [] SKIP OD


% These routines compute useful functions of the action history.

FUNC Value(j) -> V = RET AllActions(j)(S.s0()).v
% Compute value returned by j'th action; needs all outcomes <= j

FUNC AllActions(j) -> A = RET Compose((1 .. j) * Outcome)
% The composition of all the actions through j. Type error if any of them is nil.

FUNC Compose(aq: SEQ A) -> A =                            % discard intermediate Vs
     RET aq.head * (* : {a :IN aq.tail || (\ vs | a(vs.s))})

FUNC LastJ() -> J = RET {j | Outcome(j) # nil}.max [*] RET 0
% Last j for which consensus has been reached.

FUNC Outcome(j) -> (A + Null) = RET actions(j).outcome()

PROC InitActions() -> (J -> ReplCons) =                   % Make a ReplCons for each j
     VAR acts: J -> ReplCons := {}, rc: ReplCons |
          DO VAR j | ~ acts!j => acts(j) := rc.new OD; RET acts

% Here is the standard unreliable channel.
APROC Send(p1, p2, id, data) = << msgs := msgs \/ {M{p1, p2, id, data}} >>
APROC Receive(p) -> (P, ID, Data) = << VAR m :IN msgs |      % Receive msg for p
     m.rp = p => msgs - := {m}; RET (m.sp, m.id, m.data) >>
THREAD LoseOrDup() =
     DO << VAR m :IN msgs | BEGIN msgs - :={m} [] msgs \/ :={m} END >> [] SKIP OD

END PrimaryCopy
```

There is no explicit code for crashes. A crash simply resets the control state. For the client, this has the same practical effect as getting a failed response: you don't know whether the action happened or not. For the replica, either it got consensus or it didn't. If it did, the action has happened; if not, it hasn't. Either way, client will keep trying if the replica hasn't already sent a response that isn't lost in the channel. The client may see a failed response or it may get the result value.

Instead of failing if the action has already been done, we could try to return the proper result. It's unreasonably expensive to guarantee to always do this, but it's quite practical to do it for recent requests. This changes one line of DoAction:

```
     IF a IN Outcome.rng =>
          BEGIN RET fail [] VAR j | Outcome(j) = a => RET Value(j) END
```

This code is completely non-deterministic about retransmissions. As usual, it's necessary to be prudent in practice, especially since talking to too many replicas may cause needless failed responses. We have omitted any details about how the client finds the current primary; in practice, if the client talks to a replica that isn't the primary, that replica can redirect the client to the current primary. Of course, this redirection might happen several times if the system is unstable.

In this code replicas keep actions forever, both so that they can reconstruct the state and so that they can detect duplicate requests. When replicas keep the current state they don't need all the actions for that, but they still need them to detect duplicates. The reliable messages of handout 26 can't help with this, because they work only when a sender is talking to a single receiver, and here there are many receivers, one for each replica. . Real systems usually don't keep actions forever. Instead, they time them out, and often they tie each action to the current choice of primary, so that the action gets a failed response if the primary changes during its execution. To reconstruct the state of a very old replica, they copy the entire state from some other replica and then apply the most recent actions to bring it fully up to date.

The code above doesn't keep track of either the current state or the current action, but reconstructs them explicitly from the sequence of actions, using `LastJ` and `AllActions`. In a real system, the primary maintains both its idea of the last action index `j` and a corresponding state `s`. These satisfy the obvious invariant. In addition, the primary's `j` is the latest one, except while the primary is getting consensus, which it can't do atomically:

```
VAR jr          :  R -> J := {* -> 0}
    sr          :  R -> S := {* -> S.s0()}

INVARIANT (ALL r | sr(r) = AllActions(jr(r))(S.s0()).s)
INVARIANT jr(primary) = LastJ() \/ primary is getting consensus
```

This means that once the primary has obtained consensus on the action for the next `j`, it can update its state and return the corresponding result. If it doesn't obtain this consensus, then it isn't a legitimate primary. It needs to find out whether it should still be primary, and if so, bring its state up to date. The `CatchUp` procedure does the latter; we omit the code that chooses the primary. In practice we don't keep the entire action history, but catch up a severely outdated replica by copying the state from a current one; there are many variations on how to do this, and we omit this code as well.

```
PROC DoAction(id, a) -> Data =
  DO VAR j := jr(r) |                                  % Don't need to search for j.
    IF << a IN Outcome.rng => RET failed               % Has a been done already? If so, failed
    [*] j + := 1; actions(j).allow(a);                 % No. Try for consensus on a=action j
        Outcome(j) # nil =>                            % Wait for consensus
            IF Outcome(j)=a => VAR vs := a(sr(r)) |    % If we got j, return its result.
                << sr(r) := vs.s; jr(r) := j >>; RET vs.v
            [*] CatchUp(r) FI                          % Another action got j. Try again.
    FI
  OD
```

```
PROC Catchup(r) =                                      % Apply actions until you run out
    DO VAR j := jr(r) + 1, o := Outcome(j) |
        o = nil => RET;
        sr(r) := (o AS a)(sr(r)).s; jr(r) := j
    OD
```

Note that the primary is still running consensus for each action. This is necessary so that another replica can take over should the primary fail. It can, however, use the optimization for a sequence of consensus actions that is described in handout 18; this means that each consensus takes only one round-trip.

When they are running normally, the other replicas will run `Catchup` in the background, based on the information they get from the consensus. If a replica gets out of touch with the consensus, it can run the full `Catchup` to get back up to date.

We have assumed that a replica can do each action atomically. In general this will require the replica to use a transaction. The logging needed for the transaction can also provide the storage needed for the consensus outcomes.

A further optimization is for the primary to obtain a lease, so that no other replica can become primary until the lease expires. As we saw in handout 18, this means that it can respond to read-only requests from its copy of the state, without needing to run consensus. Furthermore, the other replicas can be simple read-write memories rather than active agents; in particular, they can be disk drives. Of course, if the primary fails we have to find another computer to act as primary.

Voting

The voting algorithm sketched here is based on one invented by Dave Gifford.[6] The idea is that each replica has some version of the state. Versions are indexed by `J` just as in `PrimaryCopy`, and each `Do` produces a new version. To read, you read the state of some copy of the latest version. To write, you find a copy of the current (latest) version, apply the action to create a new version, and write the new version into enough replicas; thus a write always does a read as well. A distributed transaction makes this operation atomic. A real system does the updates in place, applying the action to enough replicas of the current version; it may have to bring some replicas up to date first.

Warning: Because `Voting` is built on distributed transactions, it isn't easy to compare it to `PrimaryCopy`, which is only built on the basic `Consensus` primitive.

The definition of 'enough' must ensure that both reads and writes find the latest version. The standard way to do this is to insist that both examine a majority of the replicas, where 'majority' is defined so that any two majorities intersect. Here majority is renamed 'quorum' to emphasize the fact that it may not be a numerical majority, and we allow for separate read and write quorums, since we only need to assure that any read or write sees any previous write, not necessarily any previous read. This distinction allows us to bias the code to make reads easier at the expense of writes, or vice versa. For example, we could make every replica a read quorum; then the only write quorum is all the replicas. This choice makes it easy to do a read, since you only need to reach one replica. On the other hand, writes are expensive, and in fact impossible if even one replica is down.

There are many other ways to arrange the quorums. One simple scheme is to arrange the processes in a rectangle, make each row a read quorum, and make each row-column pair a write quorum (so that every write quorum has a replica in common with every read or write quorum.. For a square with n replicas, a read quorum has $n^{1/2}$ replicas and a write quorum $2\ n^{1/2} - 1$. By changing the shape of the rectangle you can favor reads or writes. If there are lots of replicas, these quorums are much smaller than a majority.

[6] D. Gifford, Weighted voting for replicated data. *ACM Operating Systems Review* **13**, 5 (Oct. 1979), pp 150-162.

1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	16

Note that the failure of an entire quorum makes the system unavailable. So the price paid for small quorums is that a small number of failures makes the system fail.

We abstract away from the details of communication and atomicity. The algorithm assumes that all the replicas can be updated atomically by a write, and that a replica can be read atomically. These atomic operations can be coded by the distributed transactions of handout 27. The consensus that is necessary for replication is hiding in the two-phase commit.

The abstract state is the state of a current replica. The invariant says that every `rq` has a current version, there's a `wq` in which every version is current, and two replicas with the same version also have the same state.

```
MODULE Voting [ as in Replication, R ] EXPORT Do =  % Replica (server) names

TYPE QS          =  SET SET R                       % Quorum Sets
     RWQ         =  [r: QS, w: QS]
     J           =  Int                             % Version number: 1, 2, ...

VAR  sr          :  R -> S := (* -> S.s0())         % States of replicas
     jr          :  R -> J := (* -> 0)              % Version Numbers of replicas
     rwq         := Quorums()                       % Read QuorumS

% ABSTRACTION FUNCTION: replication.s = sr({r | jr(r) = jr.rng.max}.choose)

% INVARIANT:    (ALL rq :IN rwq.r | jr.restrict(rq).rng.max = jr.rng.max)
             /\ (EXISTS wq :IN rwq.w | jr.restrict(wq).rng = (jr.rng.max}
             /\ (ALL r1, r2 | jr(r1) = jr(r2) ==> sr(r1) = sr(r2))

APROC Do(a) -> V = <<
     IF   ReadOnly(a) =>                            % Read, not update
          VAR rq :IN rwq.r,
               j := jr.restrict(rq).rng.max, r | jr(r) = j =>
               RET a(sr(r)).v
     []   VAR wq :IN rwq.w,                         % Update action
               j := jr.restrict(wq).rng.max, r | jr(r) = j =>
               j := j + 1;                          % new version number
               VAR vs := a(sr(r)), s := vs.s |
                    DO VAR r' :IN wq | jr(r') < j =>sr(r') := s; jr(r') := j OD;
                    RET vs.v
     FI >>

FUNC ReadOnly(a) -> Bool = RET (ALL s | a(s).s = s)

APROC Quorums () -> RWQ = <<
% Chooses sets of read and write quorums such that every write quorum intersects every read or write quorum.
     VAR rqs: QS, wqs: QS | (ALL wq :IN wqs, q :IN rqs \/ wqs | q/\wq # {}) =>
          RET RWQ{rqs, wqs} >>

END Voting
```


Note that because the read and write quorums intersect, every read sees all the preceding writes. In addition, any two write quorums must intersect, to ensure that writes are done with increasing version numbers and that a write sees the state changes made by all preceding writes. When the quorums are simple majorities, every quorum is both a read and a write quorum, so this complication is taken care of automatically. In the square scheme, however, although a read quorum can be a single row, a write quorum *cannot* be a single column, even though that would intersect with every row. Instead, a write quorum must be a row plus a column.

It's possible to reconfigure the quorums during operation, provided that at least one of the new write quorums is made completely current.

