On the Transfer of Control between Contexts

B. W. Lampson, J. G. Mitchell and E. H. Satterthwaite
Xerox Research Center
3180 Porter Drive
Palo Alto, CA 94304, USA

Lecture Notes in Computer Science 19, Springer, 1974, pp 181-203

Abstract

We describe a single primitive mechanism for transferring control from one module to another, and show how this mechanism, together with suitable facilities for record handling and storage allocation, can be used to construct a variety of higher-level transfer disciplines. Procedure and function calls, coroutine linkages, non-local gotos, and signals can all be specified and implemented in a compatible way. The conventions for storage allocation and name binding associated with control transfers are also under the programmer's control. Two new control disciplines are defined: a generalization of coroutines, and a facility for handling errors and unusual conditions which arise during program execution. Examples are drawn from the Modular Programming Language, in which all of the facilities described are being implemented.

1. Introduction

Transfers of control in programs can be divided into two classes. A *local* transfer stays within the same piece of program text, and does not change the naming environment. A goto which does not involve an exit from a block has traditionally been the primitive local transfer operation, and other operations have been described by translating them into sequences of (possibly conditional) gotos and assignments. Recently there has been a lot of effort to find a good set of higher-level local transfer operations, motivated by an awareness that the undisciplined use of the goto results in badly structured programs. The choice of if-then-else, for-while and case constructs, sometimes augmented by loop and exit operations, has met with wide acceptance. This is not because of theoretical proofs that they are sufficient to express any computation, but because many years of experimentation with the possibilities of the goto showed that it is most effectively used in a few stylized ways, from which these constructs were abstracted. In fact, the arguments for keeping goto available in programming languages are based on the observation that there are times when its use cannot readily be cast in one of these molds.

A *global* transfer, on the other hand, does more than alter the sequential

flow of control. It usually invokes a new piece of program text, and it always affects the allocation of storage and the binding of names. This paper is about global transfers. In fact, it is an attempt to find a suitable primitive (which we will call transfer) and to describe higher-level global transfers or *control disciplines* by translating them into sequences of transfers, assignments and other data-handling operations.

There are two reasons why this seems worthwhile. First, it is difficult to describe clearly how the control disciplines in existing languages work without resorting to the construction of a formal interpreter [Fisher]. Non-interpretive descriptions either contain large quantities of ambiguous English prose, or they involve operations (such as the Algol 60 copy rule for procedure calls) which may be precise but are certainly not clear. If a language can be used to describe itself, by defining certain operations in terms of sequences of simpler operations in the language, the amount of conceptual baggage required to understand it can be reduced considerably.

Second, it is our opinion that much remains to be learned about the proper choice of global transfer operations. Until recently very few languages other than assemblers gave the programmer any choice of control operations. Simula [Hoare and Dahl] and the new crop of languages for artificial intelligence have changed this situation to some extent, but it will be a long time before the possibilities for global transfers have been thoroughly explored. If programmers have the opportunity to create their own control disciplines they will certainly make a lot of mistakes, but from this experimentation we can hope to learn what works well and what does not.

From this discussion the flavor of the paper should be clear. We will define a transfer primitive and then exploit the local expressive power of the language to describe some control disciplines which we happen to like. This is not a trivial job, since a good discipline must satisfy a number of constraints:

- programming generality [Dennis] - independently constructed modules can work together without having to know each other's internal structure. In particular, each module can choose its names and its storage allocation strategies independently of the others;
- compatibility - modules using different disciplines can still communicate, or the advantages of diversity will be completely overwhelmed by the drawbacks of Babel;
- robustness - it is easy to get things set up, and restrictions and caveats are conspicious by their absence;
- reconfigurability - connections betweeen modules can easily be broken and reestablished, so that debugging facilities can be spliced in and the behavior of a module can be changed by attaching "adaptors" to its external connections.

This approach should not be misinterpreted. The fact that a construct can be explicated in terms of simpler ones does not mean that the programmer must have this decomposition in mind whenever he uses it. On the contrary, if he uses it as part of his working vocabulary he will normally think of it as an atomic concept. The explication is helpful in making the definition precise, and in answering questions about what will happen in unfamiliar situations; it should be thought of in that light.

Questions about the binding of names are, in our view, orthogonal to the study

of transfers, and are not considered in this paper. In particular, rules for binding non-local variables and for linking separately compiled modules are not discussed.

2. The host language

The facilities described in this paper are implemented in a general purpose system programming language called the Modular Programming Language (MPL), which is a component of a system for modular programming. MPL borrows much of its local character from Pascal [Wirth] and EL/1 [Wegbreit]. In particular, it is a typed language in which new types can be built out of old ones using *record*, *array* and *pointer* declarations. There is also a way to prevent components of a record from being accessed except by a group of procedures which are declared with it. Such a record is called *closed*, and the procedures are called its *handles*. Finally, it is possible to declare a record type s as a *direct extension* of another record type r, by adding additional components and handles. *Extension* is the transitive closure of direct extension, and the set of all extensions of r is the *class* of r. Closed records and classes were inspired by the *class* mechanism of Simula [Hoare and Dahl]; that language, however, encourages the restriction of access to handles, but does not enforce it. The control transfer operations use closed records and classes to construct and manipulate their data structures.

A construct patterned after Pascal's with is used heavily as syntactic sugar by the control disciplines. Any block can be prefixed by one or more clauses of the form:

USING p,

where p is a pointer to a record of type r. Within the block the names of the components of r can then be used without qualification. If c is such a component, then within the block c is short for p.c.

3. Contexts and frames

The entities between which global transfers occur we call *contexts*. The definition which follows reflects our views about the properties which such entities ought to have. Although nearly all of the transfer operations of existing programming languages can be described within this framework, we are not making any claims for its universal applicability. Since we make use of its properties in constructing control disciplines, our constructions will not work in systems for which contexts cannot be defined.

Within this framework we may restate the subject matter of this paper:

- the nature of contexts, and their creation and destruction;
- minimal primitives which are sufficient to describe any transfer of control between contexts;
- definition of good higher-level transfer disciplines;
- description of these disciplines in terms of the primitives.

A context consists of:

- a pointer to the text of a *program*, which we shall abstract as an array of objects called *instructions* whose internal structure and

properties are left undefined. We assume that the program is not modified during execution;
. a *binding rule* for names, which we shall abstract as a function mapping names into pointers;
. some *local storage*, including an integer index into the program text called the *program counter*.

