PROCESS STRUCTURING, SYNCHRONIZATION, AND RECOVERY USING ATOMIC ACTIONS

D. B. Lomet
IBM T.J. Watson Research Center
Yorktown Heights, N.Y. 10598 *

This paper explores the notion of an atomic **action** as a method of process structuring. This notion, first introduced explicitly by Eswaren et al [6] in the context of data base systems, reduces the problem of coping with many processes to that of coping with a single process within the atomic **action**. A form of process synchronization, the **await** statement, is adapted to work naturally with atomic actions. System recovery is also considered and we show how atomic actions can be used to isolate recovery action to a single process. Explicit control of recovery is provided by a **reset** procedure that permits information from rejected control paths to be passed to subsequent alternative paths.

Key Words and Phrases: multiprocessing, synchronization, recovery, mutual exclusion

CR Categories: 4.22, 4.32

1. Atomic Actions

Introduction

It has long been realized that some way of restricting process interaction is required if programs involving multiple processes are to be correctly implemented. Ideas similar to atomic actions have been suggested for this purpose as far back as Dijkstra's famous paper [5]. Thus Dijkstra postulates that certain primitive operations "are to be regarded as indivisible, non-interfering actions...". Brinch Hansen states [1], even more emphatically that "It is impossible to make meaningful statements about the effects of concurrent computations unless operations on common variables exclude one another in time. So, in the end, our understanding of concurrent processes is based on our ability to execute their interactions strictly sequentially." An atomic action, as we use the term, is merely a device for permitting the writer of a procedure to secure the same benefits of atomicity, i.e. indivisibility, non-interference, strict sequencing, as is enjoyed by the primitive operations.

The important properties of atomic actions can be expressed in a number of equivalent ways. We illustrate three.

1. An action is atomic if the process performing it is not aware of the existence of any other active process (can detect no spontaneous state change) and no other process is aware of the activity of this process (its state changes are concealed) during the time the process is performing the action.
2. An action is atomic if the process performing it does not communicate with other processes while it is executing the action.
3. Actions are atomic if they can be considered, so far as other processes are concerned, to be indivisible and instantaneous, such that the effects on the system are as if they were interleaved as opposed to concurrent.

Background

The current widely known process structuring mechanisms do not provide the programmer with the ability to specify atomic actions. We review some of these below.

Dijkstra [5] proposed semaphores as a mechanism by which a programmer could assure that a sequence of actions could be regarded as indivisible. The idea is to use semaphores to assure that code intended to be

* This work was performed while the author was on a sabbatical at the University of Newcastle-upon-Tyne and was partially supported by a research grant from the Science Research Council of Great Britain.

indivisible is executed by only a single process at a time. A semaphore is used to guard the code. So long as process interactions can only occur in the "critical section" guarded by the semaphore, the code will function as an atomic (indivisible) action.

When processes can interact by means of several common variables and while executing several different sections of code, mutual exclusion by means of a semaphore guarding a critical section no longer can assure atomicity. Consider first the case of a single common or **shared** variable v that is accessed by several sections of code executed by different processes. One now needs a convention by which a semaphore can be associated with a shared variable so that all code accessing the variable is required to test the same semaphore. Such a semaphore has been called a lock[4]. Locks provide a way of assuring that only one process has access to a shared variable at a time.

Brinch Hansen [1] introduced the idea of a critical region as a means of structuring the seizing and releasing of lock semaphores. Thus, to access a common (**shared**) variable "v", one specifies a critical region

1.(1)

```
region v do S
```

where only code in S is permitted access to "v".Further, if one process is in a critical region associated with shared variable "v", all other processes are excluded from regions associated with "v".

Problems arise for critical regions as soon as one is interested in accessing more than one variable. Not only is deadlock a potential problem but one may have difficulty assuring that the critical regions are atomic. Consider the code fragment below:

1.(2)

```
region v do
  region w do S1;
    •
  † •
    •
  region w do S2;
  end;
```

The outer critical region (i.e. for "v") is no longer atomic. A second process can examine "w" at (†) and change "w" so that S2 sees the change, thus communicating with the process in "**region** v" and destroying the atomic nature of the region.