```
APROC NewQuorums() = <<
     VAR new := Quorums(), j:= jr.rng.max, s:=sr({r | jr(r)=jr.rng.max}.choose) |
          VAR wq :IN new.w | DO VAR r :IN wq | jr(r) < j => sr(r) := s OD;
          rwq := new
```

Eventually consistent replication

Some services have availability and response time constraints that make it impossible to maintain sequential consistency, the illusion that there is a single copy. Instead, each operation is initially processed at one replica, and the replicas "gossip" in the background to keep each other up to date about the updates that have been performed. Such strategies are used in name services[7] like DNS, for distributing information such as password files, and for updating system binaries. We sketched a spec for this in the section on coherence in handout 12 on naming; we repeat it here in a form that parallels our other specs. Another name for this kind of replication is 'loose consistency'.

Propagating updates in the background means that when an action is processed, the replica processing it might not know about some earlier actions. `LooseRepl` reflects this by allowing *any* subsequence of the earlier actions to determine the response to an action. Such behavior is possible (though unlikely) in distributed naming systems such as Grapevine[8] or the domain name service[9]. The spec limits the nondeterminism by requiring a response to include the effects of all actions executed before the most recent `Sync`. If `Sync`'s are done reasonably frequently, the incoherence won't get out of hand. A paper by Lampson[10] goes into much more detail.

For this to make sense as the system evolves, the actions must be defined on every state, and the result must be independent of the order in which the actions are applied (that is, they must all commute). In addition, it's simpler if the actions are idempotent (for the same reason that idempotency simplifies transaction redo recovery), and we assume that as well. Thus

```
(ALL aq: SEQ A, aa: SET A | aq.rng = aa ==> Compose(aq) = Compose(aa.seq))
```

You can always get idempotency by tagging each action with a unique ID, as we saw with transactions. To make the standard `read` and `write` operations on path names described in handout 12 commutative and idempotent, tag each name in the path name with a version number or timestamp, both in the actions and in the state.

[7] also called 'directories' in networks, and not to be confused with file system directories

[8] A. Birrell at al., Grapevine: An exercise in distributed computing. *Comm. ACM* **25**, 4 (Apr. 1982), pp 260-274.

[9] RFC 1034/5. You can find these at http://www.rfc-editor.org/isi.html. If you search the database for them, you will see information about updates.

[10] B. Lampson, Designing a global name service, *Proc. 4th ACM Symposium on Principles of Distributed Computing*, Minaki, Ontario, 1986, pp 1-10. You can find this at http://research.microsoft.com/lampson.

We write the spec in two equivalent ways. The first is in the style of handout 7 on disks and file systems and handout 12 on naming; it keeps track of all the possible states that the service can get into. It defines `Sync` to pick *some* state later than the latest one at the start of the `Sync`. It would be simpler to define `Sync` as `ss := {s}` and get rid of `ssNew`, as we did in handout 7, but this is too strong for the code we have in mind. Furthermore, the extra strength doesn't help the clients. `DropFromSS` doesn't change the behavior of the spec, since it only drops states that might not be used anyway, but it does make it easier to write the abstraction function.

MODULE LooseRepl `[ V, S WITH {s0: ()->S] EXPORT Do, Sync =`

```
TYPE VS          =  [v, s]
     A           =  S -> VS                          % Action

VAR  s           :  S     := S.s0()                  % latest state
     ss          :  SET S := {S.s0()}                % all States since end of last Sync
     ssNew       :  SET S := {S.s0()}                % all States since start of Sync

APROC Do(a) -> V = <<
     s := a(s).s; ss := Extend(ss, a); ssNew := Extend(ssNew, a);
     VAR s0 :IN ss | RET a(s0).v >>                  % choose a state for result

PROC Sync() = ssNew := {s}; << VAR s0 :IN ssNew | ss := {s0} >>; ssNew := {}

THREAD DropFromSS() =
     DO << VAR s1 :IN ss, s2 :IN ssNew | ss - := {s1}; ssNew - := {s2} >>
     [] SKIP OD

FUNC Extend(ss: SET S, a) -> SET S = RET ss \/ {s' :IN ss || a(s').s}

END LooseRepl
```

The second spec is closer to the code. It remembers the state at the last `Sync` instead of the current state, and keeps track explicitly of the actions done since the last `Sync`. After a `Sync` all the actions that happened before the `Sync` started are included in `s`, together with some subset of later ones.

MODULE LooseRepl2 `[ V, SA WITH {s0: ()->SA] EXPORT Do, Sync =`

```
TYPE S           = SA WITH {"+":=Apply}
     VS, A as in LooseRepl
VAR  s           :  S     := S.s0()                  % synced State (not latest)
     aa          :  SET A := {}                      % All Actions since last sync
     aaOld       :  SET A := {}                      % All Actions between last two Syncs

APROC Do(a) -> V = <<
     VAR aa0 : SET A | aa0 <= aa \/ aaOld =>         % choose actions for result
          aa \/ := {a}; RET a((s + aa0)).v >>

PROC Sync() =
     << aaOld := aa; aa := {} >>; << s := s + aaOld; aaOld := {} >>

THREAD DropFromAA() =
     DO << VAR a :IN aa \/ aaOld | s := s + {a}; aa - := {a}; aaOld - := {aa} >>
     [] SKIP
     OD

FUNC Apply(s0, aa0: SET A) -> S = RET PrimaryCopy.Compose(aa0.seq)(s).s

END LooseRepl2
```

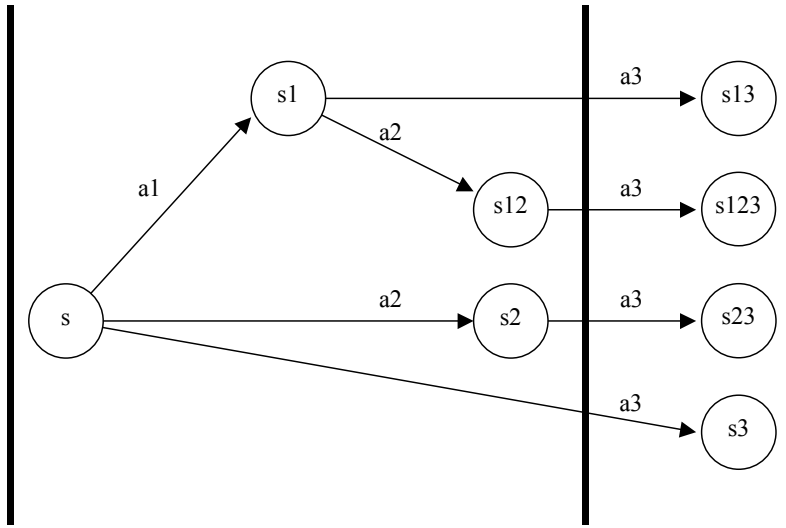

The picture shows how the set of possible states evolves as three actions are added. It assumes no actions are added while `Sync` 6 was running, so that the only state at the end of `Sync` 6 is `s`.

The abstraction function from `LooseRepl2` to `LooseRepl` constructs the states from the `Sync`ed state and the actions:

```
ABSTRACTION FUNCTION
     LooseRepl.s     = s + aa
     LooseRepl.ss    = {aa1: SET A | aa1 <= aa || s + aa1}
     LooseRepl.ssNew = {aa1: SET A | aa1 <= aa || s + (aa1 \/ aaOld)}
```

We leave the abstraction function from `LooseRepl` to `LooseRepl2` as an exercise.

The standard code has a set of replicas, each with a current state and a set of actions accumulated since the start of the last `Sync`; note that this is different from either spec. Typically actions have the form "set the value of name `n` to `v`". Any replica can execute a `Do` action. During normal operation the replicas send actions to each other with `Gossip`; more detailed code would send `a` (or a set of `a`'s) from `r1` to `r2` in a message. `Sync` collects all the recent actions and distributes them to every replica. We omit the complications of catching up a replica that has missed some `Sync`s and of keeping track of the set of replicas.



`Sync` 6 `Sync` 7

MODULE LRImpl `[ as in Replication, R ] EXPORT Do, Sync =` % implements `LooseRepl2`

```
TYPE VS          =  [v, s]
     A           =  S -> VS                          % Action
     J           =  NAT                              % Sync index: 1, 2, ...

VAR  jr          :  R -> J := {* -> 0}               % latest Sync here
     sr          :  R -> S := {* -> S.s0()}          % current State here
     hsrOld      :  R -> S := {* -> S.so()}          % history: state at last Sync
     hsOld       :  S := S.so()                      % history: state at last Sync
```

% Replica (server) names

```
    aar          :  R -> SET A := {* -> {}}          % actions since last Sync here
ABSTRACTION FUNCTION

APROC Do(a) -> V = << VAR r, vs := a(sr(r)) |
    aar(r) \/ := {a}; sr(r) := vs.s; RET vs.v >>

THREAD Gossip(r1, r2) =
    DO VAR a :IN aar(r1) - aar(r2) | aar(r2) \/ := a; sr(r2) := a(sr(r2))
    [] SKIP OD

PROC Sync() =
  VAR aa0: SET A := {},
      done: R -> Bool := {* -> false},
      j | j > jr.rng.max =>
    DO VAR r | jr(r) < j =>                          % first pass: collect all actions
         << jr(r) := j; aa0 \/ := aar(r); aar(r) := {} >> OD;
    DO VAR r | ~ done (r) =>                         % second pass: distribute all actions
         << sr(r) := sr(r) \/ aa0; done (r) := true >> OD


