Formal Methods for Design: How To Understand Your System Before (Or After) You Build It

Butler Lampson

blampson@microsoft.com

12 November 2002

My Religion

Write specs as models, not axioms
- Write down the state
- Give the actions, both external and internal

“Implements” is refinement (external behavior a subset)

Safety proofs by abstraction function and simulation
- This is complete: If *Y* implements *X*, there’s an abstraction function under which *Y* simulates *X*
- May need to add history and prophecy variables

Liveness isn’t important—time bounds are safety

Leave encoding and data structures as late as possible

Proving that Y implements X

Define an *abstraction function* *f* from *Y*'s state to *X*'s state.

Show that *Y simulates X*: For each *Y*-action and each state *y* there is a sequence of *X*-actions that is the same externally, such that the diagram commutes.

$$\begin{tikzcd}
f(y) \arrow[r, "\textbf{X-actions}"] & f(y') \\
y \arrow[u, "f"] \arrow[r, "\textbf{Y-action}"'] & y' \arrow[u, "f"']
\end{tikzcd}$$

This always works!

Invariants describe the reachable states of Y; simulation only needs to work from a reachable state.

Understanding A System: What Pays Off?

1. The specification: first the state, then the actions

    Examples: File system, group communication

2. The implementation state and the abstraction function

    Examples: redo recovery, Paxos, security

3. Invariants

    Examples:
    cache, redo recovery

3. Visible transitions

    Examples:
    Paxos, transactions

Hard Questions

What does the system really do?

- File system, group communication

What should you abstract away?

- File system, cache, redo recovery

What are the modules?

- Fedex, group communication, security, Paxos

Can you do any useful proofs?

- Yes: Paxos, cache. No: Fedex, file system

Mental Tools

Sets, functions, relations, graphs

State machines

Modules and composition—TLA, IOA, Z

These are just ways of writing down state machines

Example: File system

The tricky part is specifying happens when there's a crash before a write has made it to the disk.

type Dir = PathName $\rightarrow$ **seq** Byte
var dir : Dir

Write(p, x, data) =
  if crashed **then** *if crashed, write some prefix*
    choose i $\leq$ data.size **do** data := data.subSeq(1, i)
  else skip fi;
  dir(p) := *NewFile*(dir(p), x, data)

If there's no ordering guarantee
  if crashed **then** *if crashed, write some* ***subset***
    choose w $\subseteq$ data.domain **do** data := data.restrict(w)

Buffered File System

With buffered writes, it's even trickier:
any subset of the writes since the last *Sync* can be lost.

type Dir = PathName $\rightarrow$ **seq** Byte
var dir : Dir
oldDirs : **set** Dir := {}

Write(p, x, byte) = **var** f := dir(p) |
dir(p) := *NewFile*(f, x, data);
oldDirs(p) := {f ′ :IN oldDirs(p), w $\subseteq$ data.domain |
NewFile(f ′, x, data.restrict(w))}

Sync() = oldDirs := {dir}

Crash() = **choose** d $\in$ oldDirs **do** dir := d; *Sync*()

Example: FedEx Package Tracking

How to specify the FedEx package tracking system?

First try:

- Packages, locations, transports, routes
- Events: package is seen (scanned), transport moves
- Queries: package history, projected route

Second try:

- Packages, locations
- Events: package is seen (scanned)
- Queries: package history

Modularity: Separate tracking from routing.

An opposite example: specifying I/O hardware.

- Often the clean module includes the driver.

Example: Transactions

The spec: Make a big atomic thing out of small ones.

var	ps	:	S	*Persistent State*
	vs	:	S	*Volatile State*

Do(action) = vs := action(vs)
Commit() = ps := vs
Abort() = vs := ps

Implementing Transactions

Log the actions, commit by persisting the log, update persistent state in background.

Need idempotent actions: $s \oplus log \oplus log = s \oplus log$

var	psI	:	S	*Persistent State*	pLog	:	**seq** Action	*Persistent Log*
	vsI	:	S	*Volatile State*	vLog	:	**seq** Action	*Volatile Log*

abstract

$ps = psI \oplus pLog$

$vs = vsI$

invariant

$vsI = psI \oplus pLog \oplus vLog$

Do(action) = vsI := action(vsI); vLog := vLog + {action}

Commit() = pLog := vLog

Abort() = vs := ps $\oplus$ pLog; vLog := {}

Implementing Transactions (2)

abstract
ps = psI ⊕ pLog
vs = vsI

invariant
vsI = psI ⊕ pLog ⊕ vLog

Persist() = **await** vLog = {a} + tail **do** psI := a(psI); vLog := tail

ps	pLog	vLog
ps_0 ⊕ done	⊕ done + {a} + rest	⊕ {a} + rest
ps_0 ⊕ done ⊕ {a}	⊕ done + {a} + rest	⊕ rest

Cleanup() = **await** vLog = {} **do** pLog := {}
Crash() = vsI := ps ⊕ pLog; vLog := pLog

Example: Redo Recovery

(Lomet and Tuttle)

After *Commit*, we update persistent state in background.
These updates must not change the abstract state.

var sI : S *State* log : **seq** Action

abstract s = sI ⊕ log

Install() = **choose** a ∍ sI ⊕ log = a(sI) ⊕ log **do** sI := a(sI)

Cleanup() = **choose** hd, tail ∍ log = hd + tail **do**

await sI ⊕ log = sI ⊕ tail **do** log:= tail

But a(sI) ⊕ log = sI ⊕ ({a} + log).