3.1 Representation of contexts

A context is represented by a *frame*, which is a record whose components contain the information needed to define the context. More precisely, a *context base* is a closed record type containing a program text pointer, a binding rule and a program counter; these components are accessible only to the control transfer primitives, which are the handles. A context can almost be described as a member of the class of context bases. Unfortunately, this description cannot quite be taken literally, because we need the **transfer** primitive to describe how procedures are called, and hence this primitive cannot be defined as a procedure. The other operations on contexts, however, can properly be defined in this way.

It is interesting to compare this situation with what happens if we try to define as a class some other type which is normally taken as primitive. A 32 bit integer, for example, could be defined as a closed record containing a Boolean array with 32 elements, and we could (rather clumsily) write procedures to implement the standard arithmetic operations, without becoming involved in any circularity. As with any other closed record, these procedures must cooperate in maintaining the consistency of the representation: if **add** assumes that the integer is represented in 2's complement, then **multiply** had better not assume sign-magnitude representation. All the questions of consistency are out in the open, however, since everything having to do with the closed record is expressed in the declaration of its handles.

For contexts the consistency requirement has a new aspect. The procedure which creates a context, for instance, must build a data structure which is consistent not only with the other handles of the class context, but also with the **transfer** primitive. This is actually a rather strong requirement, because the **transfer** primitive causes instructions to be executed from the program text. When this text was constructed, some assumptions were made (by the compiler) about the environment which would be present during execution of the program. The creation procedure is responsible for setting up the environment so that these assumptions are satisfied. If they are not, chaos will result, since the foundation will be undermined on which the entire representation of the program is based. If care is taken to satisfy the assumptions of the **transfer** primitive, then, we may think of a context as a class, and the remaining discussion will proceed on that basis.

We will call a pointer to a context an *inport*. The name is intended to suggest the main purpose of this type, which is to be an operand for the **transfer** primitive. A pointer to an inport we will call an *outport*, with the idea that most of the control disciplines we are interested in need this extra level of indirection so that transfers into a context can be trapped when necessary.

3.2 Creation of contexts

We now proceed to explore in detail how contexts are created. Our discussion concentrates on the logical structure of the creation process, ignoring the details of the implementation, in which much of the work is done at compile or link time, and many of the operations described are coalesced for efficiency. We say a good deal about the treatment of the types of the various objects involved in order to make it clear that everything we are doing is consistent with the constraints of a fully typed language.

Since a context is an instance of a class, there must be a single create primitive which takes some arguments and creates a context. The arguments to create are:

. a record called a *program* which contains
  - an array of program text,
  - the type of the frame record which the program expects;
. a frame record (which is not yet a context).

We will consider later where these records come from.

With these inputs, create's job is easy. It checks that the frame record actually presented is of the type specified by the program (using the facilities of the type system to find out what the frame's type is). Then it inserts a pointer to the program text array in the frame, initializes the program counter to zero, and returns the frame record as a context.

A program must be derived eventually from a source file which has been compiled by the MPL compiler. The output of the compiler is an object file which contains the same information as the program record. There is an operation called load which converts a file name into a program record, after checking as best it can that the file is in fact a legitimate object file (the type checking machinery cannot be expected to handle this situation perfectly, since it has no control over the way in which information is stored in the file system). The only hard work load has to do is to find space for the program text array in the addressable memory of the machine on which the program is running. How this is done depends on the details of the machine and is not relevant to this paper.

Constructing a frame is more difficult. Again, we can break this operation down into two parts:

. obtaining storage for the frame;
. initializing this storage properly and returning it as the frame.

Any record type in MPL has a creation operation associated with it which is defined when the type is declared. This operation accepts a block of storage and perhaps some other parameters, and produces a record of the proper type. It is the only way to make such a record. The create primitive for contexts discussed above is an example of such an operation.

An ordinary frame creator in MPL is a special case of this general mechanism, with two distinctive characteristics. First, it usually has a standard program text, for two reasons:

. frames tend to have a rather stylized form, so that the differences between them can be efficiently encoded into a data structure called a *frame descriptor* which can then be accepted as a parameter and interpreted by the standard creator program.
. there is another circularity problem - someone has to create the frame

creator. A standard creator can itself be created in a standard way which can be part of the initial system.

The frame descriptor is usually stored in the object file along with the program text. Use of this scheme is not compulsory, however. All of the facilities for creating records can be used to create frames.

The second unusual thing about a frame creator is that it has to provide the binding function for the context. Recall that this function maps the names used in the program text into pointers to the objects which are bound to those names by this incarnation of the program. This is done by a generalization of the *display* which is often used to implement the binding rules of Algol.
The names used by the program have the form rp.v, where rp is a pointer to the frame of some other context and v is a variable local to that context. We call the set of frames r, s, t, ... referenced by a context in this way the *neighborhood* of the context; it is defined by a collection rp, sp, tp, ... of pointers to the frames. The context is automatically prefixed with a clause of the form

USING rp, sp, tp, ...

so that the program can refer to the variables without qualification, just as it refers to non-local variables in Algol. One element of the neighborhood is always the argument record.

To define the binding function, then, the creator has to define the neighborhood, i.e. set the pointers rp, sp, tp, ... to the proper frames. The type of each frame is of course fixed by the declarations in the source program, but there may several frames of the same type to choose from. As far as the typing and control mechanisms of the language are concerned, the creator is free to choose any of them. One familiar possibility is the Algol rule, which takes the unique textually enclosing occurrence [Wang and Dahl]. In a more complex control environment, however, it may be difficult to define such a unique occurrence, or the programmer may want more flexibility in defining the environment. In any event, the choice of binding rule is entirely under the programmer's control and is not relevant to the subject of this paper.

3.3 Storage allocation

There remains the question of storage allocation for a frame. This must be done with some care, since creating a context is a rather common operation which is required, for example, by every call of an Algol-like procedure. The standard solution is to allocate frames from a stack; this works well in a control discipline which ensures that contexts are created and destroyed in last-in first-out fashion. Such a restriction would not be incompatible with the basic control primitives, but it would severely constrain the set of compatible higher-level disciplines which would be designed.

In order to avoid this problem, we have made a convention that frames are allocated on a heap; they can then be created and destroyed in any order. A standard non-compacting, coalescing free storage allocator [Knuth] is used, supplemented for speed by a vector of lists of available blocks for all the commonly used sizes. To keep the vector short, frame sizes are quantized by the compiler so that they differ by about 10%. Thus the possible sizes might be 10, 12, 14, 16, 18, 20, 22, ..., 200, 220, etc. With this scheme only 40 sizes are required to span the range from 10 to 320, which is a much greater variation than is likely to be encountered in practice. Furthermore, it is

always possible to allocate a larger block than the one requested in order to reduce external fragmentation.