Such code sequences can be transformed, of course, into ones in which the variables are held for the duration of the outer region and these will be atomic. However, subtle cases can arise that require much more knowledge and care if atomicity is to be preserved. Consider the skeletal program of 1.(3).

1.(3)

```
b:procedure;                a:procedure;
  region w do                 region v do
    begin;                      S;
      a;                      end a;
      •
  † •
      •
      a;
      end;
end b;
```

Unless the writer of procedure "b" is fully aware of the code in procedure "a" (an unfortunate requirement, to say the least) and seizes "v" as well as "w", then, as shown in 1.(3), the procedure "b" will not be atomic since communication can occur at (†).

A way of assuring that some actions can be guaranteed to be atomic is to make use of monitors as expounded by Brinch Hansen [1] and Hoare [8]. A monitor is similar to an instance of a SIMULA class [3], i.e. it is a data object that possesses not only variable components but also procedure components. Then additional constraints are placed on the use of these components in a multiprocessing environment. These are, quoting [8]

1. "only one program [process] at a time [can] succeed in entering a monitor procedure..."
2. "Procedures local to a monitor should not access any non-local variables other than those local to the same monitor."
3. "these [local] variables of the monitor should be inaccessible from outside the monitor".

These constraints assure that the monitor procedures are atomic.

There are two problems with monitors. One, atomic actions involving more than one monitor must be implemented in an indirect way, perhaps by using monitors to realize semaphores. Two, the first constraint on monitors, i.e. that only one process can be executing any of the collection of monitor procedures, is more restrictive than necessary. What is required, simply and directly, is that monitor procedures be atomic.

Data base systems present many of the same problems as operating systems. In some respects, however, the problems are even more severe. In particular, the set of records (shared variables) that are to be accessed during a "transaction" may be very hard to determine ahead of time. Nonetheless, users desire to be presented with a consistent view of the data, i.e. one in which each of them appears to be the sole user

of the system. It is for this reason that transactions possessing the attribute of being atomic were introduced by Eswaren et al [6]. A number of interesting properties of such transactions were established in [6] but the terminology used is data base oriented and no concrete notation is suggested. The next section presents and motivates a notation, which will subsequently be augmented by a notation for process synchronization and recovery.

Action Procedures

What is needed is a facility by which the writer of a procedure can directly state his intention that a procedure be atomic. We regard the procedure mechanism as the extension mechanism for operations. Therefore, any property that is possessed by a primitive operation should be expressible when a user provides a procedure. In particular, it should be possible for a user to write a procedure that exactly reproduces the effect of any given operation. For this reason, it is essential that a mechanism be provided that permits the writing of atomic procedures. It is this line of reasoning, along with considerations of system recovery, that led us, independently of [6], to the notion of atomic actions and action procedures.

We suggest the following notation for **action** procedures.

1.(4)

```
<identifier>:action(<parameter-list>);
  <statement-list>
  end;
```

The semantics of actions are the same as those of procedures except that **actions** are to be performed as atomic actions, i.e. they are to be indivisible, etc. It should be clear that the difficulties of 1.(3) can then be avoided by writing:

1.(5)

```
b:action;
  a;
  •
  •
  •
  a;
  end;
```

```
a:action;
  S;
  end;
```

That "b" is an action assures that it is atomic regardless of the procedures or actions it may call. The effect of this is to shift the responsibility for resource acquisition and release to the implementor of **actions** rather than being the responsibility of the programmer using **actions**.

The shift of resource acquisition and release from user to implementation is simultaneously a great responsibility and a great opportunity. The implementation must now assure that deadlock does not occur (or can be overcome) while maximizing the amount of concurrency. The opportunity arises because the implementation is no longer constrained by explicit directions from the user. The user benefits enormously by having this entire messy area removed from his concern, thus enabling him to concentrate on the remaining program logic.

It should be clear that resources that can only be referenced by a single process require no special protection in order to assure that actions are atomic. This observation suggests that we syntactically distinguish **shared** and **private** resources. Doing this greatly eases the implementation burden by identifying those variables for which there is contention, i.e. the **shared** variables. Brinch Hansen [1] has previously made this suggestion though coupled with critical regions. By declaring variables as **shared** or **private**, the implementation problems for atomic actions should be comparable to those for critical regions.

