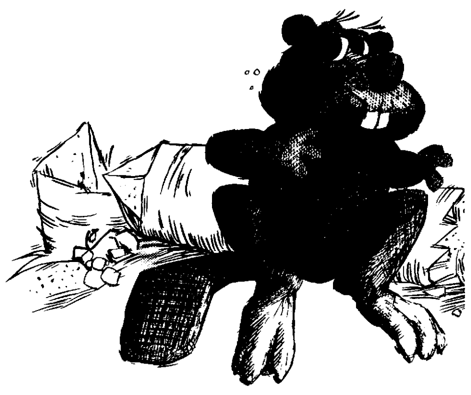

Concurrency and Forward Recovery in Atomic Actions

D.J. Taylor

CS-85-19

July, 1985

Concurrency and Forward Recovery in Atomic Actions

David J. Taylor

Department of Computer Science*
University of Waterloo
Waterloo, Ontario, Canada

ABSTRACT

Some difficulties and complexities in atomic actions occur only when the concept of atomic actions is extended to allow concurrency within atomic actions and to allow a single atomic action to execute at a number of different sites. Also, providing facilities for both forward and backward recovery presents problems not found in the more usual case of allowing only backward recovery. This paper presents an analysis of these problems and proposes a general structure for a solution. A syntax which might be used to specify this structure is also given and illustrated with examples. The practicality of the scheme is justified by sketching one possible implementation.

1. Introduction

Atomic actions, as a tool for controlling interactions when shared data is accessed, have been studied extensively. Usually, only backward error recovery is considered. Nesting of atomic actions may or may not be allowed, but usually if nested atomic actions are allowed, they must form a simple, sequential flow of control. Thus, a good understanding has been obtained of backward recovery and nested atomic actions. However, other issues related to atomic actions have not been studied extensively, and in consequence are not well understood. These include the use of forward error recovery, concurrency within atomic actions, and the use of atomic actions in distributed systems. A paper by Campbell and Randell [2] provides an approach to the problems just mentioned. This paper is an attempt to provide a simpler solution than the one developed by Campbell and Randell, without losing power and flexibility. It also attempts to go further than that paper, by presenting a possible programming language mechanism for implementing the scheme developed.

The definition of atomic action given by Anderson and Lee [1] will be used:

"The activity of a group of components constitutes an *atomic action* if there are no interactions between that group and the rest of the system for the duration of the activity."

*This work was performed while the author was a Visiting Research Scientist at the Computing Laboratory, University of Newcastle upon Tyne.

(For purposes of this definition, any use of an object by components both inside and outside the action constitutes an interaction, unless all accesses are read-only.) Other properties sometimes required of atomic actions, such as provisions for rollback on failure, are not required, although support for them as optional features is a requirement.

Some atomic actions are considered to be elementary, if a system designer or implementor does not need to consider their internal structure. Such elementary actions are usually operations provided by the underlying system. Their nature is not of direct concern here, rather it is simply assumed that a set of such actions exists and can be used to build more complex actions.

Atomic actions composed of collections of elementary actions may be *planned* or *spontaneous*. Spontaneous atomic actions, which arise because of a particular sequencing of elementary atomic actions, are not generally very useful, particularly if forward recovery is desired. Planned atomic actions need not be explicitly identified, but here it will be assumed that they are. It would be possible to allow only a single level of planned atomic actions, but modularity and hierarchical design seem to demand nesting of planned atomic actions.

In the simplest case, there will be a single, sequential control flow inside an atomic action, with inner atomic actions corresponding to portions of that control flow. In such a case, it is assumed that the system state initially observable inside an atomic action is the same as the system state observable from the containing action. In particular, an inner action will have access to the same main storage as the containing action.

If concurrency and distribution are allowed, then several inner actions may be executing simultaneously. These actions need to be protected from each other in the same way that independent atomic actions are protected. Thus, it is assumed that other than in simple, sequential cases such as described previously, the only sharing of data occurs through the same mechanisms used for sharing between independently executing actions. In particular, this means main storage will almost certainly be unshared. A fortunate side effect of this is that it minimises differences between distributed and non-distributed cases, since main storage sharing in a distributed case is likely to be impractical.

The remainder of this paper contains a more detailed discussion of the use of atomic actions containing concurrency. Section 2 presents a general description of the problems of concurrency and forward recovery in atomic actions, together with a proposed solution. Section 3 develops this solution into possible programming language mechanisms for specifying such actions. Section 4 contains two examples showing use of the mechanisms developed in Section 3. Section 5 sketches a possible implementation of the mechanisms. Finally, Section 6 provides a brief summary and suggests some possibilities for future work.

2. Choice of Concurrency

If an atomic action contains no concurrency, then the body of the action may be simply an ordinary, sequential program. Mechanisms for introducing controlled concurrency to sequential programs have been studied in the context of operating systems for many years. The issue addressed in this section is how best to add the concept of concurrency to the concept of a "sequential" atomic action. The section begins with a discussion of the basic issues of concurrency in atomic actions and then continues with consideration of alternative means of controlling such concurrency.

The execution of any atomic action must ultimately consist of the execution of some set of elementary atomic actions. Some of the elementary atomic actions may be grouped into a hierarchy of larger atomic actions. By ignoring all actions (elementary and non-elementary) which are contained within larger actions, we may obtain a partition of a given action, into a collection of elementary and non-elementary actions. These will be referred to as *subordinate actions* of the given action. As discussed in the preceding section, all non-elementary actions are assumed to be planned and explicitly identified.

For a partition of an atomic action, there must be a set of precedence constraints which determine the correct execution sequences of the subordinate atomic actions. For any correct execution sequence to exist, the precedence constraints must form a partial order, and hence there must be some total ordering of the subordinate actions to form a correct execution sequence. Thus, it is possible to execute the subordinate actions sequentially. Applying this recursively to any subordinate actions which are not elementary shows that all elementary actions can be executed sequentially to provide a correct execution of the original, given, atomic action.