4. The transfer primitive

As we have already seen, in order to handle transfers which change the environment we need at least one language feature orthogonal to that subset of the language which is used for programs which run in a single environment. This section describes a single primitive called **transfer** to meet this requirement. We have tried to make this primitive do a minumum amount of work, leaving everything possible to be done by local code surrounding it in the two contexts which are involved in the transfer.

The basic **transfer** primitive, then, takes an inport as its single argument. After it has been executed, the context which executed it is no longer running, and the context specified by the inport has started running at the location specified by its program counter. In fact, this operation bears a striking similarity to the primitive used in Multics for switching control from one process to another [Saltzer], where the system scheduler, running within a user process, picks another process to run and transfers to it. The difference is that in Multics there is no relationship between the processes except for that established by the implementation of the scheduler.

In our case, however, we almost always want to pass some kind of return link and some arguments to the new context. We do this by establishing the convention that the link should be put into a global variable called *link*, and the argument into another global called *args*, before the **transfer** is executed. The context being entered must use the values of these variables, if it cares, before doing another **transfer**. Since this convention is followed in all our examples, the remainder of the paper uses a three-argument primitive

transfer(destination inport, return outport, argument pointer).

as an abbreviation for

dest := destination inport; *link* := return outport;
args := argument pointer; **transfer**;

In the implementation the global variables *dest*, *link* and *args* are of course machine registers.

For obvious reasons we make *args* a pointer to the argument record. From the point of view of the type machinery, this will be a "universal" pointer which carries its type with it. When the receiving context tries to use it, a run-time type check is needed to ensure that it actually has the proper type. In most cases, however, this check can be done at binding time, as we shall see later.

Note that the **transfer** primitive says nothing about what is to be done with the link or the arguments, and it does not create any contexts or allocate any storage. All of this is the responsibility of higher-level conventions or control disciplines, and the existing local features of the language, together with **transfer**, are sufficient to permit almost all of the transfer operations we know about to be programmed. An actual implementation, of course, may favor certain disciplines by pre-defining them in a standard prologue and generating especially good code for them, as ours does for the port, procedure and signal disciplines described below.

5. Conventions for compatible transfers

In defining control disciplines, we would like to have as much compatibility as possible, so that it is possible to leave a context using one discipline and enter a second context using a different one. To make this work, we must be careful about storage allocation and about the rules for handling the arguments and return links. We have already discussed a suitably general method for allocating frames. This section considers the other general problems encountered in designing a fairly broad set of compatible control disciplines.

The transfer primitive allows for a single argument, which is normally a pointer to the record containing the arguments which the user wanted to pass. The semantics of binding a formal parameter, say x, to an actual parameter, say 14, is very simple. The sender of the argument record assigns the actual parameter to a suitably named component of the argument record (actualargs.x := 14). When he has finished constructing the record and is ready to transfer, he does

args := actualargs; transfer(destination).

The receiver does

formalargs := *args*,

and (automatically) prefixes his block with the clause

USING formalargs,

where formalargs is declared to have the type of the argument record he expects.

The effect of all this is that:

. the low-level convention for passing arguments is very simple - one pointer is passed;

. the entire collection of arguments is treated as a unit, so that it can be passed on unchanged by a context which is simply doing monitoring or tracing and is not interested in the internal structure of the arguments;

. the receiver can reference the formals with the usual syntax;

. the language facilities for constructing and decomposing records are automatically available for arguments. These allow, among other things

- component values to be specified by name, by position or by default;
- a record to be decomposed by assigning it to an *extractor*, a syntactic construct which looks exactly like a record constructor except that all the components are treated as left-hand-sides of assignment operators;
- variable-length records.

In this way a fairly elaborate set of facilities is made to do double duty without any need to introduce new semantics into the language.

To preserve generality, we must ensure that the storage occupied by the argument record will not be reused until the receiver is through with it. It is undesirable to put this storage in the sender's frame, as is customary in Algol implementations, because the sender's frame may not live as long as the receiver (e.g. when the sender is a returning procedure; this case can be handled specially in Algol because of the restrictions on what a function can return). We therefore allocate separate storage for the arguments, and require the receiver to free this storage when he is done with it.

Copying the entire argument into the receiver's frame would be another

alternative, but is is unattractive for variable length argument records and in situations where a receiver is not interested in the values of the arguments, but is simply going to pass them on to someone else. Copying does work well for short argument records, however, especially since the record can be constructed in the machine's registers, and this strategy is used for records of less than 6 words.

6. Coroutines and ports

In this section we take up a pair of control disciplines which treat the two parties to a transfer as equals. In particular, this means that no creation of contexts or allocation of storage is involved in a transfer, and that the relation between the parties is symmetric - each thinks that it is calling on the other one.

6.1 Coroutines

A *coroutine* (more or less as in Simula [Hoare and Dahl]; see also [McIlroy] and [Conway]) is a context which, when entered, continues execution where it left off the last time it relinquished control. Local storage survives unchanged from exit to entry (as in a Fortran procedure, interestingly enough). This is the simplest control discipline, and the easiest to describe. Each context is pointed to by static inports set up at link time. Hence a transfer passes no return outport. The linkages are normally symmetric, as shown in figure 1.

There are three problems with coroutines of this kind as a general-purpose control discipline. One is that, because of the fixed linkages, a coroutine cannot be used to provide a service to more than one customer. A procedure, by contrast, is ideally suited for this purpose, since it is created as a result of a call and destroyed when its work is done.

A second difficulty is that the control is entirely anarchic. There is nothing to prevent control from entering a coroutine in an entirely unsymmetric way. For example, in figure 1 context Q might gain control over inport a from line s1 of P, even though its program counter is at t1. If subjected to an appropriate discipline this kind of control transfer might be useful, but no such discipline is present in the simple coroutine scheme.

6.2 Initialization of coroutines

The third problem is proper initialization of a collection of coroutines. Recall that a transfer from context P to context Q does not change Q's program counter, but simply causes execution to resume at the point where it stopped, or at the beginning if Q has never run before. Since no buffering of *args* or *link* is provided by transfer, Q must save their values before doing another transfer. In general it will do this properly only if it is sitting immediately after a transfer to P. In figure 1, for example, if P is started first, it will transfer to Q at s1, but Q will transfer to R and thus lose P's argument record.