The **shared** (or **private**) attribute applies to an object as a whole and not to its separate components. Local variables of a procedure are, of course, always private. To enforce that **private** objects not be accessible to other processes, we must insist that references to **private** objects not be assigned to **shared** objects. Of course, references to **shared** objects can be assigned to **private** objects. They would not otherwise be accessible.

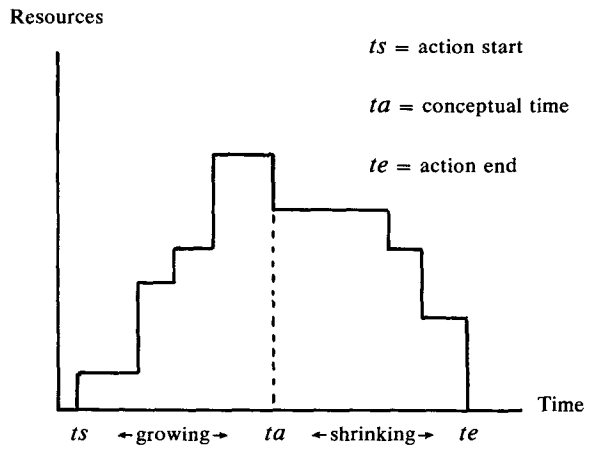
The **shared/private** attribute is useful in other ways as well. First, it serves as valuable documentation, identifying the variables that are potential communication links between processes. Second, it is useful in memory management. One can garbage collect **private** resources that are no longer accessible by their associated process. One need not examine all processes in the system looking for additional references since none can exist. Further, when a process terminates, all its private resources can be reclaimed.

Implementation Issues

There are a number of ways that atomic actions might be realized. A particularly simple one in a multiprogramming system is to execute an **action** with interrupts disabled. That is, no interrupts are taken and the action retains control of the system until it completes. In effect, it seizes all system resources during its execution. This strategem exploits the property that **actions** can be interleaved, i.e. concurrent processing in which several processes execute simultaneously is not required in order for an action to complete.

In a multiprocessor system, if we wish to exploit resources efficiently, then it is important to attempt to maximize concurrency. This requires that only resources actually needed by an **action** during its execution be acquired. Other processes wishing to use these resources must wait for them to be released.

Eswaren et al [6] has identified the pattern of resource acquisition and release required to support atomic actions. Such a pattern is called two phased. It arises as follows. As a process executing an atomic **action** proceeds, it acquires the **shared** resources it needs. This is called the "growing phase". The set of resources held is constantly increasing since a process must not release any resources so long as there may be additional resources that it will need. [See 1.(3)] Once any resource is released, no others may be acquired and the set of held resources is constantly decreasing. This is called the "shrinking phase". The conceptual "instant of time" *ta* at which the **action** occurs can be regarded as the time at which the first resource is released. It is established in [6] that this discipline of resource acquisition and release guarantees that actions have a serial schedule, i.e. their effects are as if they are interleaved. Figure 1 illustrates this strategem.



Resource acquisition and release as a function of time for atomic actions.

Figure 1

It is possible to refine this strategy. Observe that resources that are merely examined by an **action** need not be concealed from other processes. It is sufficient if other processes are prevented from changing these resources. Acquisition of resources such that other processes can examine but not change them is called "locking in the shared mode" [7]. Resources that are updated by an action must, of course, have these updates concealed from other processes. Thus, when these resources are acquired, no other process must be permitted to examine them. Such resource acquisition is called "locking in the exclusive mode" [7]. Both forms of locking must be two phased with the same *ta* [6,7].

The resource acquisition and release strategy described above does not constitute a resource management algorithm. A user cannot determine whether he executes alone or concurrently. How resource contention is handled if concurrent execution is to be achieved is not stated. Nor have we described a method for coping with deadlock or indefinite postponement. The analysis above has merely provided the framework in which a resource management algorithm must operate.