The result in the preceding paragraph is hardly surprising, and simply shows that concurrency within an atomic action is not essential for correct execution. However, in many cases concurrency will be desirable. For example, an atomic action may be quite large, and consume significant resources. Allowing the use of concurrency, and hence of multiple processors, may provide a significant improvement in execution time. In a distributed environment, efficient execution may demand that parts of an atomic action proceed concurrently at a number of sites. As well, the algorithm used by an atomic action may have natural concurrency. Transforming the natural partial ordering of events into a consistent total order may significantly increase the difficulty of implementation and significantly reduce the clarity of the resulting program.

If concurrency is allowed, then a method must be chosen for organising the concurrency of subordinate atomic actions. One possibility is simply to specify that a given action consists of a set of subordinate actions, and give the precedence constraints between these actions explicitly. Then the implementation can run each subordinate action when its precedence constraints are satisfied. Other than the ordering implied by the precedence constraints, and competition for resources, the actions then run quite independently. This alternative will be designated HN (Hierarchical Network of actions).

Figure 1 shows an atomic action structured as a network of five subordinate actions. The figure is intended to show that subordinate actions 1 and 2 may be executed in parallel. When they have both finished, action 3 may execute. When action 3 has finished, actions 4 and 5 may execute. When both 4 and 5 have finished, the containing action is complete.

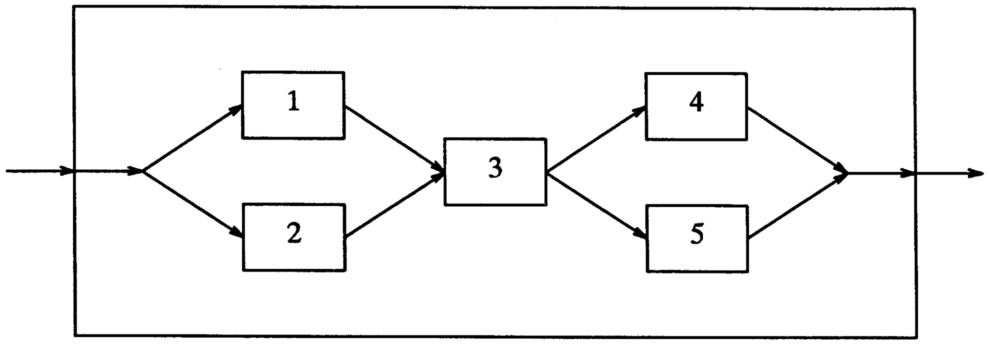

Figure 1. Hierarchical Network of Actions

Alternatively, an action can be structured as a (possibly changing) collection of processes, each of which executes a sequence of subordinate atomic actions. This structuring implies an ordering constraint between the actions which belong to a single process: additional constraints could be specified explicitly, or one of the standard operating system techniques for inter-process synchronisation could be used to enforce the additional constraints. This alternative will be designated PP (Parallel Processes).

Figure 2 shows an atomic action structured as two parallel processes. The parenthesised sections containing repeated digits represent subordinate actions executed by the two processes. Thus, the first process executes subordinate actions 1, 3, and 4, and the second process executes subordinate actions 2 and 5. The dashed arrows designate synchronisation between the processes: action 3 of the first process cannot start until action 2 of the second process has completed; action 5 cannot start until action 3 has completed; and the first process must wait for action 5 to complete before the process terminates the enclosing action. This example provides the same structure of subordinate actions as given in HN form in Figure 1.

Note that both alternatives can readily model a simple, sequential, atomic action as a special case. In HN, the precedence constraints between the subordinate actions can be specified implicitly by the ordinary control flow of the program. In PP, the action consists of a single process, with no additional precedence constraints required.

If each process, for PP, executes only a single atomic action, then the two cases become equivalent. Thus, a real distinction occurs only if a process does not execute exactly one plannned atomic action (such an action might, of course, contain an arbitrary structure of actions internally). Two difficulties occur if this is done: one involving forward recovery after an exception occurrence, the other involving resource allocation.

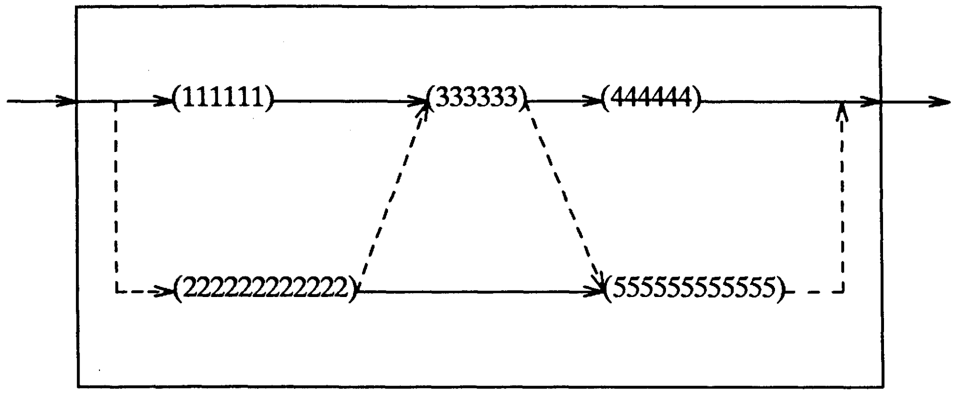

Figure 2. Atomic Actions Implemented as Parallel Processes

When an exception occurs, it is first necessary to complete all subordinate actions which are in progress, and then (if forward recovery is being used) to enter an exception handler. If each process executes a single atomic action, then it is reasonable to discard the process when the corresponding action completes. Thus, immediately before entering the exception handler, the state of the atomic action will simply consist of some number of completed subordinate actions and the state of the resources they have locked. It is then reasonable for the exception handler to be simply a new subordinate action.

If a process may execute several atomic actions sequentially, then it is not reasonable to discard the process when the atomic action it was executing completes. Thus, when the exception handler is to be entered, the state of the atomic action will include some number of processes which have each executed a sequence of subordinate actions and which might have continued executing further subordinate actions if an exception had not occurred.