This difficulty can be reduced by initializing more cautiously, as follows:

(a) Start each context in turn by transfering to it, let it run up to its

first transfer, and stop it before it sets up *args* and *link*.
(b) Carefully choose one of the contexts and restart it by transferring to it.

Step (a) is unattractive because it requires a kind of control over the internal activities of the contexts which is quite different from what is needed for normal transfers. Step (b) has more serious problems, which will become apparent on further examination.

Suppose in figure 1 that P is acting as a producer of data and Q as a consumer who may occasionally return a reply. The fact that P and Q play different roles is concealed in the figure by the identical form of the skeletal program text. In figure 2 this difference has been brought out by expanding the argument handling associated with each transfer into *send* and *receive* operations. The sequence of processing is:

P: setup - send - transfer - receive - compute - send - transfer -
...
Q: setup - - transfer - receive - compute - send - transfer -
...

The two sequences are identical except for the phase at initialization: in both cases there is a send - transfer - receive sequence which is the expansion of the simple transfer of figure 1.

The difference in phase is quite important, however, for step (b) of our cautious initialization procedure. If we choose P to restart, it will immediately transfer to Q, which will immediately transfer back, and P's first message will be lost. If, on the other hand, we choose Q to restart, all will be well. Unfortunately, it is hard to see how to make the proper choice in more complex situations (if indeed it is always possible).

6.3 Processes and messages as a model

Rather than making further attempts to patch up the simple coroutine discipline, we now turn to a much more powerful scheme: processes executing in parallel and communicating via event channels. This, of course, is more power than we need or want, but by extracting the essential functions of the parallelism and message buffering we can design a control discipline with understandable properties which preserves the strengths of coroutines while avoiding their problems.

The idea of processes executing in parallel we assume to be familiar [Dijkstra]. A message channel is an object on which two basic actions can be performed by a process: send a message and receive a message. A message is an arbitrary record, and the channel can buffer an arbitrary number of messages. An attempt to receive a message from an empty channel causes the receiving process to wait until a message is sent to that channel. There is no constraint on the number of processes which can send or receive messages on a given channel. This facility is synthesized from two operating systems [Lampson, Brinch Hansen]; we have suppressed many details which are irrelevant to our purpose.

Any transfer operation can now be modeled by some combination of send and receive. We don't have to worry about losing messages, because of the buffering provided by the channels; each process will get around to processing its messages in due course. Nor is the order in which

processes run of any importance; in fact, it is not even defined, except when processes must wait for messages. We still need a convention which allows one process to provide service for many customers, however. We get it by analogy with the *link* parameter of the transfer primitive: an event channel on which to return a reply goes along with each message.

6.4 Ports

The process-channel model has added three essential features to the coroutine discipline:

- parallel execution;
- buffering of messages;
- indirect access to processes through message channels.

Figure 3 illustrates the structure of a symmetric connection. We now proceed to adapt these features to a sequential, unbuffered environment. The first step is to define a new type for symmetric transfer of control, called a *port*, to replace the <channel, outport> pairs in figure 3 [Balzer, Krutar]. Each port is likewise a pair, consisting of an inport IP and an outport OP. IP points to the context which will get control when a transfer is made through this port, and OP is where the return link will be stored.

We can avoid the need for parallel execution in a straightforward way, by modeling the notion of "a process waiting for a message on a channel" with the new concept of "a context being *pending* on an inport". Since a process can only be waiting on one channel, we will insist that a context can only be pending on one inport. Now, if all transfers are to pending inports, it will always be possible to run the context to which a transfer is directed, and there will be no need for parallel execution. A transfer which does not obey this rule will not be executed, but instead will cause a *control fault*, with consequences which we will explore shortly.

Rather than explicitly associating the attribute "pending" with each inport, we can observe that an inport is a capability to start execution of a context, and interpret the pending rule as a requirement that only one non-null inport at a time should exist for each context. The inport components of all the other ports associated with a context will be null, and a transfer to a null inport will cause a control fault. We thus complicate the semantics of transfer as little as possible.

Note that the pending rule has nothing to do with the transfer primitive, but is a convention which we introduce in order to construct a useful higher-level control discipline, that of ports. Even within this context, it may be proper to break the rule if it can be shown that no untoward consequences will result. Since the rule is strictly internal to the port discipline, it stands or falls solely on the consistency of that discipline, and it is entirely independent of the requirements of any other, separate convention for control transfers. We do, however, want it to be compatible with a procedure discipline; fortunately, this causes no trouble.

A context gets to be pending on an inport in the same way that a process gets to be waiting for a message on a particular channel: by executing a receive operation on the port containing that inport. There is a definite relationship between the value of the program counter and the pending inport: the program is at the point where it expects control to arrive over that port.

As a result, there is no need for message buffering in a successful port transfer, since the receiving context is ready and willing to pick up the message at once.

6.5 Control faults and message buffering

During normal execution a control fault indicates an error, an attempt to transfer control to a context which was not interested in receiving it in that way. During initialization, on the other hand, a control fault may simply be an indication that there is another context to start. When a fault occurs, therefore, control is passed to the *owner* of the faulting context; the owning context must decide whether another context should be started. The mechanism by which this is done is described in section 9. Here we confine ourselves to the local consequences of the fault. The argument used above to show that no message buffering is required depended on the absence of control faults. When a fault does occur, what action should be taken to ensure that no messages are lost?

First of all, if no message is being sent (i.e. *args* is null) there is no need for buffering. For instance, when two contexts have a strict producer-consumer relationship, transfers from the consumer to the producer involve no message. This explains why no special action was needed during the simple coroutine initialization (discussed in section 6.2 above) when we chose to restart the consumer.

When a control fault occurs during a transfer from P to Q (see figure 5) and *args* is not null, we actually have to do something. We would like not to introduce any new kinds of objects, and not to complicate any existing operations. Since our repertoire of objects and operations is limited, things look unpromising at first sight. Fortunately, however, we do have contexts at our disposal, and within a context we can embed any kind of special processing and storage we want, as long as it interfaces properly to the rest of the world.

In particular, what we can do is to construct a *buffer context* B with a standard program text, and local storage within which we keep the argument. We want B to emit the argument the next time control is transferred to P through the port a. To get this effect, we put an inport for B into a's inport component, and save the inport for P which normally is there in B's local storage. When B gets control, it will restore P's inport, transmit the saved argument and destroy itself. It does this by executing:

```
a.inport := savedinport; transfer(DC, B, (a, savedargument));
```

where DC is a system-provided context which destroys B and then does:

```
transfer(a.outport, address(a.inport), savedargument);
```

The cost of all this machination is quite moderate (which is not actually very important, since control faults take place only at initialization if there are no errors), and it has the great advantage that normal transfers are not complicated at all by the requirements of control faults. Figure 5 illustrates the successive stages of initialization for our familiar two-context example.