2. Process Synchronization

Synchronization using Actions

The sufficiency of atomic actions to provide synchronization can be demonstrated by presenting an implementation of semaphores in terms of atomic actions. Since semaphores are capable of realizing critical regions, conditional critical regions, and monitors, there can be no doubts about the functional adequacy of atomic actions for providing synchronization.

We provide semaphores by means of a SIMULA like class [3], the component procedures of which either are or contain action procedures. The semaphore class is defined in 2.(1).

The code for "V" needs no particular explanation. It is an **action** procedure and hence performs its effects as an atomic operation. The code for "P" is somewhat more complicated. First, "P" is not itself an atomic action. Rather it loops continuously, the body of the loop being atomic but each cycle of the loop providing an opportunity for changes to be made to "sem". Within the **action** body, "sem" is tested. If found to be greater than zero, the continuously testing loop is terminated with "sem" decremented by one. The loop termination is accomplished by calling the **escape** procedure "proceed". This construct is a variation of the "label" procedures of Landin [9] and Clint and Hoare [2]. When an **escape** procedure terminates, it returns control to the caller of its lexically enclosing procedure. Thus, when "proceed" terminates, control returns to the caller of "P".

There are two difficulties with this semaphore class definition, in particular with the body of P.

1. The repeated testing of "sem" constitutes busy waiting, consuming real processor time.

2.(1)

```
semaphore:class;
   sem:integer initial(0)
   V:action;
      sem := sem+1;
      return;
      end;
   P:procedure;
      proceed:escape; †
         return;
         end proceed;
      repeat(
         action; ‡
            if sem ≥ 0 then
               begin;
                  sem := sem-1;
                  proceed;
                  end;
               end;)
      end P;
   end semaphore;
```

‡ An unnamed **action** procedure is written here where it is to be executed, in the same way as a **begin** block.

† An **escape** procedure named "proceed", not an "escape" statement. See the text.

2. If several processes are testing the same semaphore, a race exists and there is no guarantee that some processes will not be subjected to indefinite delay. This is so because no scheduling policy is provided.

Busy waiting has yet a third difficulty if we wish to provide synchronization within an atomic action. Notice that the busy waiting in P involves time slots in which "sem" is accessible to other processes because the "wait" loop consists of a succession of atomic actions rather than being embedded in one large action. In a single atomic action the variables within the **action**, once examined, cannot be changed by other processes. Thus, busy waiting within a single action would be in vain.

It might be argued that, as with the procedure P, one can always provide for the busy waiting to involve many atomic actions, with other processes thus capable of changing the tested variables. This is extremely difficult to arrange, however. Let us suppose that "A" is an **action** procedure, that "B" is an an ordinary procedure, and that "B" uses semaphores. So far as "B" is concerned, such use of semaphores should result in a workable program. If, however, "B" is called from "A", it becomes part of an atomic action, and hence, so does the busy waiting in "B". Now, however, the busy waiting will never detect changes in "sem" and the program will loop forever. If "sem" is permitted to change, then a communication link has been established between the process executing A and B and the process changing "sem", thus destroying the atomic nature of **action** procedures.

The Await Statement

The problem with permitting "sem" to change is the fear that communication will be established with a process inside an **action** procedure. But if such a process does not remember that it has seen previous values for "sem", i.e. if there is no way for it to subsequently determine whether the test was satisfied the first time or only after many repetitions, then we can take a different view. This view is that an action procedure "A" did not commence its execution until after "sem" had changed.

What we need in order to realize this view in which the entire **action** is delayed until the test can be satisfied on its first execution, is a mechanism that informs the system that this is our intent and permits the system to enforce the required constraints. For this purpose, the **await** statement is introduced. The intent of **await** is similar to that suggested for it in [1,8], but the description of it is different in order to maintain the integrity of atomic actions.

The await statement has the following syntax:

2.(2)

```
await(<boolean expr>) then <procedure>
```

Following our view that all executable constructs should be describable as some form of procedure, we produce 2.(3) as the semantics of the **await** statement.