So in *Install*, a suitable *a* must be *idle* if prefixed to *log*; it makes no difference to the final state.

Redo Recovery (2)

abstract s = sI ⊕ log

Install() = **choose** a ∍ sI ⊕ log = sI ⊕ ({a} + log) **do** sI := a(sI)

In *Install*, *a* must be *idle* if prefixed to *log*.

How can this happen? Easy case: all actions are *v* := const.

Then any *a* already in *log* will be idle if prefixed.

In a database system, we install actions $v := c_v$, where c_v is the current value of *v* in *vs*, the DB's buffer cache.

If actions read some variables, it's harder to find idle ones.

b is *final* if no busy action later in *log* reads its writes.

b not final *a*: v:=3 ... v:=5 ... *b*: x:=v+2 ... y:=x+4

Appending a final *b*'s writes to *log* makes *b* idle, so installable.

b made idle *a*: v:=3 ... v:=5 ... *b*: x:=v+2 ... x not read ... x:=7

Example: Replicated State Machines

The spec, good for arbitrary deterministic computations.

var s : S *State*

Do(action) = (s, v) := action(s); **return** v

The implementation:

var log : **seq** Action s_p : S *State*

n_p : Nat *last action applied*

abstract $s = s^{\text{initial}} \oplus \text{log}$

invariant $s_p = s^{\text{initial}} \oplus \text{log.subSeq}(1, n_p)$ *states agree with log*

Do(action) =

log := log + {action};

choose $p \ni n_p = \text{log.size} - 1$ **do** $n_p := n_p + 1$; $(s_p, v) :=$ action (s_p);

return v

Catchup$_p$() = **await** $n_p <$ log.size **do** $n_p := n_p + 1$; $s_p := \text{log}(n_p)(s_p).s$

Transfer$_{p,q}$() = **await** $n_p < n_q$ **do** $n_p := n_q$; $s_p := s_q$

Example: Consensus

How do we implement the global log of RSM?
As a sequence of consensus problems, one per log action.

Consensus is tricky, but it's much easier when separated from RSM and from configuration changes.

The spec, good for arbitrary deterministic computations.

var v : (V **or** nil)
allowed : **set** V

Allow(w) = allowed := allowed $\cup$ {w}
Decision() = **return** v **or** **return** nil

Decide() = **choose** w $\in$ allowed **do** v := w

Implementing Consensus: Paxos

The idea: **do** try for consensus on *v* **until** get a majority

The implementation:

var $r_{p,t}$: V **or** no **or** nil *once non-*nil*, cannot change*

abstract v = (**choose** t, v э a majority of $r_{p,t}$ = v **do** v)

A majority must agree with any previous one.
So, try the *v* of the most recent trial that isn't *dead*.

Force trials to die by getting processes to set $r_{p,t}$ to *no*.

When does it decide? When process *p* does $r_{p,t}$ and forms a majority. But no one knows this at the time!

Modularity: Use the RSM to change the set of processes.

Paxos is the best algorithm for asynchronous consensus

By Lamport and Liskov/Oki; Byzantine version by Castro/Liskov.

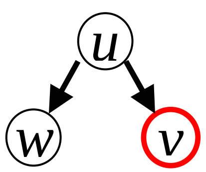
Example: Group Communication

The idea: lots of copies of RSM that form a DAG.

Each copy is called a *view*. It has an initial state and a set of processes that do RSM in the view.

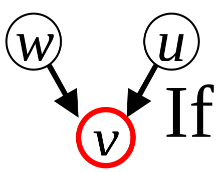
A process is in a sequence of views that's a path in the DAG.

A *view change* forms new views from existing ones.



If *v* has only one parent *u*, *v*'s initial state is *u*'s final state (perhaps with some suffix of actions dropped).

Virtual synchrony ensures that all processes moving to *v* see the same actions in *u*, hence have the same final state.



If *v* has more than one parent, must *merge* their final states to get *v*'s initial state.

Modularity: Application-dependent. Easy if actions commute.

Example: Security

Principals: abstraction of “who says?” or “who is trusted?”

P **says** “read file `foo`”

Speaks for: abstraction of trust or responsibility

P **speaks for** Q — if P says something, so does Q

Examples

Key 743891743 **speaks for** blampson@microsoft.com

blampson@microsoft **speaks for** researchers@microsoft

blampson@microsoft **speaks for**$_{\text{read/write}}$ research.microsoft.com/lampson

This logic abstracts crypto, physical security, encoding, etc.