END LRImpl
```


29. Paper: Fault-Tolerance

The paper by Jim Gray and Andreas Reuter is a chapter from their magisterial book *Transaction Processing: Principles and Techniques*, Morgan Kaufmann, 1993, which should be on the shelf of every computer systems engineer. Read it as an adjunct to the lectures on consensus, availability, and replication.

Because of copyright, this paper is not available online.

.

30. Concurrent Caching

In the previous handout we studied the fault-tolerance aspects of replication. In this handout we study many of the performance and concurrency aspects, under the label 'caching'. A cache is of course a form of replication. It is usually a copy of some 'ground truth' that is maintained in some other way, although 'all-cache' systems are also possible. Normally a cache is not big enough to hold the entire state (or it's not cost-effective to pay for the bandwidth needed to get all the state into it), so one issue is how much state to keep in the cache. The other main issue is how to keep the cache up-to-date, so that a read will obtain the result of the most recent write as the `Memory` spec requires. We concentrate on this problem.

This handout presents several specs and codes for caches in concurrent systems. We begin with a spec for `CoherentMemory`, the kind of memory we would really like to have; it is just a function from addresses to data values. We also specify the `IncoherentMemory` that has fast code, but is not very nice to use. Then we show how to change `IncoherentMemory` so that it codes `CoherentMemory` with as little communication as possible. We describe various strategies, including invalidation-based and update-based strategies, and strategies using incoherent memory plus locking.

Since the various strategies used in practice have a lot in common, we unify the presentation using successive refinements. We start with cache code `GlobalImpl` that clearly works, but is not practical to code directly because it is extremely non-local. Then we refine `GlobalImpl` in stages to obtain (abstract versions of) practical code.

First we show how to use reader/writer locks to get a practical version of `GlobalImpl` called a coherent cache. We do this in two stages, an ideal cache `CurrentCaches` and a concrete cache `ExclusiveLocks`. The caches change the guards on internal actions of `IncoherentMemory` as well as on the external read and write actions, so they can't be coded externally, simply by adding a test before each read or write of `IncoherentMemory`, but require changes to its insides.

There is another way to use locks to get a different practical version of `GlobalImpl`, called `ExternalLocks`. The advantage of `ExternalLocks` is that the locking is decoupled from the internal actions of the memory system so that it can be coded separately, and hence `ExternalLocks` can run entirely in software on top of a memory system that only implements `IncoherentMemory`. In other words, `ExternalLocks` is a practical way to program coherent memory on a machine whose hardware provides only incoherent memory.

There are many practical codes for the methods that are described abstractly here. Most of them originated in the hardware of shared-memory multiprocessors.[1] It is also possible to code shared memory in software, relying on some combination of page faults from the virtual memory and checks supplied by the compiler. This is called 'distributed shared memory' or DSM.[2] Interme-

[1] J. Hennessy and D. Patterson, *Computer Architecture: A Quantitative Approach*, 2nd ed., Morgan Kaufmann, 1996, chapter 8, pp 635-754.
[2] K. Li and P. Hudak, Memory coherence in shared virtual memory systems. *ACM Transactions on Computer Systems* **7**, 4 (Nov. 1989), pp 321-359. For recent work in this active field see any ISCA, ASPLOS, OSDI, or SOSP proceedings.

diate schemes do some of the work in hardware and some in software.[3] Many of the techniques have been re-invented for coherent distributed file systems.[4]

All our code makes use of a global memory that is modeled as a function from addresses to data values; in other words, the spec for the global memory is simply `CoherentMemory`. This means that actual code may have a recursive structure, in which the top-level code for `CoherentMemory` using one of our algorithms contains a global memory that is coded with another algorithm and contains another global memory, etc. This recursion terminates only when we lose interest in another level of virtualization. For example,

a processor's memory may consist of a first level cache plus
a global memory made up of a second level cache plus
a global memory made up of a main memory plus
a global memory made up of a local swapping disk plus
a global memory made up of a file server ....

Specs

First we recall the spec for ordinary coherent memory. Then we give the spec for efficient but ugly incoherent memory. Finally, we discuss an alternative, less intuitive way of writing these specs.

Coherent memory

The first spec is for the memory that we really want, which ensures that all memory operations appear atomic. It is essentially the same as the `Memory` spec from Handout 5 on memory specs, except that `m` is defined to be total. In the literature, this is sometimes called a 'linearizable' memory; in the more general setting of transactions it is 'serializable' (see handout 20).

```
MODULE CoherentMemory [P, A, V] EXPORT Read, Write =
% Arguments are Processors, Addresses and Data

TYPE M          =  A -> D SUCHTHAT (ALL a | m!a))
VAR  m

APROC Read(p, a) -> D = << RET m(a) >>
APROC Write(p, a, d) = << m(a) := d >>

END CoherentMemory
```

From this point we drop the `a` argument and study a memory with just one location; that is, we study a cached register. Since everything about the specs and code holds independently for each address, we don't lose anything by doing this, and it reduces clutter. We also write the `p` argument as a subscript, again to make the specs easier to read. The previous spec becomes

```
MODULE CoherentMemory [P, V] EXPORT Read, Write =
% Arguments are Processors and Data
```

[3] David Chaiken and Anant Agarwal. Software-extended coherent shared memory: performance and cost. *Proceedings of the 21st Annual Symposium on Computer Architecture*, pages 314-324, April 1994 (http://www.cag.lcs.mit.edu/alewife/papers/soft-ext-isca94.html). Jeffrey Kuskin et al., The Stanford FLASH multiprocessor. In *Proceedings of the 21st International Symposium on Computer Architecture*, pages 302-313, Chicago, IL, April 1994 (http://www-flash.stanford.edu/architecture/papers/ISCA94).

[4] M. Nelson et al., Caching in the Sprite network file system. *ACM Transactions on Computer Systems* **11**, 2 (Feb. 1993), pp 228-239. For recent work in this active field see any OSDI or SOSP proceedings.


```
TYPE M          =  D                                    % Memory
VAR  m

APROC Read_p -> D = << RET m >>
APROC Write_p(d) = << m := d >>

END CoherentMemory
```

Of course, code usually has limits on the size of a cache, or other resource limitations that can only be expressed by considering all the addresses at once, but we will not study this kind of detail here.

Incoherent memory

The next spec describes the minimum guarantees made by hardware: there is a private cache for each processor, and internal actions that move data back and forth between caches and the main memory, and between different caches. The only guarantee is that data written to a cache is not overwritten in that cache by anyone else's data. However, there is no ordering on writes from the cache to main memory.

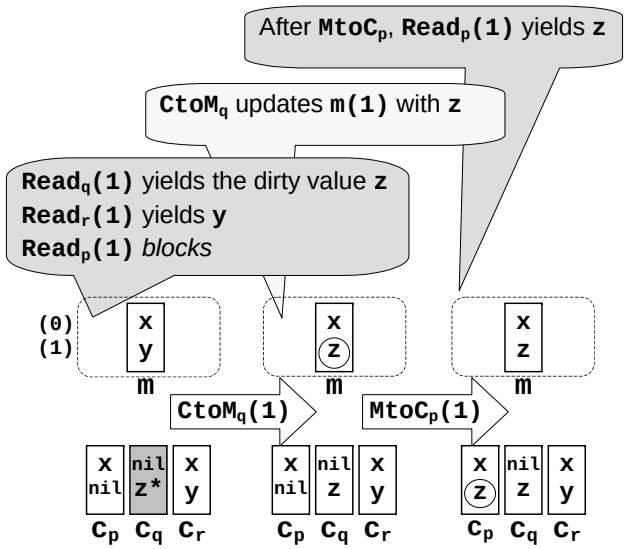
This is not enough to get any useful work done, since it allows writes to remain invisible to others forever. We therefore add a `Barrier` synchronization operation that forces the cache and memory to agree. This can be used after a `Write` to ensure that an update has been written back to main memory, and before a `Read` to ensure that the data being read is current. `Barrier` was called `Sync` when we studied disks and file systems in handout 7, and eventual consistency in handouts 12 and 28.

Note that `Read` has a guard `Live` that it makes no attempt to satisfy (hardware usually has an explicit flag called `valid`). Instead, there is another action `MtoC` that makes `Live` true. In a real system an attempt to do a `Read` will trigger a `MtoC` so that the `Read` can go ahead, but in Spec we can omit the direct linkage between the two actions and let the non-determinism do the work. We use this coding trick repeatedly in this handout. Another example is `Barrier`, which forces the cache to drop its data by waiting until `Drop` happens; if the cache is dirty, `Drop` will wait for `CtoM` to store its data into memory first.

You might think that this is just specsmanship and that a nondeterministic `MtoC` is silly, but in fact transferring data from `m` to `c` without a `Read` is called prefetching, and many codes do it under various conditions: because it's in the next block, or because a past reference sequence used it, or because the program executes a prefetch instruction. Saying that it can happen nondeterministically captures all of this behavior very simply.

We adopt the convention that an invalid cache entry has the value `nil`.

```
MODULE IncoherentMemory [P, A, V] EXPORT Read, Write, Barrier =

TYPE M          =  D                                    % Memory
     C          =  P -> (D + Null)                      % Cache

VAR  m          :  CoherentMemory.M                     % main memory
     c          := C{* -> nil}                          % local caches
     dirty      :  P -> Bool := {*->false}              % dirty flags

% INVARIANT Inv1: (ALL p | c!p)                         % each processor has a cache
% INVARIANT Inv2: (ALL p | dirty_p ==> Live_p)          % dirty data is in the cache
```



IncoherentMemory with processes p, q, r

```
APROC Read_p -> D = << Live_p => RET c_p >>                  % MtoC gets data into cache
APROC Write_p(d)  = << c_p := d; dirty_p := true >>

APROC Barrier_p = << ~ Live_p => SKIP >>                     % wait until not in cache

FUNC Live_p -> Bool = RET (c_p # nil)

% Internal actions

THREAD Internal_p = DO MtoC_p [] CtoM_p [] VAR p' | CtoC_p,p', [] Drop_p [] SKIP OD

APROC MtoC_p = << ~ dirty_p => c_p := m >>                   % copy memory to cache
APROC CtoM_p = << dirty_p => m := c_p; dirty_p := false >>   % copy cache to memory
APROC CtoC_p,p' = << ~ dirty_p' /\ Live_p => c_p' := c_p >>  % copy from cache p to p'
APROC Drop_p = << ~ dirty_p => c_p := nil >>                 % drop clean data from cache