2.(3)

```
await:action(test:boolean function, body:procedure);
‡ delay(atomic action until prescience
   tells us that "test" is true, or that
   it escapes, then immediately execute
   the following)
   if test then
      begin;
         body;
         return;
         end;
   else error;  †
   end await;
```

† "error" might be, for example, an **escape** procedure.

It is, of course, true that such a procedure could not be written which is why **await** must be primitive. The "*delay*" at (‡) represents a bit of magic that cannot be expressed otherwise. It must be guaranteed that no subsequent testing on the part of a process can

determine how many times the "test" expression is executed. In order to assure this, it is required that "test" have no side effects. This prevents the retention of any state change other than the result of the expression, which will be **true** when control finally passes to the **then** clause. Notice that "test" is evaluated in the **action** procedure with "body". This assures that there is no possibility of the variables in "test" changing between the evaluation of "test" and the execution of "body" and hence guarantees that "test" remains true until (or unless) "body" changes those variables. When **await** is itself executed within an **action** procedure, the evaluation of "test" ensures that the variables upon which "test" depends can no longer be changed, except by the process executing this **action**. Two **await**s, one with "test" and the other with "¬test" as below:

```
2.(4)
      action;
         •
         await test then S1;
         •
         •
      †  await ¬test then S2;
         •
         end;
```

in which both are within the same action, will result in the process executing this action being indefinitely delayed at (†), provided the process itself did not change the variables of "test". Of course, if the variables of "test" cannot be changed by some other process, then the process executing 2.(4) cannot complete. Such situations can never be completely eliminated without drastically reducing the power of the language. This is true whether or not **await** is provided. One can, in fact regard endless looping or recursion as instances of the same problem.

Implementation Issues

The preceding section introduced **await** statements without placing any constraints on the form of the boolean expression that was used for synchronization. To reduce implementation problems, it may be desirable to restrict the boolean expression.

Whenever an **await** expression is not satisfied immediately, it is necessary to suspend the executing process and place it on a queue of waiting processes. Many strategies for this are available, particularly if we are not concerned with whether our waiting processes resume as quickly as possible. However, it seems desirable for the implementation to attempt re-execution of an **await** expression whenever one of its variables changes. One would like, therefore, to identify those variables that might cause the resumption of some waiting process.

One could interpretively test some indicator associated with every **shared** variable to determine whether a process waits on this variable. However, one can greatly reduce such interpretation while enhancing program readability if variables used for synchronization are explicitly designated. Thus, we suggest that at least one of the variables in an **await** expression be designated as a **synchronizing** variable, i.e. be declared with the **synchronizing** attribute. Our implementation problem is then confined to **synchronizing** variables. Only **synchronizing** variables need be permitted to change during the repeated evaluations of the **await** expression, and only the updating of **synchronizing** variables need result in interpretation to discover whether waiting processes should be resumed. With respect to acquiring and releasing resources, only **synchronizing** variables might ever be acquired and released several times by an atomic action, and then only during the repeated evaluations of the first **await** expression in which they occur.

Additional restrictions might be required in terms of the number of variables and the operations permitted upon them in order to reduce implementation cost and improve efficiency. The most restrictive requirement would be for each **await** expression to consist solely of a single **synchronizing** boolean variable. Less severe restrictions should also be feasible.

One important feature of the **await** statement in conjunction with action procedures is that, unlike the case for monitors and conditional critical regions, the concepts do not require the exposure of an underlying implementation in order for them to be understood. Thus, no explicit mention (at the conceptual level) of process queues is required, though obviously, an implementation will exploit queues and will require a scheduling strategy. Further, one need not be concerned with maintaining invariants at the point where an **await** statement occurs. Those parts that have become temporarily invalid because of updates preceding an **await** are exactly those parts of the system state that are not accessible to other processes. The components available to other processes, since they are unchanged, still satisfy their required invariants.

An Example: Buffers

Buffering is a common technique for optimizing the performance of parallel processes of the producer-consumer variety. While a consumer cannot consume what a producer has not yet produced, a buffer permits a producer to "race ahead" of the consumer, producing results that are retained in the buffer for subsequent consumption. Thus, in addition, buffers

reduce the possibility that a consumer will be delayed by waiting for a result from a producer.