6.6 Linkage faults

We also want to be able to do dynamic linking, as in Multics [Bensoussan et. al], so that we must be prepared to deal with a transfer through an

outport which has not yet been defined. Fortunately, the techniques we have developed can handle this situation without difficulty. Undefined outports are initialized to point to a standard context which constructs a buffer context, if necessary, to save the argument of the transfer which caused the linkage fault, and then passes the fault on to the owner of the faulting context. If the owner fills in the outport and transfers to it, everything will proceed exactly as for a control fault. Indeed, it is quite possible that a control fault will then occur.

6.7 Railroad switching

As we have already pointed out in passing, the outport component of a port is used to hold the return *link* passed by transfer. Figure 6 makes the purpose of this arrangement clearer. If context Q transfers through port q which is joined to context R through port r, then r.outport is set to q. A subsequent transfer through r will then return control to Q. If later P transfers through p to r, then r.outport will be reset to p, so that control will subsequently return to P. This action, which resembles the action of a spring-loaded railroad switch, allows many-to-one connections of ports, and provides the memory required to return control correctly. Switching is done by the receiving context, since it is part of the port control discipline and has nothing to do with the transfer primitive. Often it produces no change, as for example in the transfers from R back to P or Q. To preserve compatibility with procedure returns (section 7.3) we make the convention that a null *link* suppresses switching.

7. Procedures

Procedures have semantics much like that of Algol procedures. The implementation makes use of almost all the facilities which have been described in the preceding sections.

7.1 Procedure calls

If p is declared as a procedure, then p(a,b,...) is a procedure call, just as it would be a port call if p had been declared as a port. There are two differences:

- a procedure p is simply an outport; all procedure calls from a given context share a single inport in the frame, called the *shared inport*. Since only one such call can be outstanding at a time (because of the pending rule), the pair (p, shared inport) behaves exactly like a port.
- there is no switching done when control returns from a procedure call, because the call is regarded as a completed event, which may be repeated but cannot be resumed.

Because of the way in which responsibility is distributed during a transfer, these attributes of a procedure call are not visible to the context which receives control, but are solely the local responsibility of the context making the call.

7.2 Procedure entry

Whenever a procedure is entered, a context P must be created. This is done

by another context C called the creator, as discussed in section 3. Since the transfer which results in creation of a procedure context is not special in any way, the creator must also take care to start the newly created context and pass it the argument supplied by the transfer. Furthermore, C must leave itself ready to create additional contexts, since the procedure may be entered recursively. Thus there must be a unique inport for C, and the behavior of C must be constant with respect to all transfers through that inport. On the other hand, C is basically an artifact introduced to obtain a uniform control interface, and there is no reason for it to be involved in the return of control from P.

The consequence of these design constraints is that the transfer operation which suspends the creator is used in a somewhat unconventional manner. The *link* that was received in C when it was started is simply passed on to P. The inport through which control arrived in C remains unchanged, and C loops right after the transfer to P, so that it will execute the same code the next time it gets control.

The following, somewhat simplified code describes the body of a typical procedure creator:

```
start: q := ALLOCATE(framesize);
       q.pc := initialpc; q.neighborhood := accesslink;
       q.sharedport := NIL; q.startport := q;
       transfer(q.startport, link, args);
       goto start
```

When the creator is created, the values of the local variables framesize and initialpc are extracted from the text of Q, and accesslink is set up based on the rules for defining the binding function.

This code is misleadingly long in the sense that a clever implementation can achieve the effect it describes with just a few machine instructions, and it is too short in the sense that the "ALLOCATE" operator conceals some additional complexity; it has been discussed in section 3.

7.3 Procedure return

A procedure context has an outport called *returnport* into which it puts *link* when it is entered. A first stab at its return sequence would be:

```
transfer(returnport, NIL, returnargs);
```

but this won't do, because it ignores the fact that the context must be destroyed as part of the return. The caller cannot be expected to take care of this, since he doesn't necessarily know that he called a procedure. The actual return, then, is more like the sequence used by a buffer context (section 6.5):

```
transfer(DP, self, (returnport, returnargs));
```

where DP is a standard context which destroys *link* and then does

```
transfer(returnport, NIL, returnargs);
```

Note that this, like the procedure call described in section 7.1, is fully compatible and does not, for example, depend on any assumptions about the nature of the context pointed to by the returnport. Furthermore, an arbitrary return record can be transmitted. The null *link* suppresses railroad switching if the call was made through a port (see section 6.7).

8. Signals

Finally, we take up a control discipline designed to handle exceptional events efficiently and conveniently. The basic elements of this discipline are:

. a set of names for events, called *signal codes* (e.g. "out of storage", "overflow");

. for each context, an ordered set of outports called *handlers*;

. a system procedure called the *signaller* whose argument s is a pair (signal code, argument record).

8.1 Signalling

Anyone can signal the occurrence of an event by calling the signaller with the appropriate signal code as an argument, thus:

```
signal(OutOfStorage, spaceneeded);
```

The second argument may be an arbitrary record which can be used to pass additional information to the handler. There is also an optional third argument which specifies the context in which the signal should be generated; usually this is the current context. An identifier declared as a signal code is treated by default like an identifier declared as a procedure: a search is made, according to whatever binding rules are in force, for a definition which can be bound to the identifier. The value of a signal code is simply an integer, guaranteed to be different for different codes. Its only purpose is to permit two signal codes to be compared for equality.

The signaller calls the first handler, passing it s as an argument. If the handler returns to the signaller, its result r is a pair (action, return record). If the action is *reject*, the signaller tries the next handler; if it is *resume*, the signaller returns the return record to its caller.

Usually each context supplies a handler, starting with the current context, and the handlers are ordered by a pointer in each context called the *signal port*, which can be set by the user. The default choice of signal port for a procedure is the return link. Thus if all the contexts were Algol procedures, the effect would be to search up the stack, trying each procedure to see if it was interested in the signal.

Normally, handlers are declared in line with the program text of the context which will supply them, and there is convenient syntax for declaring a handler with each control transfer and with each block. If several handlers are declared in a context, they are concatenated into a single one, using the same rule that the signaller uses. These declared handlers have the form of case statements which test the value of the signal code. By writing **any** as a case, however, the programmer can get hold of all the signals that go by and apply his own tests to them.