It is argued in the Campbell and Randell paper that each of these processes must be involved in the forward recovery activity. That is, that each of them must enter a new atomic action, with the set of such actions constituting the exception handler. Unfortunately, it is difficult for any one process to obtain enough global information to handle the exception. (If the exception can be handled completely by the process executing the atomic action which signalled the exception, then the atomic actions should be restructured so that the exception is not signalled by a directly subordinate action.) If the exception truly does have global significance for the atomic action, then global state information is likely necessary to deal properly with it. Distributing recovery among several process may well be counter-productive, and grouping several actions to form a process may obscure the overall state of the action, which is largely determined by which actions have completed. Thus, allowing a process to execute several planned atomic actions adds significant complication to forward recovery.

The second difficulty involves the locking* of shared objects. Each atomic

*Although the problem is described here in terms of locking, the same problems exist if a timestamp mechanism is used.

action must lock all shared objects which it uses, in order to achieve atomicity. If several actions are concatenated to form a process, should locks then persist from one action to the next? If the answer is yes, then the whole execution of the process will be atomic, so the process will not really consist of a sequence of separate atomic actions, when viewed by other processes in the same parent action. If the answer is no, then each successive action cannot rely on any shared data being unchanged from the previous action. If this is the case, then there seems little point in grouping the actions to form a process. Several actions might be assigned to the same process by the implementation, to avoid operating system overheads of process creation and destruction, but it seems undesirable to make such structuring visible at the user level.

The only possible advantage of grouping several actions to form a process then seems to be that unshared data, belonging to the process, can be passed from one action in the sequence to the next. It is possible that there are situations in which this might be useful. However, the information which can be passed in this way is quite limited, since any data derived from shared data could become invalid due to changes made to the shared data by other actions.

Thus, it appears that allowing a sequence of actions to be grouped as a process presents some significant problems, and does not have significant compensating advantages. Therefore, as an initial model of concurrency, each atomic action is implicitly associated with a separate process. For practicality, some refinement is clearly required. Thus, an atomic action which contains a strictly sequential collection of subordinate actions can have a single process associated with all subordinate actions. Similarly, if an action has some purely sequential intervals of execution, the actions associated with those intervals can also have a single process associated with them. One might associate a separate process with each such interval, but it may be useful to assume that it is the same process executing in each such interval. (Note that the major significance of allowing explicit grouping of actions into processes is that such a sequence can then be coded in the manner usual for sequential programs, and that local data can be passed from one action to the next.)

When concurrent execution of several actions is desired, it is necessary to specify the required actions and their precedence. There are two basic possibilities: initial specification of the complete set of actions and incremental specification by continuing execution of the original process. The first alternative requires a static specification of the subordinate actions and their precedence, the second alternative allows a dynamic specification.

Since continued execution of the original process apparently contradicts the above argument about grouping actions into processes, the static specification might appear to be preferable. Unfortunately, a static specification cannot deal easily with complex cases that are likely to arise. In particular, the set of actions to be executed may well be data dependent. While a sufficiently powerful static specification could deal with such cases, it seems unfortunate to require the use of a separate, powerful technique when conventional programming languages can readily deal with such problems. Dynamic specification, through continued execution of the original process, allows a conventional programming language to be used. (In fact, the text of the appropriate part of the program constitutes a static specification of the actions and their precedence, but this is not of practical significance.)

The potential problems of this scheme can be avoided in the following way. The activities of a process must constitute a sequence of atomic actions. Some of these actions may be explicitly identified actions, consisting of sequences of elementary atomic actions, the others will be elementary atomic actions. Creation of a process requires creation of a corresponding atomic action. The process is thus initially alone inside its own action, and can execute an arbitrary sequence of elementary and non-elementary actions. However, when it requests concurrent execution of an action, it becomes restricted to executing elementary actions and those elementary actions must not access any shared data. (Moss [8] also prevents access to shared data in these circumstances, apparently for the same reasons motivating the restriction here.) It can request execution of further concurrent actions, and can wait for termination of actions. It thus takes on the role of a "traffic director" of the collection of actions executing concurrently. Once all concurrent actions have terminated, the process may resume unconstrained activity. All of this takes place inside whatever explicitly identified action is being executed by the process when it first requests concurrent execution of another action. This may be the outer action of the process, or a (directly or indirectly) subordinate action.

The designers of LOCUS [9] have adopted a different solution to the locking problems caused by concurrency within an atomic action. Rather than causing a process, effectively, to give up all locks when it creates a concurrent action, the process continues to hold locks, and these locks cannot be acquired by subordinate actions. If access by a subordinate action is to be allowed, the parent action must release the lock, causing it to become a *retained* lock. An action may lock a resource only if no other action *holds* the lock and all actions *retaining* the lock are ancestors. Thus, an action which invokes a subordinate action must release locks before invoking a subordinate action and reacquire any still needed after the subordinate action has completed. This must be done for all locks unless enough is known about the implementation of the subordinate action to determine what resources it will lock. This solution is more complex and, if proper independence is maintained between the design of actions, will cause much unnecessary releasing and reacquiring of locks.

From the above, it appears that concurrency within atomic actions can best be achieved by making the creation of new processes correspond to the creation of new actions, with the lifetime of the process being the same as the lifetime of the action. The creation and synchronisation of such concurrent actions can be conveniently managed by allowing the original process in the parent action to function as a "traffic director," but not allowing it to perform any other activity while concurrent actions exist.

Figure 3 shows such a "traffic director" organisation. The original process executes action 3 directly, but creates new processes to execute the remaining four actions. The original process also ensures that action 3 does not begin until actions 1 and 2 have completed, and that actions 4 and 5 do not begin until action 3 has completed. This example provides the same structure of subordinate actions as given in Figures 1 and 2.