END IncoherentMemory
```

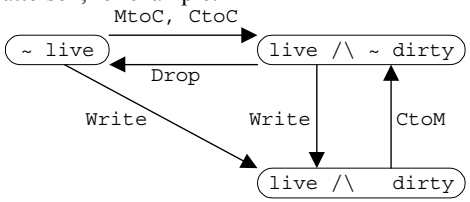
In real code some of these actions may be combined. For example, if the cache is dirty, a real barrier operation may do `CtoM; Barrier; MtoC` by just storing the data. These combinations don't introduce any new behavior, however, and it's simplest to study the minimum set of actions presented here.

This memory is 'incoherent': different caches can have different data for the same address, so that adjacent reads by different processors may see completely different data. Thus, it does not implement the `CoherentMemory` spec given earlier. However, after a $\texttt{Barrier}_p$, c_p is guaranteed

to agree with `m` until the next time `m` changes or `p` does a `Write`.[5] There are commercial machines whose memory systems have essentially this spec.[6] Others have explored similar specs.[7]

Here is a simple example that shows the contents of two addresses `0` and `1` in `m` and in three processors `p`, `q`, and `r`. A dirty value is marked with a `*`, and circles mark values that have changed. Initially $\texttt{Read}_q\texttt{(1)}$ yields the dirty value `z`, $\texttt{Read}_r\texttt{(1)}$ yields `y`, and $\texttt{Read}_p\texttt{(1)}$ blocks because $c_p\texttt{(1)}$ is `nil`. After the $\texttt{CtoM}_q$ the global location `m(1)` has been updated with `z`. After the $\texttt{MtoC}_p$, $\texttt{Read}_p\texttt{(1)}$ yields `z`. One way to ensure that the $\texttt{CtoM}_q$ and $\texttt{MtoC}_p$ actions happen before the $\texttt{Read}_p\texttt{(1)}$ is to do $\texttt{Barrier}_q$ followed by $\texttt{Barrier}_p$ between the $\texttt{Write}_q\texttt{(1)}$ that makes `z` dirty in c_q and the $\texttt{Read}_p\texttt{(1)}$.

Here are the possible transitions of `IncoherentMemory` for a given address. This kind of state transition picture is the standard way to describe cache algorithms in the literature; see pages 664-5 of Hennessy and Patterson, for example.



This is the weakest shared-memory spec that seems likely to be useful in practice. But perhaps it is too weak. Why do we introduce this messy incoherent memory? Wouldn't we be much better off with the simple and familiar coherent memory? There are two reasons to prefer `IncoherentMemory`:

- Code for `IncoherentMemory` can run faster—there is more locality and less communication. As we will see later in `ExternalLocks`, software can batch the communication that is needed to make a coherent memory out of `IncoherentMemory`.
- Even `CoherentMemory` is tricky to use when there are concurrent clients. Experience has shown that it's necessary to have wizards to package it so that ordinary programmers can use it safely. This packaging takes the form of rules for writing concurrent programs and procedures that encapsulate references to shared memory. We studied these rules in handout 14 on practical concurrency, under the name 'easy concurrency'. The two most common examples are:

  Mutual exclusion / critical sections / monitors together with a "lock before touching' rule, which ensure that a number of references to shared memory can be done without interference from other processors, just as in a sequential program. Reader/writer locks are an important variation.

[5] An alternative version of `Barrier` has the guard $\texttt{~ live}_p \texttt{ \/ } (c_p = \texttt{m})$; this is equivalent to the current $\texttt{Barrier}_p$ followed by an optional $\texttt{MtoC}_p$. You might think that it's better because it avoids a copy from `m` to c_p in case they already agree. But this is a spec, not an implementation, and the change doesn't affect its external behavior.

[6] Digital Equipment Corporation, *Alpha Architecture Handbook*, 1992. IBM, *The PowerPC Architecture*, Morgan Kaufmann, 1994.

[7] Gharachorloo, K., et al., Memory consistency and event ordering in scalable shared-memory multiprocessors, *Proc. 17th Symposium on Computer Architecture*, 1990, pp 15-26. Gibbons, P. and Merritt, M., Specifying non-blocking shared memories, *Proc. 4th ACM Symposium on Parallel Algorithms and Architectures*, 1992, pp 158-168.

Producer-consumer buffers.

For the ordinary programmer only the simplicity of the package is important, not the subtlety of its code. We need a smarter wizard to package `IncoherentMemory`, but the result is as simple to use as the packaged `CoherentMemory`.

Specifying legal histories directly

It's common in the literature to write the specs `CoherentMemory` and `IncoherentMemory` explicitly in terms of legal sequences of references in each processor, rather than as state machines (see the references in the previous section). We digress briefly to explain this approach informally; it is similar to what we did to specify concurrent transactions in handout 20.

For $\texttt{CoherentMemory}^{LH}$, there must be a total ordering of *all* the $\texttt{Read}_p$ and $\texttt{Write}_p\texttt{(v)}$ actions done by the processors (for all the addresses) that

- respects the order at each `p`, and
- such that for each `Read` and closest preceding `Write(v)`, the `Read` returns `v`.

For $\texttt{IncoherentMemory}^{LH}$, *for each address separately* there must be a total ordering of the $\texttt{Read}_p$, $\texttt{Write}_p$, and $\texttt{Barrier}_p$ actions done by the processors that has the same properties. `IncoherentMemory` is weaker than `CoherentMemory` because it allows references to different addresses to be ordered differently. If there were only one address and no other communication (so that you couldn't see the relative ordering of the operations), you couldn't tell the difference between the two specs. A real barrier operation usually does a `Barrier` for every address, and thus forces all the references before it at a given processor to precede all the references after it.

It's not hard to show that $\texttt{CoherentMemory}^{LH}$ is equivalent to `CoherentMemory`. It's less obvious that $\texttt{IncoherentMemory}^{LH}$ is almost equivalent to `IncoherentMemory`. There's more to this spec than meets the eye, because it doesn't say anything about how the chosen ordering is related to the real times at which different processors do their operations. Actually it is somewhat more permissive than `IncoherentMemory`. For example, it allows the following history

- Initially `x=1`, `y=1`.
- Processor `p` reads `4` from `x`, then writes `8` to `y`.
- Processor `q` reads `8` from `y`, then writes `4` to `x`.

For `x` we have the ordering $\texttt{Write}_q\texttt{(4)}$; $\texttt{Read}_p$, and for `y` the ordering $\texttt{Write}_p\texttt{(8)}$; $\texttt{Read}_q$.

We can rule out this kind of predicting the future by observing that the processors make their references in some total order in real time, and requiring that a suitable ordering exist for the references in each prefix of this real time order. With this restriction, the two versions of $\texttt{IncoherentMemory}^{LH}$ and `IncoherentMemory` are equivalent. But the restriction may not be an improvement, since it's conceivable that a processor might be able to predict the future in this way by speculative execution. In any case, the memory spec for the Alpha is in fact $\texttt{IncoherentMemory}^{LH}$ and allows this freedom.

Coding coherent memory

We give a sequence of refinements that implement `CoherentMemory` and are successively more practical: `GlobalImpl`, `Current Caches`, and `ExclusiveLocks`. Then we give a different kind of code that is based on `IncoherentMemory`.

Global code

Now we give code for `CoherentMemory`. We obtain it simply by strengthening the guards on the operations of `IncoherentMemory` (omitting `Barrier`, which we don't need). This code is not practical, however, because the guards involve checking global state, not just the state of a single processor. This module, like later ones, maintains the invariant `Inv3` that an address is dirty in at most one cache; this is necessary for the abstraction function to make sense. Note that the definition of `Current` says that the cache agrees with the abstract memory.

We show only the code that differs from `IncoherentMemory`, boxing the new parts.

```
MODULE GlobalImpl [P, A, V] EXPORT Read, Write =   % implements CoherentMemory

TYPE ...                                           % as in IncoherentMemory
VAR  ...

% ABSTRACTION: CoherentMemory.m = (Clean() => m [*] {p | dirty_p || c_p}.choose)

% INVARIANT Inv3: {p | dirty_p}.size <= 1          % dirty in at most one cache

APROC Read_p -> D = << Current_p => RET c_p >>     % read only current data
APROC Write_p(d) =                                 % Write maintains Inv3
    << Clean() \/ dirty_p => c_p := d; dirty_p := true >>

FUNC Current_p =                                   % p's cache is current?
    RET c_p = (Clean() => m [*] {p | dirty_p || c_p}.choose)
FUNC Clean() = RET (ALL p | ~ dirty_p)             % all caches are clean?

% Same internal actions as IncoherentMemory.

END GlobalImpl
```

Notice that the guard on `Read` checks that the data in the processor's cache is current, that is, equals the value currently stored in the abstract memory. This requires finding the most recent value, which is either in the main memory (if no processor has a dirty value) or in some processor's cache (if a processor has a dirty value). The guard on `Write` ensures that a given address is dirty in at most one cache. These guards make it obvious that `GlobalImpl` implements `CoherentMemory`, but both require checking global state, so they are impractical to code directly.

Code in which caches are always current

We can't code the guards of `GlobalImpl` directly. In this section, we refine `GlobalImpl` a bit, replacing some (but not all) of the global tests. We carry this refinement further in the following sections. Our strategy for correctness is to always strengthen the guards in the actions, without changing the rest of the code. This makes it obvious that we simulate the previous module and that existing invariants hold. The only thing to check is that new invariants hold.

The main idea of `CurrentCaches` is to always keep the data in the caches current, so that we no longer need the `Current` guard on `Read`. In order to achieve this, we impose a guard on a write that allows it to happen only if no other processor has a cached copy. This is usually coded by having a write invalidate other cached copies before writing; in our code `Write` waits for `Drop` actions at all the other caches that are live. Note that `Only` implies the guard of `GlobalImpl.Write` because of `Inv2` and `Inv3`, and `Live` implies the guard of `GlobalImpl.Read` because of `Inv4`. This makes it obvious that `CurrentCaches` implements `GlobalImpl`. `CurrentCaches` uses the non-local functions `Clean` and `Only`, but it eliminates `Current`. This is

progress, because Read, the most common action, now has a local guard, and because Clean and Only just test Live and dirty, which is much simpler than Current's comparison of c_p with m.

As usual, the parts not shown are the same as in the last module, GlobalImpl.

```
MODULE CurrentCaches ... =                          % implements GlobalImpl

TYPE ...                                            % as in IncoherentMemory
VAR  ...

% ABSTRACTION to GlobalImpl: Identity on m, c, and dirty.

% INVARIANT Inv4: (ALL p | Live_p ==> Current_p)    % data in caches is current

...

FUNC Only_p -> Bool = RET {p' | Live_p',} <= {p}    % appears at most in p's cache

APROC Read_p -> D = << Live_p => RET c_p >>          % read locally; OK by Inv4
APROC Write_p(d) =                                  % write locally the only copy
     << Only_p => c_p := d; dirty_p := true >>

...

APROC MtoC_p = << Clean() => c_p := m >>            guard maintains Inv4
...

END CurrentCaches
```

Code using exclusive locks

The next code refines CurrentCaches by introducing an exclusive (write) lock with a Free test and Acquire and Release actions. A writer must hold the lock on an object while it writes, but a reader need not hold any lock (Live acts as a read lock according to Inv4 and Inc6). Thus, multiple readers can read in parallel, but only one writer can write at a time, and only if there are no concurrent readers. This means that before a write can happen at p, all other processors must drop their copies; making this happen is called 'invalidation'. The code ensures that while a processor holds a lock, no other cache has a copy of the locked object. It uses the non-local functions Clean and Free, but everything else is local. Again, the guards are stronger than those in CurrentCaches, so it's obvious that ExclusiveLocks0 implements CurrentCaches. We show the changes from CurrentCaches.

```
MODULE ExclusiveLocks0 ... =                        % implements CurrentCaches

TYPE ...                                            % as in IncoherentMemory
VAR  ...
     lock        :  P -> Bool := {*->false}         % p has lock on cache?

% ABSTRACTION to CurrentCaches: Identity on m, c, and dirty.

% INVARIANT Inv5: {p | lock_p}.size <= 1            % lock is exclusive
% INVARIANT Inv6: (ALL p | lock_p ==> Only_p)      % locked data is only copy

...

APROC Write_p(d) =                                  % write with exclusive lock
     << lock_p => c_p := d; dirty_p := true >>
```



```
...

FUNC Free() -> Bool = RET (ALL p | ~ lock_p)        % no one has cache locked?

THREAD Internal_p =
     DO    MtoC_p [] CtoM_p [] VAR p' | CtoC_p,p' [] Drop_p
        [] Acquire_p [] Release_p  [] SKIP OD

APROC MtoC_p =                                      % guard maintains Inv4, Inv6
     << Clean() /\  (lock_p \/ Free()) => c_p := m >>
APROC CtoC_p,p' =                                   % guard maintains Inv6
     << Free()  /\ ~ dirty_p, /\ Live_p => c_p' := c_p >>

APROC Acquire_p = << Free() /\ Only_p => lock_p :=true >> % exclusive lock is on cache
APROC Release_p = << lock_p := false >>             % release at any time

...