Thus, for example, one can write:

```
...
begin
   enabling OutOfStorage:
      begin print("Storage exhausted"); exit computation end

   ...
   BuildTable(x, y
      enabling OutOfStorage(spaceneeded: integer):
         if tablespace > 1000 then return
```

```
        GetTableSpace(spaceneeded)
        else reject );
    ...
end
...
```

If the OutOfStorage signal is generated within the call of BuildTable, it will first be given to the handler associated with the call of BuildTable, and then to the handler for the block. The first (innermost) handler checks to see if more space is available. If so, it obtains the space and returns it to the context which did the signalling. If not, it rejects the signal, and it is passed to the handler for the block, which prints an error message and does a (structured) non-local goto. The consequences of this last action are discussed later.

The handlers have the same semantics as ordinary procedures, differing only in the syntax for declaring them. Furthermore, the programmer is free to provide his own handler for a context; all he has to do is to put an outport into the component called handler in the context's frame. The handlers declared with **enabling** have some advantages, however. A great deal of trouble is taken to make the cost of declaring a handler small, since it is assumed that signals are unusual, so that most declarations will never be invoked. In fact, entering the scope of a declared handler does not cause any instructions to be executed. Instead, the compiler generates some recognizable instructions which do nothing, and distributes them strategically in the program text where the signaller can find them.

When the signaller gets to a context which has no explicit handler, then, it examines the program text for in-line handlers. If one is found, its associated program text is located from the clues left by the compiler, and it is called in the usual way.

This scheme for handling signals has a good deal in common with the ON-condition facilities of PI/1. There are also a number of differences, however:

a) enabling a handler in MPL is a declaration, not an executable statement;
b) the program has much greater control of signal handling than in PI/1. In particular:
   . **any** and **reject** together allow decisions about signal handling to be made in a very flexible way;
   . if this isn't good enough, the user can write his own handler, rather than use **enabling**;
c) arguments can be passed with a signal, and results can be returned, as in the example above;
d) the zero time-cost for enabling a handler makes the facility very attractive to use.

8.2 Unwinding

Sometimes is is necessary to abandon a computation in mid-flight and restart from some earlier point. We call this operation *unwinding*. For example, when an error is detected in a compiler, the current state becomes useless and we want to make a fresh start, perhaps with the input advanced to the next statement of the source program. In general when this happens there is some collection of contexts which are no longer useful and should be destroyed. To

deal with this situation, we need:
a) a way of deciding which contexts should be destroyed;
b) a procedure for destroying each context in an orderly way;
c) some place to send control when the unwinding is complete.

If there are a lot of contexts around which are not related hierarchically, it is not at all clear who should be destroyed during unwinding. We therefore provide a standard procedure which does the right thing for nested procedure calls, and leave it to the programmer to write his own unwinder for more complex situations, using the operations of the next two paragraphs. The standard procedure is

```
unwind(from context, to context, signal),
```

and it destroys all the contexts encountered in propagating the signal between the two contexts, not including the end points. It is normally used in a handler, thus:

```
unwind(myself, myparent, mysignal).
```

The parent is passed to the handler when it is entered, along with the signal and signal argument.

Destroying a context is a two-step process. First it must be given a chance to put its house in order, i.e. to restore to a consistent state any non-local data structures which it may have been modifying. This is done by passing the signal **cleanup** to its handler. If the context wants to get control before being destroyed, it should enable this signal. When the handler returns, the context is destroyed, using the same facilities which would be used to destroy any other record. With the destroy operation in hand, we can write a skeletal program for unwind:

```
c := fromcontext;
for c := NextSignalHandler(c, signal) while c # tocontext do
destroy(c).
```

Finally, we consider how to continue the computation, for the special case in which the context doing the unwind is an in-line handler of the one which is to receive control. Since the handler knows about the program text of the destination in this case, it can simply set the destination's program counter to the proper value, and then exit by destroying itself, exactly like a buffer context (section 6.5). The **exit** statement in the previous "OutOfStorage" example is syntactic sugar for:

```
unwind(myself, myparent, mysignal);
myparent.programcounter := ExitfromComputation;
transfer(DC, myself, (myparent, NIL) ).
```

The Multics system [Organick] supports an unwind operation somewhat similar to what has just been described.

9. Control faults

The discussion of control faults in section 6.5 left two questions open:
. who gets notified when a control fault occurs?
. how is the notice served?

The first question is handled like the similar problem for signals. Each context has an *owner* outport which defines who should be notified. By default this is set to the creator of the context, but the user can establish any

relationships he likes by resetting it.

When a control fault occurs, it is simply converted into a signal called **controlfault** which is started off at the context specified by the owner outport of the faulter, and then propagates in the usual way. This makes it reasonably convenient for the owner to differentiate a fault from a normal exit. During startup, when control faults are expected, each handler will probably specify an exit to the next statement.

10. Conclusion

We have created an environment for describing global control disciplines, consisting of *contexts* within which execution takes place, and a **transfer** primitive for passing control from one context to another.

Records and classes were used to create contexts and to handle arguments. We showed how to define the binding function for names in a fairly general way, and described a strategy which allocates storage for contexts. We established conventions for passing arguments and return links which can accomodate a wide variety of control disciplines in a compatible way.

Ports were introduced as a non-hierarchical control discipline, and we saw how to initialize a collection of contexts connected by ports, how to handle linkage faults, and how to switch port connections so that several contexts can use a single port. We showed how to handle Algol-like *procedures* without any new primitives, and compatibly with ports.

Finally, we introduced *signals* as a control discipline for dealing with unusual events, described how to give the programmer complete control over signal propagation and how to implement signal handlers efficiently, and used the signal mechanism to provide for orderly retreat from untenable situations.