We wish to provide buffers by means of **actions** and **await** statements. Our first attempt will be to modify slightly previous solutions in terms of conditional critical regions or monitors. This is shown in 2.(5).

2.(5)

```
buffer:class shared;
  frame:array(0:N-1) of T;
  count:integer initial(0) synchronizing;
  head:integer initial(0);
  send:action(x:T);
    await(count ≤ N-1) then
      begin;
        frame(head ¤ count) := x; †
        count := count+1;
        end;
    end send;
  receive:action(y:T);
    await(count > 0) then
      begin;
        y := frame(head);
        head := head ¤ 1; †
        count := count-1;
        end;
    end receive;
  end buffer;
```

† ¤ is addition modulo N.

This solution is adequate when the use of the buffers occurs outside of all atomic actions. Unfortunately, a problem arises when "send" or "receive" are used within atomic actions. Consider "receive". When a process P executes "receive" in an atomic action, the changes it makes to "head" and "count" cannot be seen by other processes. Hence, these processes cannot execute "send" (or "receive"), and in particular, cannot refill the buffer, until P completes its atomic action.

Thus, only as many messages can be received in an atomic action as are in the buffer at the time that the first "receive" is executed. This is highly unfortunate as it introduces a large measure of time dependence, and hence, uncertainty. One would like to exploit the full potential of the buffer, i.e. all its frames, whether the buffer is used outside of or within an atomic action.

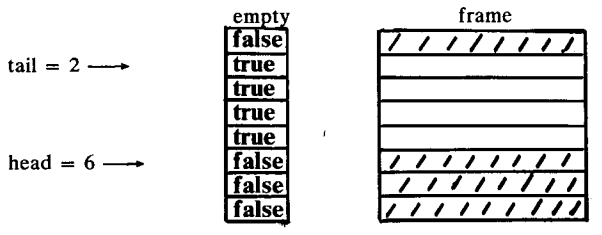
Another pitfall must be avoided. In an atomic action, it should not be possible to receive more messages than can be contained at one time in the buffer. Otherwise, we will have established communication into the atomic action. The desired solution allows the maximum flexibility in sending and receiving messages consistent with the constraints imposed by the atomicity of actions of the communicating processes. The class defined in 2.(6) provides precisely that.

2.(6)

```
buffer:class shared;
  frame:array(0:N-1) of T;
  empty:array(0:N-1) of boolean
    initial(true) synchronizing;
  head:integer initial(0);
  tail:integer initial(0);
  send:action(x:T);
    await(empty(tail)) then
      begin;
        frame(tail) := x;
        empty(tail) := false;
        tail := tail ¤ 1; †
        end;
    end send;
  receive:action(y:T);
    await(¬empty(head)) then
      begin;
        y := frame(head);
        empty(head) := true;
        head := head ¤ 1; †
        end;
    end receive;
  end buffer;
```

† ¤ is addition modulo N.

Each buffer frame is only accessed if it is actually needed and the control information governing the buffer is distributed as separate information for each frame. Further, importantly, the pointers "head" and "tail" are distributed to consumer ("receive") and producer ("send"), respectively. This distribution of control information can be readily seen by examining Figure 2 below which illustrates the relations between the various components of the buffer as implemented in 2.(6).



The bounded (circular) buffer described by the program of 2.(6). The shaded elements of "frame" contain data.

Figure 2

Note that once a frame has been accessed, it cannot be reused until the action is complete (except by the action itself). The "send" and "receive" procedures are almost completely symmetric, and hence, the flexibility provided to receivers is also provided to senders. Thus, merely because a buffer is full (or almost so) when the first send is issued within an action does not prevent it from ultimately sending as many messages as there are buffer frames. Other processes can continue to read messages deposited in the buffer by prior actions, making those frames ultimately available to the sending action.

3. System Recovery

A Unit of Recovery

By system recovery we mean the undoing of errors as opposed to their correction. This is usually thought of as consisting of two phases:

1. the rolling back of the system to a previous state, assumed to be valid, by undoing some set of actions, presumably including the erroneous ones.
2. the re-performance of the actions undone in 1. that were not (known to be) erroneous.