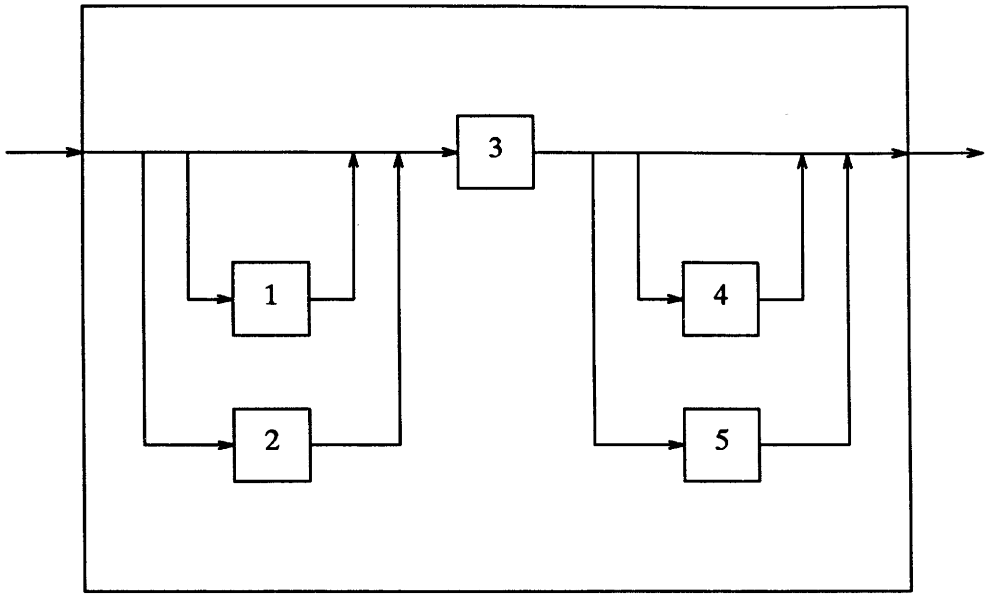

Figure 3. HN Implementation using PP

It may be useful to compare the scheme developed in this section with other proposals and implementations of concurrency in atomic actions. The mechanism proposed by Moss [8] has very similar concurrency features. ARGUS [6] allows the concurrent invocation of several subordinate actions. The invoking action does not continue until all subordinate actions have completed. Thus, only series-parallel process flow graphs can be created. An example in which this restriction is undesirable is given in Section 4. LOCUS [9] allows several processes to execute inside the same atomic action, but it is not clear how interactions between those processes are controlled. When a process invokes a subordinate atomic action, locking is handled as described above. Campbell and Randell [2] also allow "unconstrained" parallel processes inside an atomic action and allow several such processes to enter a subordinate action together. The extreme flexibility of this scheme causes various complications with locking, exception handling, and entering and leaving atomic actions.

3. A Proposal for Atomic Actions

This section provides some detail concerning the way in which concurrent atomic actions might be specified. The basic framework for concurrency developed in the preceding section will be assumed. To provide a concrete environment for the proposal, a UNIX* system and the C language will be assumed. The concurrency techniques are influenced somewhat by this choice, but the proposal attempts to be as independent as possible of both language and

*UNIX is a trademark of Bell Laboratories operating system.

To support the required concurrency and recovery techniques, the following must be provided.

1) Sequential invocation of (subordinate) atomic actions.
2) Concurrent invocation of subordinate atomic actions.
3) Specification of precedence among subordinate concurrent atomic actions.
4) Specification of type of recovery for each exception, including handlers for forward recovery and alternates for backward recovery.
5) Specification of a resolution technique for multiple, concurrent exceptions.

There is no apparent reason for using different techniques for specifying and invoking top-level atomic actions and subordinate (sequential) atomic actions. Commitment of an action must be handled quite differently in the two cases, but this does not imply that they must be specified differently. Using a uniform technique for specifying an atomic action also has the virtue that a single atomic action may be used both as a top-level action and as a subordinate action. Similarly, an action to be invoked concurrently can be specified in the same way as an action to be invoked sequentially. Only the actual invocation needs to be different.

It seems reasonable to use the method proposed by other authors [6, 7], and specify an atomic action as a kind of function. Then, an atomic action invocation looks like an ordinary function call, and the function declaration looks like an ordinary declaration, except for the inclusion of a special keyword. Selecting *atomic* as the keyword, an example of such a declaration is

```
atomic int func(a) { /* body */ }
```

The creation of a subordinate concurrent atomic action requires the use of an additional process. (It is simplest to imagine the creation of a new process, but an implementation might simply pass a message to an existing process.) It appears that the best way of defining and activating such a subordinate atomic action is to proceed exactly as in the case of a subordinate (sequential) atomic action, except for the inclusion of a "parallel" specification at the point of invocation. ("Concurrent" is only two letters longer than "parallel", but it just seems too long.) Thus, one would request sequential invocation of the above atomic action by

```
x = func(27);
```

and concurrent invocation by

```
id = parallel(func(27));
```

In the latter case, the result returned cannot be the actual function result, instead it is an identifier which can be used to refer to the concurrent atomic action. Alternatively, the invoking program might provide an identifier and specify it as part of the concurrent invocation, but this seems slightly more complex with no compensating advantages.

If the result of the function is required, then a construct which both waits for completion of the concurrent activity and retrieves the result seems appropriate, for example

```
x = result(id);
```

Specifying precedence among subordinate atomic actions is potentially quite difficult. One approach would be to let the containing atomic action handle all synchronisation, by waiting for appropriate actions to terminate before invoking a subsequent action. This could become quite complex and would require a primitive allowing a wait for one of a number of subordinate actions to terminate. As well, in a distributed system, it would result in all synchronisation being performed at the "home" site, which may be inefficient. A simple technique is to allow the explicit specification that a subordinate concurrent action cannot be started until the completion of certain other subordinate actions. For example,

```
id = parallel(func(27), after(id1, id2));
```

where *id1* and *id2* are atomic action identifiers returned by previous concurrent invocations. (This automatically results in a set of precedence relations which form a partial ordering.)