References

Balzer, R. M., "PORTS - A Method for Dynamic Interprogram Communication and Job Control," *Proc AFIPS Conf.* 39 (1971 SJCC)

Bensoussan, A. et. al., "The Multics Virtual Memory: Concepts and Design," *Comm ACM* 15, 5 (May 1972)

Bobrow, D. G. and Wegbreit, B., "A Model and Stack Implementation of Multiple Environments," *Comm. ACM* 16, 10 (Oct 1973)

Brinch Hansen, P., "The Nucleus of a Multiprogramming System," *Comm. ACM* 13, 4 (April 1970)

Conway, M. E., "Design of a Separable Transition-diagram Compiler," *Comm. ACM* 6, 7 (July 1963)

Dennis, J. B., "Programming Generality, Parallelism and Computer Architecture," *Proc. IFIP Congress 1968*, North-Holland Publishing Co., Amsterdam, 1969

Dijkstra, E. W., "Cooperating Sequential Processes," in *Programming Languages*, Genuys, ed., Academic Press, New York, 1967

Fisher, D. A., *Control Structures for Programming Languages*, Ph.D. Thesis, Carnegie-Mellon University, May 1970 (AD 708611)

Hoare, C. A. R. and Dahl, O-J., "Hierarchical Program Structures," in *Structured Programming*, Academic Press, New York, 1972

Knuth, D., *Fundamental Algorithms*, Addison Wesley, Reading, Mass., 1968, p. 425

Krutar, R. A., "Conversational Systems Programming," in *Sigplan Notices* 6, 12 (Dec 1971)

Lampson, B. W., "On Reliable and Extendable Operating Systems," in *The Fourth Generation*, Infotech, Maidenhead, Berks., 1971

McIlroy, M. D., "Coroutines: Semantics in Search of a Syntax," Bell Telephone Laboratories, Murray Hill, N.J., unpublished report

Organick, E. I., *The Multics System: An Examination of Its Structure*, MIT Press, Cambridge, Mass., 1972

Saltzer, J. H., *Traffic Control in a Multiplexed Computer System*, Sc.D. Thesis, MIT, 1966 (MAC TR-30)

Wang, A. and Dahl, O-J., "Coroutine Sequencing in a Block Structured Environment," *BIT* 11 (1971), p 425

Wegbreit, B., "The Treatment of Data Types in EL/1," *Comm. ACM* 17, 4 (April 1974)

Wirth, N., "The Programming Language Pascal," *Acta Informatica* 1, 1 (1971)