The soundness of the abstraction is the hardest part.

Can do positive proofs in the logic,
negative ones by simulation or model checking.

Example: Cache

var m : $A \to V$

Read(a) = **return** m(a)   *Write*(a, v) = m(a) := v

Implementation:

var mI : $A \to V$

c : $A \to$ (V **or** nil)

def a.live $\equiv$ c(a) $\neq$ nil

a.dirty $\equiv$ a.live $\wedge$ c(a) $\neq$ m(a)

abstract m(a) = **if** a.live **then** c(a) **else** mI(a)

invariant {a | c(a) $\neq$ nil}.size $\leq$ N

Read(a) = **await** a.live **do return** c(a)

Write(a, v) = **await** a.live **do** c(a) := v

MtoC(a) = **if** a.live **then skip**

else choose a′ **do** *CtoM*(a′) **od**; c(a) := m(a)

CtoM(a) = **if** a.dirty **then** m(a) := c(a) **else skip fi**; c(a) := nil

Multiprocessor Cache

var	mI	$: A \to V$	$locked_p$	$: A \to Bool$
	c_p	$: A \to (V$ **or** $nil)$	$dirty_p$	$: A \to Bool$

def	$a.clean$	$\equiv (\forall p \mid \sim a.dirty_p)$	$a.live_p$	$\equiv c_p(a) \neq nil$
	$a.free$	$\equiv (\forall p \mid \sim a.locked_p)$	$a.current_p$	$\equiv (c_p(a) = m(a))$
			$a.only_p$	$\equiv (\forall q \neq p \mid \sim a.live_q)$

abstract $m(a) =$ (**if** $a.clean$ **then** $mI(a)$
else choose $p \ni a.dirty_p$ **then** $c_p(a)$)

invariant $a.dirty_p \Rightarrow a.live_p \Rightarrow a.current_p$
$a.dirty_p \Rightarrow a.locked_p \Rightarrow \sim a.live_q \wedge \sim a.locked_q$

Read(a) = **await** $a.live_p$ **do return** $c_p(a)$
Write(a, v) = **await** $a.locked_p$ **do** $c_p(a) := v$; $a.dirty_p :=$ true

Multiprocessor Cache (2)

def $a.clean \equiv (\forall p \mid \sim a.dirty_p)$ $\quad a.live_p \equiv c_p(a) \neq nil$

$a.free \equiv (\forall p \mid \sim a.locked_p)$ $\quad a.current_p \equiv (c_p(a) = m(a))$

$a.only_p \equiv (\forall q \neq p \mid \sim a.live_q)$

invariant $a.dirty_p \Rightarrow a.live_p \Rightarrow a.current_p$

$a.dirty_p \Rightarrow a.locked_p \Rightarrow \sim a.live_q \wedge \sim a.locked_q$

Read(a) = **await** $a.live_p$ **do return** $c_p(a)$

Write(a, v) = **await** $a.locked_p$ **do** $c_p(a) := v; a.dirty_p := true$

$MtoC_p(a)$ = **await** $\sim a.dirty_p \wedge (a.locked_p \vee a.free)$ **do** $c_p(a) := m(a)$

$CtoM_p(a)$ = **await** $a.dirty_p$ **do** $m(a) := c_p(a); a.dirty_p := false$

$Drop_q(a)$ = **await** $\sim a.dirty_p$ **do** $c_p(a) := nil$

$Acquire_p(a)$ = **await** $a.free \wedge a.only_p$ **do** $a.locked_p := true$

$Release_p(a)$ = **await** $\sim a.dirty_p$ **do** $a.locked_p := false$

$CtoC_{p,q}(a)$ = **await** $a.free \wedge a.live_p$ **do** $c_q(a) := c_p(a)$

Marketing

To sell, you must have "metal" tools that help the developer

- Type-checking and other kinds of abstract execution
- Model-checking of important properties
- Proofs (usually only for hardware)
- Test coverage analysis

A crisis helps—floating divide bug, buffer overruns

Sometimes a fad will do—the internet sold type-checking and GC in Java. But it must be automated.

Why so hard? Willpower is best as long as it works.

- But often you find out only later that it's not working.

Proofs?

Many things are possible—cost-benefit is the issue

Some things that have worked:

- Simple properties of software: type-correct, no races, device driver follows OS protocol
- Proofs of tricky algorithms, especially concurrent ones
- Hardware, esp. model checking

Sound and complete? No.

- "Sorry, I can't find any more bugs."

Acknowledgements

I learned a lot about this from

Leslie Lamport

Nancy Lynch

Tony Hoare

Martín Abadi

Further reading:

Principles of Computer Systems

For security, *Computer Security in the Real World*

For consensus, *ABCD's of Paxos*

All are at research.microsoft.com/lampson.