As well as providing for the invocation of one subordinate action on the completion of other actions, it may be desirable to pass results directly from one subordinate action to another. This could be accomplished as

```
id = parallel(func(result(id1)), after(id2));
```

in which the dependence on the result from *id1* implies that this concurrent action cannot begin until *id1* has completed.

Some restriction must be placed on the way in which the result is passed on from an atomic action, since otherwise all results would have to be retained by the underlying system until the parent action terminated. There are a variety of possibilities, including the provision of a means for stating explicitly that a particular result is no longer required. A simpler, although more restrictive, scheme is proposed, namely, that a result can be passed to only one destination. That is, a subordinate atomic action may be referenced only once by the execution of a "result()" clause. (Any subsequent use of "result()" with that atomic action identifier will raise an exception.) This does not introduce a real restriction, since a result may be accepted by the parent action and then passed on to many subordinate actions. (This restriction may be considered to be, ultimately, a consequence of using a dynamic specification of concurrency. A static specification, of suitably restricted power, would allow the atomic action implementation to deduce exactly what data flows would be required, and hence how long to retain the result of an atomic action.) If an action produces no result, its type must be specified as *void* so that no attempt will be made to retain the nonexistent result.

If an action is to be executed remotely, then the concurrent invocation may specify a site, as id = parallel(func(result(id1)), after(id2), at("node5"));

If no site is specified in the invocation, such an action will be executed in a default location. This location is normally the site of the invoking action, as assumed in the examples above. If it is desired to change the default location, to avoid specifying *at* in the action invocation, *default_loc* may be used, as

default_loc(func, "node5");

If this was executed previously, then 'at("node5")' would not be needed in the immediately preceding example. The binding set up by *default_loc* only applies to the atomic action which executes it. Thus, if a subordinate action is invoked by this action, the default location of func will intially be local, even though it is "node5" in the invoking action.

The preceding part of this section has described the definition and invocation of atomic actions without any reference to exception handling. In the following, exception handling facilities for atomic actions are described, but it is not appropriate to discuss exception handling itself in depth. As stated earlier, facilities for both forward and backward recovery are desired. For forward recovery the code to be executed when an exception occurs is simply referred to as a *handler*. For backward recovery, the code executed when an exception occurs (after state restoration has taken place) is referred to as an *alternate*, using the terminology established for recovery blocks [5]. A set of exception which can be handled, and the way they are to be handled, is called a *handling context*.

A method is required for associating exception handling contexts with segments of the normal control flow. The association can be static or dynamic, but a dynamic association, as with the ON-units of PL/I [10] is now generally considered undesirable. A static association might be made at a number of levels: operation, statement, function, or atomic action. Goodenough proposed a scheme allowing all these possibilities [4]. Cristian proposed a scheme which does not allow a handling context to be associated with an operation, but allows the other cases [3]. However, in order to allow backward recovery, it must be possible to restore the state which existed when the handling context was established. This implies that the control flow associated with a handling context must have the properties of an atomic action. Thus, it is proposed to make handling contexts coincide with atomic actions, rather than portions of actions. Exception handlers and alternates must then also be atomic actions, since they will effectively replace an atomic action which raises an exception. This simply means that an action must be entered whenever a new handling context is desired.

When a handler or alternate is invoked, it will have access to all the information accessible to the action which raised the exception, with the obvious qualification that, if backward recovery is used, state restoration will have taken place.

Thus, it seems natural to specify the handling of exceptions as part of the function header for an atomic function. For each exception it is first necessary to specify whether forward or backward recovery is desired. For forward recovery a handler must also be specified. For backward recovery, normally an alternate routine will be specified. If no alternate routine is available, implying that the action will signal failure after state restoration, this must be indicated. For convenience, there should also be a way of specifying a method for handling all exceptions not explicitly named. For example,

```
atomic int func(a)
exception  failure     backward  alt_1(),
           bad_assert  forward   handle_1(),
           default     backward  alt_2();
{/* body */}
```

If an alternate is to be executed remotely, this must also be included in the specification. (A remotely executed handler does not appear to be useful, and thus a mechanism is provided only for remote execution of alternates.) Thus, if alt_1 was to be executed at "node3" and the other alternate executed locally, the example would become

```
atomic int func(a)
exception  failure     backward  alt_1()     at("node3"),
           bad_assert  forward   handle_1(),
           default     backward  alt_2();
{/* body */}
```

The expression used to specify the location for execution of an alternate or handler should be evaluated when the atomic action is invoked, allowing the implementation to check in advance whether there is such an atomic action available at the indicated site. Thus, the allowed expressions will probably be very restricted, according to the scope rules of the underlying language.

As well, a mechanism is required for resolving multiple simultaneous exceptions. When an exception is detected, it should be handled only after all subordinate concurrent actions have completed. (Aborting such actions is discussed briefly below.) As these actions complete, some may signal exceptions, producing multiple exceptions before recovery processing can begin. The exception tree proposed by Campbell and Randell is an appropriate mechanism for handling this situation. (An *exception tree* is simply a tree whose nodes represent exceptions. When several exceptions occur, the root of the smallest subtree containing all those exceptions is taken as the one exception which should be raised.) The tree can be specified by giving for each exception its parent in the exception tree. It seems reasonable to assume that exception trees have the form suggested in Campbell and Randell, with the universal exception as the root and *other_exceptions* directly below the root. Then, this part of the tree need not be given explicitly. As well, all otherwise unspecified exceptions can be taken as immediate descendants of *other_exceptions*. Thus, an atomic function definition including an exception tree might be

```
atomic int func(a)
exception  failure     backward  alt_1()     at("node3"),
           bad_assert  forward   handle_1() sub_to failure,
           default     backward  alt_2();
{/* body */}
```

This specifies that *bad_assert* occurring by itself will be handled by the forward recovery routine *handle_1*; *failure* by itself or occurring with *bad_assert* will be handled by backward recovery into routine *alt_1*; finally, any other exception, by itself or combined with one or both of these two will be handled by backward recovery into routine *alt_2*. (Note that the default handling is associated with *other_exceptions* since it is not a named exception.)