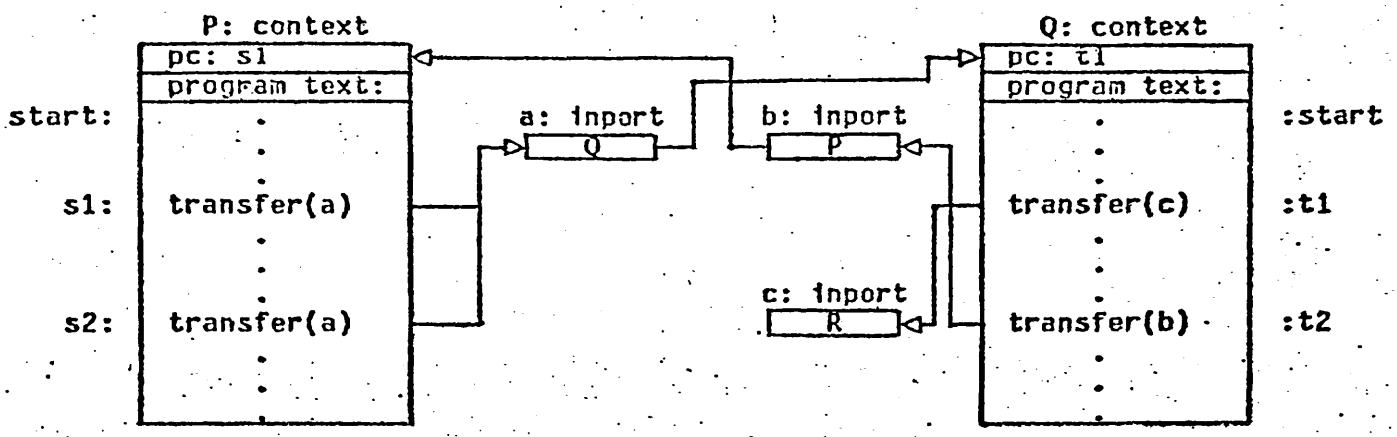


Figure 1: Coroutine linkages

Figura 2: Producer and consumer with reply

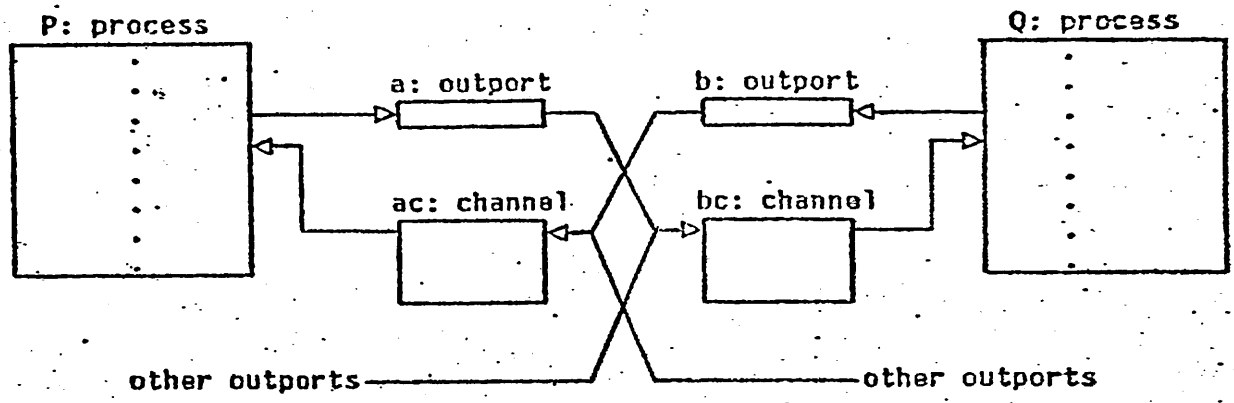


Figure 3: Symmetric communication between processes using message channels

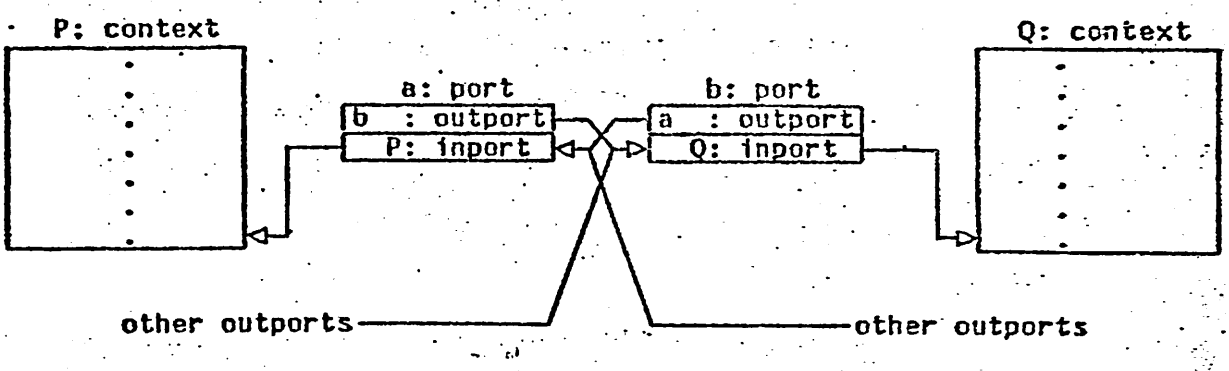
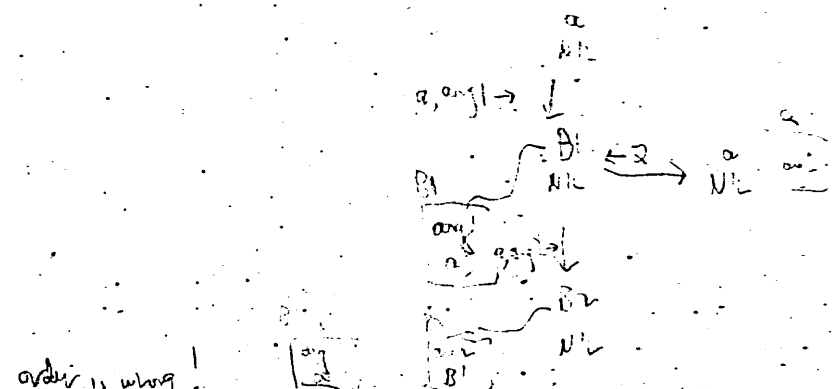
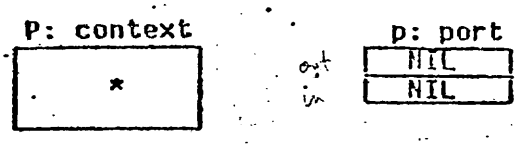
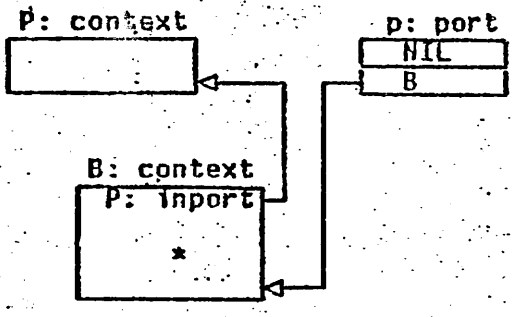


Figure 4: Symmetric communication between contexts using ports

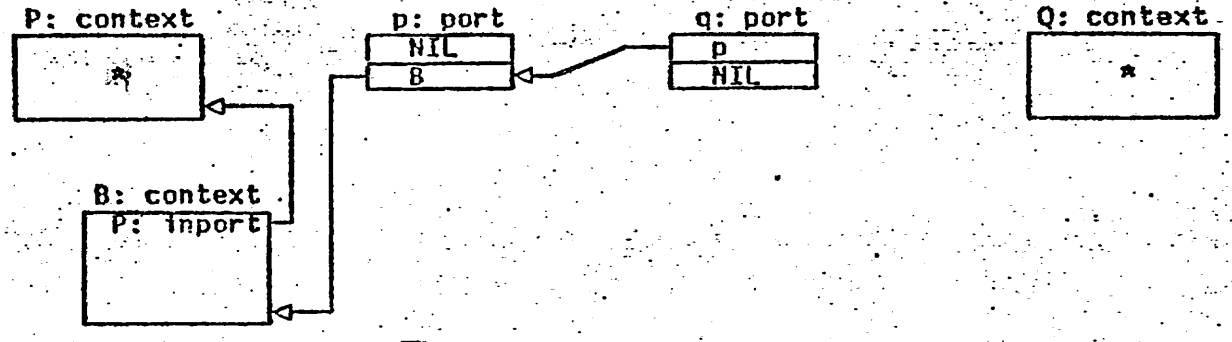




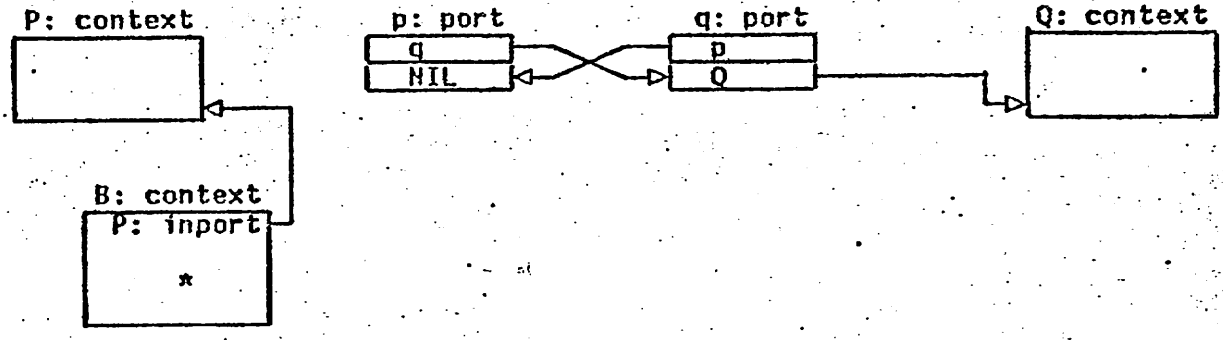
(a) Before P has attempted to use p



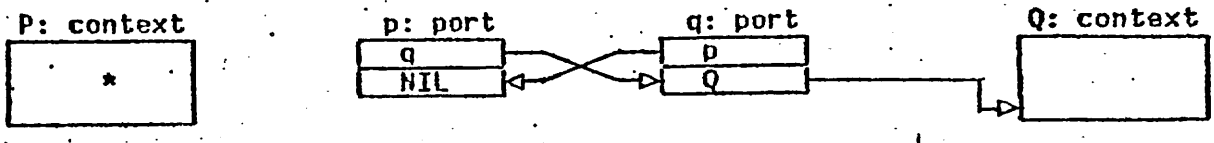
(b) P has transferred to p, and a buffer context B has been created



(c) Q has been created and started, and q has been connected to p



(d) Q has transferred through q and p to B, which is running



(e) B has destroyed itself and returned to P.
Initialization is complete

Figure 5: Initialization using a buffer context to handle control faults