END ExclusiveLocks0
```

Note that this all works even in the presence of cache-to-cache copying of dirty data; a cache can be dirty without being locked. A strategy that allows such copying is called *update-based*. The usual code broadcasts (on the bus) every write to a shared location. That is, it combines with each $Write_p$ a $CtoC_{p,\,p'}$ for each live p'. If this is done atomically, we don't need the $Only_p$ in $Acquire_p$ This is good if for each write of a shared location, the average number of reads on a different processor is near 1. It's bad if this average is much less than 1, since then each read that goes faster is paid for with many bus cycles wasted on updates.

It's possible to combine updates and invalidation. They you have to decide when to update and when to invalidate. It's possible to make this choice in a way that's within a factor of two of an optimal algorithm that knows the future pattern of references.[8] The rule is to keep updating until the accumulated cost of updates equals the cost of a read miss, and then invalidate.

Both Read and Write now do only local tests, which is good since they are supposed to be the most common actions. The remaining global tests are the Only test in Acquire, the Clean test in MtoC, and the Free tests in Acquire, MtoC, and CtoC. In hardware these are most commonly coded by snooping on a bus. A processor can broadcast on the bus to check that:

- No one else has a copy (Only).
- No one has a dirty copy (Clean).
- No one has a lock (Free).

It's called 'snooping' because these operations always go along with transfers between cache and memory (except for Acquire), so no extra bus cycles are need to give every processor on the bus a chance to see them.

For this to work, another processor that sees the test must either abandon its copy or lock, or signal false. The false signals are usually generated at exactly the same time by all the processors and combined by a simple 'or' operation. The processor can also request that the others relinquish their locks or copies; this is called 'invalidating'. Relinquishing a dirty copy means first writing it back to memory, whereas relinquishing a non-dirty copy means just dropping it from

[8] A. Karlin et al, Competitive snoopy caching. *Algorithmica* **3**, 1 (1988), pp 79-119.

the cache. Sometimes the same broadcast is used to invalidate the old copies and update the caches with new copies, although our code breaks this down into separate `Drop`, `Write`, and `CtoC` actions.

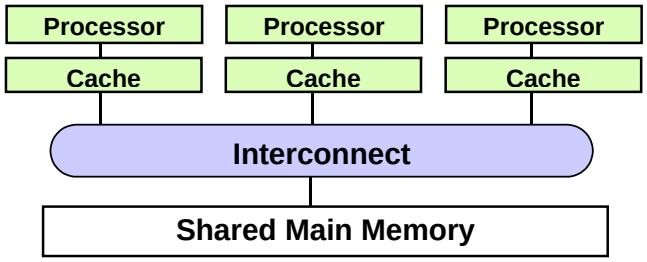
Keeping dirty data locked

In the next module, we eliminate the cache-to-cache copying of dirty data; that is, we eliminate updates on writes of shared locations. We modify `ExclusiveLocks` so that locks are held longer, until data is no longer dirty. Besides the delayed lock release, the only significant change is in the guard of `MtoC`. Now data can only be loaded into a cache `p` if it is not dirty in `p` and is not locked elsewhere; together, these facts imply that the data item is clean, so we no longer need the global `Clean` test.

```
MODULE ExclusiveLocks ... =                          % implements ExclusiveLocks0

TYPE ...                                             % as in ExclusiveLocks0
VAR  ...

% ABSTRACTION to ExclusiveLocks0: Identity on m, c, dirty, and lock.

% INVARIANT Inv7: (ALL p | dirty_p ==> lock_p)       % dirty data is locked

...

APROC MtoC_p =                                       % guard implies Clean()
     << ~ dirty_p /\ (lock_p \/ Free()) => c_p := m >>
APROC Release_p = << ~ dirty_p => lock_p := false >>  % don't release if dirty
...

END ExclusiveLocks
```

For completeness, we give all the code for `ExclusiveLocks`, since there have been so many incremental changes. The non-local operations are boxed.

```
MODULE ExclusiveLocks[P,A,V] EXPORT Read,Write =     % implements CoherentMemory

TYPE M = D                                           % Memory
     C = P -> (D + Null)                             % Cache

VAR  m           :  CoherentMemory.M                 % main memory
     c           := C{* -> nil}                      % local caches
     dirty       :  P -> Bool := {*->false}          % dirty flags
     lock        :  P -> Bool := {*->false}          % p has lock on cache?

% ABSTRACTION to ExclusiveLocks: Identity on m, c, dirty, and lock.

% INVARIANT Inv1: (ALL p | c!p)                      % every processor has a cache
% INVARIANT Inv2: (ALL p | dirty_p ==> Live_p)       % dirty data is in the cache

% INVARIANT Inv3: {p | dirty_p}.size <= 1            % dirty in at most one cache
% INVARIANT Inv4: (ALL p | Live_p ==> Current_p)     % data in caches is current
% INVARIANT Inv5: {p | lock_p}.size <= 1             % lock is exclusive
% INVARIANT Inv6: (ALL p | lock_p ==> Only_p)       % locked data is only copy
% INVARIANT Inv7: (ALL p | dirty_p ==> lock_p)      % dirty data is locked

APROC Read_p -> D = << Live_p => RET c_p >>           % read locally; OK by Inv4
APROC Write_p(d) =                                   % write with exclusive lock
     << lock_p => c_p := d; dirty_p := true >>
```

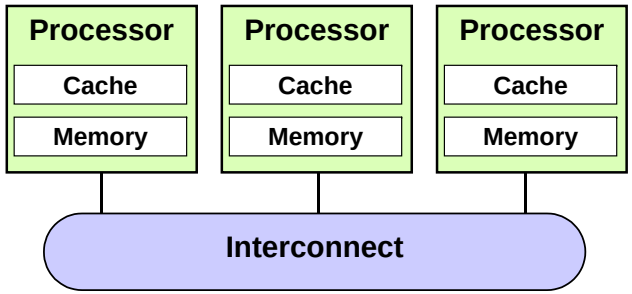
Uniform memory access: Processors with local caches and uniform access to shared memory

```
FUNC Live_p -> Bool = RET (c_p # nil)
FUNC Only_p -> Bool = RET {p' | Live_p',} <= {p}          % appears at most in p's cache?
FUNC Free() -> Bool = RET (ALL p | ~ lock_p)              % no one has cache locked?

THREAD Internal_p =
     DO    MtoC_p [] CtoM_p [] VAR p' | CtoC_p,p' [] Drop_p
        [] Acquire_p [] Release_p [] SKIP OD

APROC MtoC_p =                                            % guard implies Clean()
     << ~ dirty_p /\ (lock_p \/ Free()) => c_p := m >>
APROC CtoM_p = << dirty_p => m := c_p; dirty_p := false >> % copy cache to memory.
APROC CtoC_p,p' =                                         % guard maintains Inv6
     << Free() /\ ~ dirty_p' /\ Live_p => c_p' := c_p >>
APROC Drop_p = << ~ dirty_p => c_p := nil >>               % drop clean data from cache

APROC Acquire_p = << Free() /\ Only_p => lock_p :=true >> % exclusive lock is on cache
APROC Release_p = << ~ dirty_p => lock_p := false >>      % don't release if dirty

END ExclusiveLocks
```

Practical code

The remaining global tests are the `Only` test in the guard of `Acquire`, and the `Free` tests in the guards of `Acquire`, `MtoC` and `CtoC`. There are many ways to code them. Here are a few:

- Snooping on the bus, as described above. This is only practical when you have a cheap synchronous broadcast, that is, in a bus-based shared memory multiprocessor. The shared bus limits the maximum performance, so typically such systems are not built with more than about 8 processors. As processors get faster, a shared bus gets less practical.
- Directory-based: Keep a "directory", usually associated with main memory, containing information about where locks and copies are currently located. To check `Free`, a processor need only interact with the directory. To check `Only`, the same strategy can be used; however, there is a difficulty if cache-to-cache copying is permitted—the directory must be informed when such copying occurs. For this reason, directory-based code usually eliminates cache-to-cache copying entirely. So far, there's no need for broadcast. To acquire a lock, the directory may need to communicate with other caches to get them to relinquish locks and copies. This can be done by broadcast, but usually the directory keeps track of all the live processors and sends a message to each one. If there are lots of processors, it may fall back to broadcast for locations that are shared by too many processors.



Non-uniform memory access (NUMA): Memory attached to processors

These schemes, both snooping and directory, are based on a model in which all the processors have uniform access to the shared memory.

The directory technique extends to large-scale multiprocessor systems like Flash and Alewife, distributed shared memory, and locks in clusters[9], in which the memory is attached to processors. When the abstraction is memory rather than files, these systems are often called 'non-uniform memory access', or NUMA, systems.

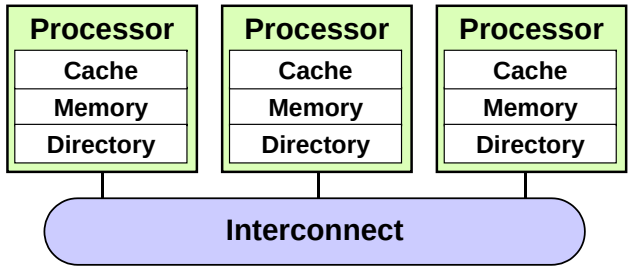
The directory itself can be distributed by defining a 'home' location for each address that stores the directory information for that address. This is inefficient if that address turns out to be referenced mainly by other processors. To make the directory's distribution adapt better to usage, store the directory information for an address in a 'master' processor for that address, rather than in the home processor. The master can change to track the usage, but the home processor always remembers the master. Thus:

```
FUNC Home(a) -> P = …                                  % some fixed algorithm
VAR  master: P -> A -> P                               % master(p) is partial
     copies: P -> A -> SET P                           % defined only at the master
     locker: P -> A -> P                               % defined only at the master
INVARIANT (ALL a, p, p' |
            master(Home(a))!a                          % master is defined at a's home P,
         /\ master(p)!a /\ master(p')!a ==>            % where it's defined, it's the same
              master(p)(a) = master(p')(a)
         /\ copies!p = (p = master(Home(a))(a)) )      % and copies is defined only at mast
```

The `Home` function is often a hash of `a`; it's possible to change the hash function, but if this is not atomic it must be done very carefully, because `Home` will be different at different processors and the invariants must hold for all the different `Home`'s.

- Hierarchical: Partition the processors into sets, and maintain a directory for each set. The main directory attached to main memory keeps track of which processor sets have copies or locks; the directory for each set keeps track of which processors in the set have copies or locks. The hierarchy may have more levels, with the processor sets further subdivided, as in Flash.

[9] Kronenberg, N. et al, The VAXCluster concept: An overview of a distributed system, *Digital Technical Journal* **1**, 3 (1987), pp 7-21.



NUMA with distributed directory

There are many issues for high-performance code: communication cost, bandwidth into the cache into tag store, interleaving, and deadlock. The references at the start of this handout go into a lot of detail.

Purely software code is also possible. This form of DSM makes `V` be a whole virtual memory page and uses page faults to catch memory operations that require software intervention, while allowing those that can be satisfied locally to run at full speed. A live page is mapped, read-only unless it is dirty; a page that isn't live isn't mapped.[10]

Code based on `IncoherentMemory`

Next we consider a different kind of code for `CoherentMemory` that runs on top of `IncoherentMemory`. This code guarantees coherence by using an external read/write locking discipline. This is an example of an important general strategy—using weaker memory together with a programming discipline to guarantee strong coherence.

The code uses read/write locks, as defined earlier in the course, one per data item. There is a module $\texttt{ExternalLocks}_p$ for each processor `p`, which receives external `Read` and `Write` requests, obtains the needed locks, and invokes low-level `Read`, `Write`, and `Barrier` operations on the underlying `IncoherentMemory` memory. The composition of these pieces implements `CoherentMemory`. We give the code for $\texttt{ExternalLocks}_p$.