This has added sufficiently to the standard declaration of a function that is appropriate to specify formally the syntax intended.

```
<func> ::= atomic <type> <name>(<param_list>)
        [<param_decls>]
        [exception <except> {, <except>}* ;]
        <func_body>

<except> ::= <exc_name>
          {forward <call> | backward <alternate>}
          [sub_to <name>]

<exc_name> ::= <name> | default

<alternate> ::= <call> [at(<expr>)] | no_alternate

<call> ::= <name>(<params>)
```

The various non-terminals which are not defined are to be understood with their usual meaning in the underlying programming language (in this case, C).

The above could be expanded to allow convenient handling of special cases, such as backward recovery with no alternate for all exceptions, forward recovery consisting only of signalling an exception, and so on. It seems unwise to complicate things further until it becomes clear which special cases really deserve such treatment.

As mentioned above, it may be desirable to abort ongoing subordinate atomic actions when an exception is raised, rather than waiting for them to complete normally. This can be accomplished by raising a specified exception, *abort,* in all active subordinate atomic actions which can handle such an exception. (This is the proposal given by Campbell and Randell.) It seems unreasonable to allow *abort* to be included in the default case, so we require explicit declaration of a handler for *abort*. Thus, to allow the example function to be aborted, we would specify

```
atomic int func(a)
exception abort      backward no_alternate,
          failure    backward alt_1()     at("node3")    sub_to abort,
          bad_assert forward  handle_1()  sub_to failure,
          default    backward alt_2()     sub_to abort;
{ /* body */ }
```

This specification causes any exception occurring along with *abort* to be ignored, which seems appropriate since *abort* simply causes state restoration followed immediately by termination. However, if some exceptions required other handling, even in the presence of an *abort* exception, this could be specified.

Since a tree is being used, rather than a lattice, there is not complete freedom in making such a specification, but it seems unlikely that this will be a practical problem. Likely, most abortable actions will resemble the example.

The effect of raising an exception in an atomic action with active subordinate concurrent actions will thus be as follows. An attempt will be made to raise *abort* in each active subordinate action. This may in turn result in *abort* being raised in more deeply nested actions. (The propagation of *abort* downward is thus automatic, not requiring any special mechanism.) Once the exception has been propagated to all appropriate subordinate actions, it is only necessary to wait for all subordinate actions to terminate. Once all actions have terminated, the appropriate recovery action can be taken.

It would be possible to propagate *abort* more thoroughly than suggested above, by raising *abort* in all directly or indirectly subordinate actions which have an *abort* handler. This would have the effect of raising *abort* in abortable actions subordinate to non-abortable actions, which seems quite unreasonable. A logical extension of this idea would be to continue observing the non-abortable action and every time it invoked a new subordinate action, try to abort that. This strategy could result in a non-abortable action being completely unable to make any further progress, in spite of the non-abortable specification. It seems to make much more sense to regard a non-abortable action as one which has declared that it does not want to be interfered with as a result of any exceptions raised outside itself.

4. Two Examples

This section contains two examples which illustrate the use of atomic actions containing the type of concurrency described in the last two sections. For reasons of space, both examples are quite simple. The first one shows the use of atomic actions to allow well-controlled concurrency in a distributed activity. The second shows use of the exception handling mechanism.

The first example is a file transfer facility. It is reasonable to imagine a file transfer as an atomic action, since reading of the sending file must interact with updates to that file in a manner which will yield an internally-consistent version of the file, and writing the receiving file must interact with other accesses to that file so that other activities observe an internally-consistent version of the file.

For purposes of this example, the following are assumed. All facilities of OSI levels 1-6 are provided by the underlying system [11], the file transfer running as an application program at level 7. The underlying system does not allow writing or reading of files at remote sites: the action performing I/O and the file must be at the same site. Because reading and writing files requires locking of either files or records, and in order to avoid a centralised lock facility, it seems essential to handle locking at the site holding a file. It would be possible to accept lock requests from remote sites, but a great deal of information about the atomic action would have to be transmitted in order to handle locking properly. While this is not impossible, this level of complication is avoided by omitting the facility to lock and use remote objects except through an action at that site.

In order to allow the presentation layer of OSI to perform any necessary translation or restructuring of records, we will need to perform the transfer on a record by record basis. Thus, an obvious possibility is to create a subordinate atomic action for each record transfer, and within each such action, create further subordinate actions to perform the reading of the record on the sending system, and the writing of the record on the receiving system. To guarantee that records are added to the receiving file in the proper order, all "record transfer" actions must be performed sequentially, and clearly the write action must follow the read action within each transfer action. Thus, a strictly sequential execution will be imposed, allowing no concurrency between the sending and receiving sites.

Fortunately, this is not the only organisation possible, given the facilities described in the previous section. Instead of taking "transfer record" as the first level subordinate action, we can structure the file transfer directly as a collection of reads and writes. Then, it is possible to specify that the reads must occur sequentially, the writes must occur sequentially, and each write must follow the corresponding read. This will provide proper behaviour, and also allow the two sites involved to perform reading and writing concurrently.

The following assumes that the *file_transfer* action is being executed on the site of the sending file. If this is not naturally the case, then the effect can be achieved by creating another action which locates the sending file and then invokes *file_transfer* at the appropriate site. For reasons of brevity, such an action is not shown.

```
atomic file_transfer(in, out, num_recs)
char *in, *out;
int num_recs;
{
    int fd_in, fd_out, i, id1, id2;
    char *loc_out;
    loc_out = where(out);
    default_loc(write_rec, loc_out);
    fd_in = open_in(in);
    fd_out = result(parallel(open_out(out), at(loc_out)));
    id1 = parallel(read_rec(fd_in));
    id2 = parallel(write_rec(fd_out, result(id1)));
    for (i = 1; i < num_recs; ++i) {
        id1 = parallel(read_rec(fd_in), after(id1));
        id2 = parallel(write_rec(fd_out, result(id1)), after(id2));
    }
}
```

```
atomic char* read_rec(fd)
int fd;
{
    char *buffer;
    buffer = malloc(RECSIZE);
    read(fd, buffer);
    return(buffer);
}

atomic void write_rec(fd, buffer)
char *buffer;
{
    write(fd, buffer);
}
```

(The functions which open the two files have been omitted.)

The function *where* is assumed to return a string giving the site at which a file is located. The functions *read* and *write,* although similar to the standard UNIX functions, are assumed to be record-oriented for purposes of this example.

Note that various complications can be readily handled by the above structure. For example, it might be desired to split the file between two receiving sites, based on record contents. Then, it is necessary to retain the atomic action identifier for the last write action sent to each site. After testing to determine where to send a record, a write action can be invoked which is specified as "after" the last write at the appropriate site. This can be accomplished by a minor modification to the above code and will produce as much concurrency as is consistent with correct execution.

Also note that a series-parallel process flow graph, with reads and writes as the elementary actions cannot achieve the concurrent activity allowed by the above scheme.

Whether the above is a reasonable way to structure a file transfer depends on the cost of using atomic actions. If the cost is large compared to disk I/O and network communication, then an efficient file transfer might have to be implemented below the atomic action interface available to the user.

The second example is a skeleton, rather than a complete example. Assume we want to extract certain summary data from a file which is available locally. The same summary data can be extracted from a remote file (possibly a copy, possibly a quite different file). We adopt the strategy of attempting the operation locally and trying it remotely only if the local attempt fails. There is a distinguished exception, indicating an apparent structural error in the file. If this is detected, we attempt to repair the error and retry the operation locally. All other exceptions cause the remote execution to be tried immediately.

While this example is much like a recovery block scheme, the special handling of the structural error exception is not provided for in a standard recovery block, which handles all exceptions in the same way.

The following code, leaving the *do_processing* and *do_correction* functions unspecified, will accomplish exception handling according to the above strategy.

```
atomic extract_info()
exception default backward no_alternate;
{
    char *str;
    str = result(parallel(alt_1());
    printf("%s\n", str);
}

atomic char* alt_1()
exception  struct_error    backward  correct(),
           default         backward  remote_1()      at("remote");
{
    return(do_processing());
}

atomic char* correct()
exception default backward remote_1() at("remote");
{
    do_correction();
    return(do_processing());
}
```

The declaration of *remote_1* at *remote* is not shown. If the remote file is a copy of the local file, *remote_1* could be identical to *alt_1* except for the default exception handling. However, it could be a completely different action, and would have to be if a differently-organised file were used at the remote site.

Note that the invocation of *alt_1* is required to be a concurrent invocation because of a combination of two conditions. One is that an action can be executed remotely only as a concurrent action. The other is that an action which is executed locally but has a remote handler or alternate must be treated as a remote action.

5. Implementation

This section sketches an implementation for the concurrent atomic actions described in Section 3. It is intended only to provide a better understanding of the mechanisms described there, and to demonstrate the feasibility of implementing such a scheme. Thus, no attempt is made to describe details of data structures or messages used. As well, no attempt is made to describe interactions with the operating system. Preferably, several atomic actions will be bound together as one executable module, so the overhead of invoking an action will be only that of a function call plus a small amount of bookkeeping. Similarly, the mechanism for binding an atomic action name, at a remote site, to a particular function in a particular executable module, is left unspecified.

The implementation described here uses locks for synchronisation, although timestamps could be used. The mechanism proposed by Moss [8] does not allow forward recovery, but the details of locking are largely unaffected by that restriction. Thus, his approach could be used to supply some of the detail omitted below.

Details of recording data for backward recovery and performing state restoration are also omitted. A checkpointing mechanism could conceivably be used, but a form of recursive cache [5] is assumed. Various forms of cacheing have been proposed [1, Chapter 7], but it is not necessary for purposes of this description to specify a particular one.

On invocation of an atomic action, there are essentially three cases to consider: simple invocation of an atomic action, invocation of a subordinate concurrent action at the same site, and invocation of a subordinate concurrent action at a different site. In the first case, it is only necessary to assign a (local) identifier for the action, save relevant information such as the exception tree and how exceptions are to be handled, and establish a new level of cacheing, if appropriate. If this action requires backward recovery, a new cache level is required. If this action does not require backward recovery, but an enclosing action does, the previous cache level can continue to be used. If neither action requires cacheing, none need be done.

When an outer action is created, the steps are as above. The only essential difference is that a globally unique identifier must be generated. Any standard technique, such as concatenating the site number and current time, can be used.

On invocation of a subordinate concurrent action, it is necessary to check precedence constraints, and hold the action pending if some are unsatisfied. Once all are satisfied, the action may be started concurrently with the parent action, performing the steps described above in order to initialise the new action.

If the subordinate action is to be at a different site, it will be held at this site until all precedence constraints have been satisfied, other than those at the destination site. Once these constraints are satisfied, a message may be sent to the destination site, specifying (1) the complete list of action identifiers, from the outermost action to the new action, (2) precedence constraints to be satisfied at the destination site, and (3) the function to be invoked, and its parameters. The remote site is also added to the *remote site list* for the action.

At the remote site, an environment for the action must be established. This involves recording the identity of the new action and any of its ancestor actions which are not already known at the site. In this way, a subtree of the complete tree of actions will be constructed at the site. It will contain all actions executing at the site, and all of their ancestors. It is necessary to maintain information concerning the ancestors so that when an action terminates, the locks it holds and information it has cached can be properly retained.

Once the environment for the action has been established, precedence constraints relative to other actions at this site can be checked. Then, the action is either held pending completion of other actions, or is started as specified above.