```
MODULE ExternalLocks_p [A, V] EXPORT Read, Write =     % implements CoherentMemory

% ReadAcquire_p acquires a read lock for processor p.
% Similarly for ReadRelease, WriteAcquire, WriteRelease

PROC Read_p =
    ReadAcquire_p; Barrier_p; VAR d| d := IncoherentMemory.Read_p; ReadRelease_p; RET

PROC Write_p(d) = WriteAcquire_p; IncoherentMemory.Write_p(d); Barrier_p; WriteRelease

END ExternalLocks_p
```

This code does not satisfy all the invariants of `CurrentCaches` and its code. In particular, the data in caches is not always current, as stated in `Inv4`. It is only guaranteed to be current if it is read-locked, or if it is write-locked and dirty.

[10] K. Li and P. Hudak, Memory coherence in shared virtual memory systems, *ACM Transactions on Computer Systems* **7**, 4 (Nov 1989), pp 321-359.

Invariants Inv1, Inv2, and Inv3 are still satisfied. Invariants Inv5 and Inv6 no longer apply because the lock discipline is completely different; in particular, a locked copy need not be the only copy of an item. Let wLockPs be the set of processors that have a write-lock, and rLockPs be those with a read-lock.

We thus have Inv1-3, and new Inv4a-Inv7a that replace Inv4-Inv7:

```
% INVARIANT Inv4a:                                          % Data is current
    (ALL p | dirty_p \/ (p IN rLockPs /\ Live_p) ==> Current_p())

% INVARIANT Inv5a:                                          % Write lock is exclusive.
     wLockPs.size <= 1

% INVARIANT Inv6a:                                          % Write lock excludes read locks.
     wLockPs # {} ==> rLockPs = {}

% INVARIANT Inv7a: (ALL p | dirty_p ==> p IN wLockPs)       % dirty data is write-locked
```

With these invariants, the identity abstraction to GlobalImpl works:

```
% ABSTRACTION to GlobalImpl: Identity on m, c, and dirty.
```

We note some differences between ExternalLocks and ExclusiveLocks, which also uses exclusive locks for writing:

- In ExclusiveLocks, Read can always proceed if there is a cache copy. In ExternalLocks, Read has a stronger guard in ReadAcquire (requiring a read lock).
- In ExclusiveLocks, MtoC checks that no other processor has a lock on the item. In ExternalLocks, an MtoC can occur as long as it doesn't overwrite dirty writes.
- In ExternalLocks, the guard for Acquire only involves lock conflicts, and does not check Only. (In fact, ExternalLocks doesn't use Only at all.)
- Additional Barrier actions are required in ExternalLocks.
- In ExclusiveLocks, the data in the cache is always current. In ExternalLocks, it is only guaranteed to be current for read-lock holders, and for write-lock holders who have already written.

In practice we don't surround every read and write with Acquire and Release. Instead, we take advantage of the rules for easy concurrency and rely on the fact that any reference to a shared variable must be in a critical section, surrounded by Acquire and Release of the lock that protects it. All we need to add is a Barrier at the beginning of the critical section, after the Acquire, and another at the end, before the Release. Sometimes people build these barrier actions into the acquire and release actions; this is called 'release consistency'.

Note—here we give up the efficiency of continuing to hold the lock until someone else needs it.

Remarks

Costs of incoherent memory

IncoherentMemory allows a multiprocessor shared memory to respond to Read and Write actions without any interprocessor communication. Furthermore, these actions only require communication between a processor and the global memory when a processor reads from an address

that isn't in its cache. The expensive operation in this spec is Barrier, since the sequence $Write_p$; $Barrier_p$; $Barrier_q$; $Read_q$ requires the value written by p to be communicated to q. In most code Barrier is even more expensive because it acts on all addresses at once. This means that roughly speaking there must be at least enough communication to record globally every address that p wrote before the $Barrier_p$, and to drop from p's cache every address that is globally recorded as dirty.

Read-modify-write operations

Although this isn't strictly necessary, all current codes have additional external actions that make it easier to program mutual exclusion. These usually take the form of some kind of atomic read-modify-write operation, for example an atomic swap or compare-and-swap of a register value and a memory value. A currently popular scheme is two actions: ReadLinked(a) and WriteConditional(a), with the property that if any other processor writes to a between a $ReadLinked_p$(a) and the next $WriteConditional_p$(a), the WriteConditional leaves the memory unchanged and returns an indication of failure. The effect is that if the WriteConditional succeeds, the entire sequence is an atomic read-modify-write from the viewpoint of another processor, and if it fails the sequence is a SKIP. Compare-and-swap is obviously a special case; it's useful to know this because something as strong as compare-and-swap is needed to program wait-free synchronization using a shared memory. Of course these operations also incur communication costs, at least if the address a is shared.

We have shown that a program that touches shared memory only inside a critical section cannot distinguish memory that satisfies IncoherentMemory from memory that satisfies the serial spec CoherentMemory. This is not the only way to use IncoherentMemory, however. It is possible to program other standard idioms, such as producer-consumer buffers, without relying on mutual exclusion. We leave these programs as an exercise for the reader.

Caching as easy concurrency

We developed the coherent caching code by evolving from the obviously correct GlobalImpl to code that has no global operations except to acquire locks. Another way to look at it is that coherent caching is just a variation on easy concurrency. Each Read or Write touches a shared variable and therefore must be done with a lock held, but there are no bigger atomic operations. The read lock is Live and the write lock is lock. In order to avoid the overhead of acquiring and releasing a lock on every memory operation, we use the optimization of holding onto a lock until some other cache needs it.

Write buffering

Hardware caches, especially the 'level 1' caches closest to the processor, usually come in two parts, called the cache and the write buffer. The latter holds dirty data temporarily before it's written back to the memory (or the level 2 cache in most modern systems). It is small and optimized for high write bandwidth, and for combining writes to the same cache block that happen close together in time into a single write of the entire block.

Invalidation

All caching systems have some provision for invalidating cache entries. A system that implements CoherentMemory usually must invalidate a cache entry that is written on another processor. The invalidation must happen before any read that follows the write touches the entry. Many

systems, however, provide less coherence. For example, NFS simply times out cache entries; this implements `IncoherentMemory`, with the clumsy property that the only way to code `Barrier` is to wait for the timeout interval. The web does caching in client browsers and also in proxies, and it also does invalidation by timeout. A web page can set the timeout interval, though not all caches respect this setting. The Internet caches the result of DNS lookups (that is, the IP address of a DNS name) and of ARP lookups (that is, the LAN address of an IP address). These entries are timed out; a client can also discard an entry that doesn't seem to be working. The Internet also caches routing information, which is explicitly updated by periodic OSPF packets.

Think about what it would cost to make all these loosely coherent schemes coherent, and whether it would be worth it.

Locality and granularity

Caching works because the patterns of memory references exhibit locality. There are two kinds of locality.

- Temporal locality: if you reference an address, you are likely to reference it again in the near future, so it's worth keeping that item in the cache.
- Spatial locality: if you reference an address, you are likely to reference a neighboring address in the near future. This makes it worthwhile to transfer a large block of data to the cache, since the overhead of a miss is only paid once. Large blocks do have two drawbacks: they consume more bandwidth, and they introduce or increase 'false sharing'. A whole block has to be invalidated whenever any part of it is written, and if you are only reading a different part, the invalidation makes for extra work.

Both temporal and spatial locality can be improved by restructuring the program, and often this restructuring can be done automatically. For instance, it's possible to rearrange the basic blocks of a program based on traces of program execution so that blocks that normally follow each other in traces are in the same cache line or virtual memory page.

Distributed file systems

A distributed file system does caching which is logically identical to the caching that a memory system does. There are some practical differences:

- A DFS is usually built without any hardware support, whereas most DSM's depend at least on the virtual memory system to detect misses while letting hits run at full local memory speed, and perhaps on much more hardware support, as in Flash.
- A DFS must deal with failures, whereas a DSM usually crashes a program that is sharing memory with another program that fails.
- A DFS usually must scale better, to hundreds or thousands of nodes.
- A DFS has a wider choice of granularity: whole files, or a wide range of block sizes within files.

32. System Administration

The goal of system administration (admin for short) is to reduce the customer's total cost of owning a computer and keeping it working (TCO). Most of this cost is people's time rather than purchased hardware or software. The way to reduce TCO without giving up functionality is to make software that does more things automatically. What we want is plug-and-play, both hardware and software, both system and applications. We certainly don't have that now.

There are two parts to admin:

- Establishing policy: what budget, what priorities, how much availability, etc.
- Executing the policy: installation, repair, configuration, tuning, monitoring, etc.

The problem

The customer's problem is that too much time (and hence too much money) goes into futzing with computers: making them work rather than using them to do work. Companies and end-users have different viewpoints. Companies care about the cost of getting a certain amount of useful computing. End-users care about the amount of hassle. They can't measure the cost or the hassle very well, so perception is reality for the most part.

Companies

Corporate customers want to reduce total cost of ownership, defined as all the money spent on the computer that doesn't contribute directly to productive work. This includes hardware and software purchases, budgeted support people, and most important, time spent by end-users in 'futzing': installing, diagnosing, repairing, and learning to use the computer, as well as answering other users' questions.[1]

Most of TCO is support cost and end-user futzing with the computer, not purchase of hardware and software. At least two studies have tabulated costs of $8k/year, distributed as follows in one of them:[2]

Hardware	$2,000	24%
Software	$940	12%
Training	$1,400	17%
Management	**$3,830**	**47%**
Total	$8,170	

The management cost breaks down like this

End-user downtime	$1,350	35%
Desktop administrator/tools	$1,280	34%

[1] Perhaps it should also include time the user spends playing games, surfing the Web for fun, and writing love letters, but that's outside the scope of this handout.
[2] Forrester, quoted in Datamation, June 1 1996, p 10. There's a similar study by Gartner Group that reports about the same total cost/year. I've also seen $11k/year!

Coworker time	$540	14%
Disaster prevention/recovery	$660	17%

It's hard to take the three significant figures seriously, but the overall story is believable. Certainly it's consistent with my own experience and with some back-of-the-envelope calculations.

Another example of the dominating cost of admin is storage. It costs $100-$1000/year to keep one gigabyte of storage on a properly run server, with adequate backups, expansion and replacement of hardware, etc. To buy one gigabyte for your PC costs about $30. So the cost of purchase is negligible compared to the cost of admin.

End-users

End-users don't think in terms of dollars; they just want the system to work without any hassles—not such an unreasonable demand. Put another way, they want less futzing and more time for real work or play. Of course, they also want a fast system with lots of features.

Here are the problems users have that lead to futzing:

Install	I plugged it in and it didn't work.
Repair	It worked yesterday, but today I can't ...
Replicate	I can't get to a copy of ...
Reconcile	I have two copies, and I just want one.
Convert	I have a Word file, but my Word won't read it.
Learn	I don't know how to.... I did ... before, but how?
Find	I can't lay my hands on ...

The ones above the line can be addressed by better system administration. The ones below need better usability, help, and information management tools.

Architecture

Admin is what is left over after the algorithms programmed into all the system components have done their best. This description implies a modular structure: components doing their best, and admin controlling them.

Ideally, the components would adapt automatically to changes in their environment. Here are some examples of components that work that way. Most of them are network components.

Ethernet adapters, hubs, and bridges (switches).[3]

IP routers within a domain, using OSPF to set up their routing tables.[4]

The AN1 network (see handout 22).[5]

[3] R. Perlman, *Interconnections: Bridges and Routers* , Addison-Wesley, 1992, chapter 3.
[4] Perlman, chapters 8-10.
[5] Autonet: A high-speed, self-configuring local area network using point-to-point links, *IEEE Journal on Selected Areas in Communications* **9**, 8, (Oct. 1991), pp1318-1335.

The Petal disk server.[6]

A replicated storage server.[7]

Today admin is almost entirely manual, both for setting policy and for executing it. Setting policy has to be manual, since it's the way the system's owners express their intentions. We want to make execution automatic, so the only manual action is stating the policy, and software does everything else.

Policy has two parts:

- Registering users and components
- Allocating resources, by establishing quotas, setting priorities, or whaterver.

We won't say anything more about this.

What does execution do? It keeps the system in a good state. To make this idea useful we need to:

- Define the system state that is relevant for admin. This is a drastic abstraction of all the bytes on the disk and in memory.
- Describe the set of good states. This is partly defined by the user's policy, but mostly by the needs of the various components.
- Build software that gets the system into a good state and keeps it there.

Describing good states is better than giving procedures for changing the state, because you can easily check that the system is in a good state and report the ways that it isn't. Often you can also compute a cheap way to get from the current state to a good one, instead of redoing a whole installation. More generally you can analyze the description in many useful ways.

This section describes how to organize a system this way, how to define the state, and how to describe good states using predicates.

Organizing the system: Methodology

How should we organize a system in order to minimize its total cost of ownership without hurting its functionality and performance? I've divided this question into three main topics: modularity, large scale admin, and certification.

We have two kinds of modules, *components* that do real work, and *admin apps* that coax or bully the components into states that meet each other's needs and carry out the user's policy. Components let admin read and change the state and track performance. Admin builds a model of state and performance, tells components what to do, and tells the user about the model.

For the admin app itself to be used effectively on a large scale, it has to be able to handle and to summarize lots of similar components automatically. Large scale admin requires some way to

[6] E. Lee and C. Thekkath, Petal: Distributed virtual disks, *Proc. 7th ACM Conference on Architectural Support for Programming Languages and Operating Systems*, Oct. 1996, pp 84-92.
[7] B. Liskov and B. Oki, Viewstamped replication: A new primary copy method to support highly available distributed systems, *Proc. 7th ACM Conference on Principles of Distributed Computing*, Aug. 1988.

move lots of bits; this requires connectivity, but many combinations of CD-ROM and network connections can work.

It would be nice if the architecture were perfect and everyone implemented it perfectly. Since that won't happen, we need a way to certify components and sets of components that are easy to administer, so that customers can make rational choices.

Modularity

Today admin is almost entirely manual. It includes:

Allocating resources (disk space, IRQ's, etc.).

Organizing the file system name space and (usually) deciding where to put each file.

Establishing and implementing a security policy.

Replicating things (for backup, onto multiple machines, into multiple documents, etc.).

Installing and upgrading hardware, software, fonts, templates, etc..

Configuring components to fit the environment (network stuff, I/O devices, etc.).

Diagnosing and repairing broken hardware and software.

Finding things.

The tools that allow you to do these things also give you lots of opportunities to get it wrong. We want admin to be automatic. That means it needs to be modular, separate from the components; for this to work, the components have to provide some help.

If components took care of everything, we wouldn't need any separate admin (well, a little for setting policy). Certainly it's good for a component to be more self-managing, but it's not practical to solve the whole admin problem this way. We need the modularity of separate components and admin for several reasons:

- Developing components is hard enough; it mustn't be held back by admin development. What we want is the opposite: to be able to work on admin separately from working on components.
- Components have to be coordinated, and it's easier to do that from one place than to have them all talk to each other.
- Different kinds of admin are appropriate for different situations. You don't want to manage an aircraft carrier and a dentist's office in the same way.
- Automatic admin is fairly hard, which means it has to be done with common code, either services that components call or admin apps that call the components. Services are a lot harder to design.
- It's easier to change an admin app than to change a lot of components or to change services that many components call.
- No single vendor controls all of the components, and it takes a long time to get the whole industry to change its practices. This means that you have to work with legacy components.

How are responsibilities divided between admin apps and components?

An admin app builds a model of the state, updates it efficiently to reflect changes, understands the predicates that describe the good states, and tells the components to make state changes so that the predicates hold. It takes care of completing or undoing these changes after failures if necessary. It presents the state to the user and accepts instructions about policy. Finally, it observes and models performance and presents the results to the user.

Here are three examples of admin apps that together suggest the range of things we want to do:

A diagnostician that tells you what's wrong or strange in your system.

A software installer that keeps a set of apps correctly installed and up to date.

A hierarchical storage manager that automatically backs up and archives your files.

A component allows every bit of state to be named and provides operations to read the state (and track it efficiently, using change logs; more on this later) and to update the state. The Internet's Simple Network Management Protocol (SNMP) is an example of how to do this; it was designed to manage networking components, but can be applied more widely. Admin has to be able to make a set of changes reliably, even a large set that spans several components. This means that the update operations must have enough idempotence or testability that admin can implement redo or undo. The update operations must also be atomic enough to meet any requirements for the system to provide normal service concurrently. If these are strong, the components will have to do locking much like transaction processing resource managers. Usually, however, it's OK to do admin more or less off line.[8]

Modularity means interfaces, in this case the interfaces between components and admin. Like all interfaces, they need to be reasonably stable. They also need to be as uniform as possible; one of the reasons for the states-and-predicates architecture is to provide a standard framework for designing these interfaces. Currently, the state of the art in interfaces is rather primitive, well represented by the Internet's SNMP, which is just a version of the naming interface defined in handout 12.

Of course everything in computing is recursive, so one man's admin app is another man's component. Put another way, you can use this architecture within a single component, to put it together out of sub-components plus admin.

Large scale admin

Obviously large organizations want to do large scale admin of hundreds or thousands of machines and users. But that isn't the whole story. Vendor and third party support organizations also want to do this. What are the essentials for administering lots of systems?

- *Telepresence*[9]: You can do everything from one place over the network—no running around.

[8] If all the components implement general multi-component distributed transactions, that would do the job very nicely. Since it's unlikely that they will, it's a good thing that it's unnecessary. Admin can take on this responsibility with just a little help.

[9] I mean appropriate telepresence—we won't need videoconferencing any time soon.

- *Program access*: You can do everything from a program—nothing has only a GUI. That way you can do the same thing many times with the same human effort as doing it once. Of course there have to be parameters, to accommodate variations.
- *Leverage*: You can say what you want declaratively, and software figures out how to get it done in many common cases. You can say each thing once, and have it propagated to all the places where it applies. To keep a number of machines in the desired state you describe that state once with a predicate and install that predicate by name on each machine. Then the system maintains each machine in the desired state automatically. You don't give a procedure for getting into the desired state, which probably won't work for some of the machines.
- *Problem reporting*: Currently it's an incredible pain to report a problem. All the "what was the state" and "what happened recently" should be automatic. For this to work the components obviously have tell you what to report.

Certification

A customer should be able to assemble a system (hardware and/or software) from approved components, or assemble a collection of components that passes a "this system is approved" test, and be pretty confident that it will just work. For the components, this means writing specs and testing for conformance with them. For the system, which won't really have a spec, it at least means testing that things hang together and don't conflict.

Of course there shouldn't be anything compulsory about certification. People could also live wild and free as they do today, without any such assurance. But perhaps we could do a bit better by pointing to the uncertified components that are causing the problems.

Defining the state

The most important architectural idea after modularity is to pay careful attention to the system state. Here 'state' means state that is relevant to admin, stuff the users can change that is not their 'content'. Some examples: file names and properties, style definitions, sort order in mail folders, network bindings. 'State' doesn't mean the page table or the file allocation table; those are not part of the user model. Of course the line between admin state and content is not sharp, as some of the examples illustrate. But in general the admin state exists separately from the content.

Today the state is scattered around all over the place. A lot is in the registry in Windows, or in environment variables in Unix, though most of it is very application-specific, without common schemas or inheritance. But a lot is also in files, often encoded in a complicated way. There's no way to dump out all the state uniformly, say into a database.

Microsoft Word is an interesting example. It stores state in the registry, in several kinds of templates, in documents, in files (for instance, the custom dictionaries) and in the running application. It has dozens of different kinds of state: text, a couple of kinds of character properties, paragraph, section, and document properties, fields, styles, a large assortment of options, macros, autotext and autocorrect, windows on the screen, and many more. It has several different organizing mechanisms: styles, templates, document properties, "file locations". Some things can be grouped and named, others cannot. It's not that any of this is bad, but it's quite a jumble. There are lots of chances to get tangled up, and you don't get much help in getting untangled.

Other things that are much simpler are much worse. I won't try to catalog the confusing state of a typical mail client, or of network configuration. I don't know enough to even try to write down the latter, although I know a lot about computer networking.

Design principles

The basic principle is that both users and programs should be able to easily see and change all the state. Again, 'state' here means state that is relevant to admin. That includes resource consumption and unusual conditions.

Second, the state should be understandable. That means that similar things should look similar, and related things should be grouped together. The obvious way to achieve this is to mak all the state available through a database interface so you can use database tools to view and change it, and the database methodology of schemas to understand it. This is good both for people and for programs. It's especially good for customizing views of the state, since millions of people know how to do this in the context of database queries, views, reports, and forms. This has never been tried, and the state may be too messy to fit usefully into the relational mold.

This does not imply that the 'true' state is stored in a database; aside from compatibility, there are lots of reasons why that's not a good idea. It's pretty easy to convert from any other representation into a database representation. Reflecting database updates back into another representation is sometimes more difficult, but with careful design it's possible. It's not necessary to be able to change all of the database view. Some parts might be computed in such a way that changing them doesn't make sense; this is just what we have today in databases.

It's also a good idea to have an HTML interface, so that you can get to the state from a standard web browser. Perhaps the easiest way to get this is to convert to database form first and use an existing tool to make HTML from that.

Naming and data model

To have general admin, we need a uniform way to name all of the state and some standard ways to represent it externally. This is usually called a 'data model'. Of course anything can be coded into any data model, but we want something natural that is easy both to code and to use.[10]

A hierarchical name space or tree is the obvious candidate, already used in file systems, object properties, the Windows registry, and the Internet's SNMP. It's good because it's easy to extend it in a decentralized way, it's easy to map most existing data structures into it, and it's easy to merge two trees using a first-one-wins rule for each path name. The abstraction that goes along with trees is path names.

We studied hierarchical naming in handout 12. Recall that there are three basic primitives:

```
Lookup(d, pn) -> ((D, PN) + V)
Set(d, n, v)
Enum(d) -> SET N
```

With this `Lookup` you can shift responsibility for navigating the naming graph from the server to the client. An alternative to `Enum` is `Next(n) -> N`.

[10] We want to stay out of the Turing tarpit, where everything is possible and nothing is easy (Alan Perlis).

Directories can have different code for these primitives. This is essential for admin, since many entities are part of the name space, including file systems, DNS, adapters and i/o devices, and network routers.

Nodes may have types. For directory nodes, the type describes the possible children. In a database this information is called the 'schema'. In SNMP it is the 'management information block' or MIB.

The other obvious candidate for a data model is tables, already used in databases, forms, and spreadsheets. It's easy to treat a table as a special case of a tree, but the reverse is not true: many powerful operations on tables don't work on trees because there isn't enough structure. So it seems best to use a naming tree as the data model and subclass it with a table where appropriate.

Describing the good states: Predicates

The general way to describe a set of states too big to be enumerated is to write down a predicate (that is, a Boolean function) that is true exactly on those states. Since any part of the state can be named, the predicates can refer to any part of it, so they are completely general.

For this approach to be useful, we have to be able to state predicates formally so that the computer can process them. Some examples of interesting predicates (not stated formally):

> This subtree of the file system (for instance, the installed applications) is the same as that one over there (for instance, on the server).
>
> All the .dll's referenced from this .exe are present on the search path, in compatible versions.
>
> Tex is correctly installed.
>
> All the connected file servers are responding promptly.
>
> There is enough space for you to do the day's work.
>
> All the Word and Powerpoint files have been converted to the latest version.

We also have to be able to process the predicates. Here are three useful things to do with a predicate that describes the good or desired states of a system:

- *Diagnose*. Point out things about the current state that stop it from being good, that is, from satisfying the predicate. For static state this can be a static operation. Active components should be monitored continuously for good state and expected performance; in most systems there are things that hang up pretty often.
- *Install or repair*. Change the state so that it satisfies the predicate. It's best if the needed changes can be computed automatically from the predicate, as they usually can for replication predicates ("This should be the same as that"). Sometimes we may need rules for changing the state, of the form "To achieve X, make change Y". These are hard to write and maintain, so it's best to have as few of them as possible.
- *Analyze*. Detect inconsistency (for instance, demands for two different versions of the same .dll), weirdness, or consequences of a change (will the predicate still be true after I do this?).
- *Anticipate* failures. Run diagnostics and look at event logs to give advice or take corrective action when things are abnormal.

Stating predicates

Predicates that are used for admin must be structured so that admin operations can be done automatically. If a predicate is written as a C function that returns 0 or 1, there's not much we can do with it except evaluate it. But if it's written in a more stylized way we can do lots of processing. A simple example: if a predicate is the conjunction of several named parts (Hardware working, Word installed, Data files backed up recently, ...) with named sub-parts, we can give a more precise diagnosis ("Word installation is bad because winwrd.hlp is corrupted"), find conflicts between the parts, and so on.

The stylized forms of predicates need to be common across the system, so that:

> Users can learn a common language.
>
> Every component builder doesn't have to invent a language.
>
> The same predicates can work for multiple components. Example: "I don't want to lose more than 15 minutes of work." Applications, file systems, and clusters could all contribute to implementing this.

We need a common place to store the predicates, presumably the file system or the registry. This saves coding effort, makes it possible for multiple components to see them, and makes it easier to apply them to a larger environment.

We also need names for predicates, just as we need them for all state components and collections of values, so they can be shared, composed, and reported to the user in a meaningful way. Often giving a predicate a name is a way of reducing the size of the state. You say "Word is installed" rather than "winword.exe is in \msoffice\winword, hyph32.dll is in \msoffice\winword\, ...". Today many systems have weak approximations to named predicates: "profiles" or "templates". Unfortunately, they are badly documented and very clumsy to construct and review. They usually also lack any way to combine them, so as soon as you have two profiles it's a pain to change anything that they have in common.

What stylized forms for predicates are useful? Certainly we want to name predicates. We also want to combine them with 'and', 'or', 'not', 'all', and 'exists', as discussed earlier. The rest of this section discusses three other important forms. Predicates with parameters are essential for customizing. Predicates that describe replication and dependency are the standard cases for admin. They let you say "this should be like that except for ..." and "this needs that in order to work". Just evaluating them tells you in detail what's wrong: "this part of the state doesn't agree" and "this dependency isn't satisfied". Furthermore, it's usually obvious how to automatically make one of these predicates true. So you get both diagnosis and automatic repair.

Customizing: Predicates with parameters

Global names are good because they provide a meaningful vocabulary that lets you talk more abstractly about the system instead of reciting all the details every time. Put another way, they reduce the size and complexity of the state you have to deal with. Parameters, which are local names that you can rebind conveniently, are good because they make the abstractions much more general, so each one can cover many more cases: "All the files with extension *e* have been backed up", or "All the files that match filter *f*", instead of just "All the files". Subroutines with parameters have the same advantages that they do for programming.

Computed values are closely related to parameters, since computing on the parameters makes them much more useful. The most elaborate example of this that I know of is Microsoft Word templates, which let you compute the current set of macros, menus, etc. by merging several named templates. The next section on replication gives more examples.

Replication

This is the most important form of predicate: "Mine should be like hers, except for ...". In its simplest form it says "Subtree A (folder A and its contents) is the same as subtree B". Make this true by copying B over A (assuming B can't be changed), or by reconciling them in some other way. Disconnected operation, caching, backup, and software distribution are all forms of replication, as well as the obvious ones of disk mirroring and replicated storage servers.

Variations on replication predicates are:

> "A contains B, that is, every name in B has the same value in A, but A may have other names as well." To make this true, copy everything in B over A, leaving things that are only in A untouched.
>
> "A is ahead of B, that is, every name in B is also in A with a version at least as recent." To make this true, copy everything in B over a missing or earlier thing in A.

All of these are good because they can control a lot of state very clearly and concisely. For example, a department has a predicate "standard app installation" that says "Lotus Notes is installed, budget rollup templates are installed, ...", and "Lotus Notes is installed" says "C:\Lotus\Notes equals \\DeptServer\Notes, ...", etc.. Then just asserting "standard app installation" on a machine ensures that the machine acquires the standard installation and keeps it up to date.

In all these cases B might be some *view* of a folder tree rather than the actual tree, for instance, all the items tagged as being in the minimal installation, or all the *.doc files. (I suppose A might be a view too, though I can't think of a good example.) Or B might contain hashes of values or UID's rather than the values themselves; this is useful for a cheap check that A is in a good state, though it's obviously not enough for repairing problems.

Replication predicates are the basic ones you need for installing software; note that there are usually several A's: the app's directory, a directory for dll's, the registry. Replication predicates are also what you need for disconnected operation, for backup, and for many other admin operations.

See the later section on coding techniques for ways of using these predicates efficiently.

Dependency

This is the next most important form of predicate: "I need ... in order to work". Its general form is "If A, then there must be a B such that P is true." Dependency is the price of modularity: you can't just replicate the whole system, so you need to know how to deal with the boundaries. Programs have dependencies on fonts, .dll's, other programs, help files, etc. Documents also have dependencies, on apps that interpret them, templates, fonts, linked documents, etc.. Often unsatisfied dependencies are the most difficult problems to track down.

The hardest part about dependencies is finding out what they are. There are three ways to do this:

- Declare them manually (I need this version of `foo.dll`; I need COM interface `baz`; I need Acrobat 4.0 or later).
- Deduce them by static analysis (What are all the links from this document, what fonts does it use, what Corba components? What .dll's does this .exe use statically?).
- Execute and trace what gets used.

These approaches are complementary, not competitive. In particular, a static analysis or a trace can generate a declaration, or flag non-compliance with an existing declaration.

Given dependencies, we can do a lot of useful analysis.

> Are all the dependencies satisfied now?
>
> What will break if I make this change?
>
> Do the requirements of two apps conflict?
>
> How can this dependency be eliminated?
>
> When something goes wrong, what state needs to be captured in reporting the problem?

Components should provide operations for getting rid of dependencies, by doing format conversions, embedding external things like fonts and linked documents, etc. There is an obvious tradeoff between performance and robustness here; it should be under control of the admin policies, not of obscure check boxes in Save dialogs.

The key to fast startup (of the system or of an app) is pre-binding of facts about the environment. The problem is that this creates more dependencies: when the environment changes the prebindings become obsolete, and checking for this has to be fast. If these dependencies are public then the system can check any changes against them (probably by processing the change log with a background agent) and notify the app next time it runs. Otherwise the app can scan the change log when it starts up. A problem is that it's hard to register these dependencies reliably; it may have to be done by tracing the component's calls on other components.

Resource allocation

Resource allocation is an aspect of the state that is sometimes important for workstations and always important for shared servers. We need predicates that describe how to allocate storage, CPU cycles, RAM, bandwidth, etc. among competing apps, users, documents, or other clients. 'Committed' resources like disk storage or modems can't be taken away and given back later; they need special treatment. This is why HSM (hierarchical storage management) is good: it makes disk a revocable rather than a committed resource, as VM does for RAM.

A related issue is garbage collection. Abstractly, this follows from predicates: any state not needed to satisfy the predicates can be discarded. It's unclear whether we can make this idea practical.

Coding resource allocation in a system with more than one component that can do a given task requires load balancing. It also requires monitoring the performance, to detect violations or potential violations of the predicates. Often the predicate will be defined in terms of some model of the system which describes how it ought to respond to an offered load. Monitoring consists of collecting information, comparing it to the model, and taking some action when reality is out of

step. Ideally the action is some automatic correction, but it might just be a trouble report to an administrator.

One important form of monitoring is keeping track of failures. As we saw in handout 28, a fault-tolerant system tends to become fault-intolerant if failed component are not repaired promptly.

Coding techniques

The basic coding strategy is to start with the "real" state that a component runs against, and compute a view of the state that's easy to present and program. This way admin isn't tied down by legacy formats. The computed view is a kind of cache, with the same coherence issues. Of course the idea works in the other direction too: compute the stuff a component needs from stuff that's convenient for admin.

To make this efficient, if a component has a state that's expensive to read, it writes a change log. Entries in the log pinpoint the parts of the state that have changed. Admin reads this log plus a little of the state and tracks the changes to keep its view coherent with the real thing. The file system, for instance, logs the names or ids of files that are written or renamed. Note that this is much simpler and more compact than a database undo/redo log, which has to have all the information needed to undo or redo changes. Some components already generate this information in the form of notifications, but recording it permanently means that apps that are not running at the time can still track the state. Also, writing a log entry is usually much cheaper than calling another program.

Change logs with some extra entries are also good for monitoring performance, for diagnosis, and for load balancing.

There may be no change log, or it may be incomplete or corrupt. In this case a way to pinpoint the parts of the state that have changed is to remember hashes of the files that represent it. Recomputing a hash is fast, and it's only necessary to look at a file if its hash is different. This technique can be applied recursively by hashing folders as well (including the hashes of their contents).

A more complete log makes it possible to undo changes. This log can take up a lot of space if a lot of data is overwritten or deleted, but disk space is often cheap enough that it's a good deal to save enough information to undo several days worth of changes.

Words of wisdom from Phil Neches (founder of Teradata)

1. It's cheaper to replace software than to change it.
2. It's cheaper to process, store, or communicate than to display
3. It's cheaper to be networked than standalone. The implications for software development are now widely accepted: continuous updates, shared data, and availability through replication.
4. Public transactions are cheaper than anonymous ones. This is because of accountability. For example, credit cards are cheaper than cash (after all costs are taken into account).

Finally, software has its face to the user and its back to the wall.