When a subordinate action terminates, it is necessary to dispose appropriately of the locks it holds and any data which may have been cached for it. The locks are simply merged into the set of locks held by its parent action. If a new cache level was established for this action, the cache data must now be merged with the cache level of the parent, unless the parent was not performing cacheing, in which case the cache can simply be discarded. Then, a message is sent to each site on the remote site list for this action. At those sites, the message causes the same actions to be performed: the lock list is merged into the lock list of the parent and the cache data for this action is merged into the cache of the parent action or discarded.

When a concurrent action terminates, first all of the above steps must be taken. Then, the parent action must be notified of termination, which may involve sending a message to another site. This message includes a copy of the remote site list for the terminating action.

When notification is received that a subordinate concurrent action has terminated, the following steps are taken. The received remote site list is merged into the remote site list for this action. Precedence constraints of waiting actions are checked. This may result in an action becoming ready, in which case it is started as described above. It may also result in all constraints for a remote action being satisfied, except for those at the receiving site, in which case it may be sent to the remote site, as described above.

The rules for locking are the same as those in most systems supporting nested atomic actions. An object will be locked for shared use if all exclusive use locks are held by ancestor actions. An object will be locked for exclusive use if all locks are held by ancestor actions. Note that this simple locking rule is only possible because an action effectively gives up all its locks when it creates a subordinate concurrent action. To enforce this rule, any lock request from an action which has active subordinate actions is treated as an error. (It is assumed that any access to a shared object results in testing whether the object is locked, and causes a lock request if it is not.)

When a top level action terminates, all changes made by the action must be made permanent, and this process must be carried out atomically. For this purpose, a two-phase commit protocol may be used. The sites needing to participate will all be indicated in the remote site list for the top level action. Then, all locks may be released, and all record of the atomic action disposed of.

When an exception is raised, any active subordinate concurrent actions must be completed before the exception can be dealt with. As described in Section 3, the termination of such actions may be expedited by raising *abort* in any which have declared an *abort* handler. When all subordinate concurrent actions have terminated, there may be a set of several exceptions, including the one first raised. These exceptions are processed using the exception tree to determine which single exception corresponds to the entire set. Then, appropriate steps are taken to handle that exception.

If backward recovery is specified, the state at entry to the action is restored using the cache, then the cache and all locks are discarded. Next, a message is sent to each site on the remote site list to cause the same state restoration and discarding of locks to take place there. Finally, the alternate routine is invoked.

This alternate routine effectively replaces the original atomic action. Although it may be convenient to think of the handler invocation as a recursion, there is no possibility of return other than to the parent action, so no data concerning the action raising the exception need be retained. If there is no alternate routine, then *failure* is raised in the parent action, instead of invoking the alternate.

If forward recovery is specified, then the handler is simply invoked. As in the case of backward recovery, the handler effectively replaces the original atomic action.

6. Conclusions and Further Work

This paper has described methods of dealing with the difficulties which arise when concurrency and forward recovery are allowed in atomic actions. First, basic issues of concurrency were discussed, then a proposal for specifying concurrent atomic actions, allowing forward recovery, was presented. The suitability of this proposal was then argued through examples of its use and a sketch of a possible implementation.

Clearly, much more work remains to be done in this area. A number of the other systems for concurrent atomic actions, mentioned in this paper, have been implemented. To obtain further insight into practical difficulties, and to provide a fair basis for comparison, the scheme described here should be implemented. As well, the details of the programming language mechanisms presented are based on the C language. To a certain extent they are also based on the concurrency mechanisms customarily used in C, as provided by the UNIX operating system. While this is appropriate for purposes of the initial implementation, which will likely be in the UNIX environment, the effects of choosing another language or operating system should be investigated. Ideally, one would like an atomic action mechanism which was independent of implementation language and operating system, but it is not at all clear how this could be achieved.

Acknowledgements

This work was supported by the Natural Sciences and Engineering Research Council of Canada under grant A3078, and the Science and Engineering Council of the United Kingdom through a Senior Visiting Fellowship. The development of this paper has benefited from numerous discussions with Brian Randell, Jay Black, and Santosh Shrivastava.

References

1. T. Anderson and P. A. Lee, *Fault Tolerance: Principles and Practice,* Prentice-Hall, Englewood Cliffs, N.J. (1981).
2. R. H. Campbell and B. Randell, Error recovery in asynchronous systems, UIUCDCS-R-83-1148, Department of Computer Science, University of Illinois at Urbana-Champaign (1983).
3. F. Cristian, Exception handling and software fault tolerance, *IEEE Transactions on Computers* **31**(6) pp. 531-540 (June 1982).
4. J. B. Goodenough, Exception handling: Issues and a proposed notation, *Communications of the ACM* **18**(12) pp. 683-696 (December 1975).
5. J. J. Horning, H. C. Lauer, P. M. Melliar-Smith, and B. Randell, A program structure for error detection and recovery, pp. 171-187 in *Lecture Notes in Computer Science*, ed. E. Gelenbe and C. Kaiser, Springer-Verlag, Berlin (1974).
6. B. Liskov and R. Scheifler, Guardians and actions: Linguistic support for robust, distributed programs, *ACM Transactions on Programming Languages and Systems* **5**(3) pp. 381-404 (July 1983).
7. D. B. Lomet, Process synchronization, communication and recovery using atomic actions, *SIGPLAN Notices* **12**(3) pp. 128-137 (March 1977).
8. J. E. B. Moss, Nested transactions and reliable distributed computing, *Proceedings, Second Symposium on Reliability in Distributed Software and Database Systems*, pp. 33-39 (July 19-21, 1982).
9. E. T. Mueller, J. D. Moore, and G. J. Popek, A nested transaction mechanism for LOCUS, *Proceedings of the Ninth ACM Symposium on Operating System Principles*, pp. 71-89 (October 10-13, 1983).
10. J. M. Noble, The control of exceptional conditions in PL/I object programs, *Proceedings of IFIP 68*, pp. C78-C83 North Holland, (1969).
11. H. Zimmermann, OSI reference model--The ISO model of architecture for open systems intereconnection, *IEEE Transactions on Communications* **COM-28**(4) pp. 425-432 (April 1980).