An error is usually associated with or detected in some process while recovery to a "checkpointed" state may involve many other processes. Thus, step 2. is needed so that correct actions are not lost. It should be clear that with a sufficiently comprehensive system log, such system recovery is always possible, though at rather great expense, so long as errors have not escaped to the "outside world".

One need not back up the entire system to provide a method of undoing errors. In an appropriately structured system, in which a programmer identifies the units of recovery, it becomes possible to restrict the undoing of errors to the process (or unit) in which they occurred. A mechanism for so structuring systems has been introduced by Randell [10] who calls this unit a recovery block.

The idea of a recovery block, in so far as undoing errors is concerned, is to isolate the process executing it from other processes. Randell states[10] that "communication, whether it involves explicit message passing or merely reference to common variables, would destroy the value of the...recovery block, and hence must be prohibited." This restriction assures that recovery blocks are, in fact, atomic.

By preventing other processes from becoming dependent upon the effects of an atomic action until the action is complete, only the process executing the action is affected by errors in the action. Hence, only this process needs to be restored to a previous state. And restoring this process involves restoring to a previous state only that part of the system that is modified by this process during the execution of the atomic action. It is unnecessary to re-perform actions of other processes since none of these were undone.

Gray et al [7] point out that a somewhat less restrictive form of "transaction" than atomic transactions also possesses this attribute of being independently recoverable. These "transactions", called "degree 2 transactions" (atomic "transactions" are degree 3) only conceal all changes until completion. Atomic actions not only prevent this communication out of an action but also prevent communication from other actions into an atomic action. Being subject to this weaker restriction, degree 2 transactions do not necessarily, as a result, possess a serial schedule nor are their effects reproducible if they are re-executed.

Recovery Bookkeeping

In order to permit atomic actions to be recoverable, their implementation must be such that

1. updated resources, i.e. those locked in the exclusive mode, are not released until the action is completed. Once a modified resource is released, independent recovery can no longer be assured as another process may examine the resource and hence become dependent upon it.
2. the initial states of all resources modified by the action can be reconstructed. This usually involves maintaining a time ordered log of update operations on which overwritten information is recorded together with its location.

In [10], recovery is realized by means of a mechanism called a "recursive cache". Rather than recording all modifications and then undoing them in reverse time order, only the first change in any location is recorded. All modifications to the system state must first be checked, interpretively at run time, to determine if a previous change has already been recorded. This means, of course, that only the starting state of a recovery block can be restored, and not intermediate ones, but this is all that is required.

A "recursive cache" is but one of a number of methods for providing recovery. A compiler could, in a large number of cases, identify updates that do not represent initial changes in a recoverable atomic action and permit these updates to run without additional interpretive overhead. Updates that might represent initial changes in an atomic action could be logged. One might, as with the "recursive cache", try to eliminate from the log all changes after the first one, but there is no need to do so. Further, strategies that are only partially successful in eliminating redundant log entries are also possible. One might employ, for example, a small associative store with recently logged

items and eliminate additional potential log entries already in this associative memory. This is quite similar to dynamic address translation in a virtual memory. These recovery strategies all need to be evaluated carefully.

Reset Procedures

Recovery facilities are, in effect, means of providing backtracking. Such backtracking is usually presented at the programming language interface in a more or less implicit guise, e.g. recovery blocks, backtrack programming, etc. There are important advantages to explicit invocation of recovery facilities, particularly if the ability to communicate information from the "failed" program path to alternative ones is desired. What we introduce is just such a feature, called a **reset** procedure.

A **reset** procedure derives its effect from its lexical context in much the same way as an **escape** procedure, previously used in 2.(1) and described in [2,9]. In addition to the effects of an **escape**, a **reset** procedure also undoes all changes produced by code executed since its enclosing procedure was invoked. We require that this enclosing procedure be an **action** procedure so as to isolate the recovery of the process executing it from other processes. An **action** procedure then becomes the unit of recovery. Consider the skeletal program of 3.(1).

3.(1)

```
x:action;
   y:reset(a);
      •
      •
      end y;
   z:procedure;
      b: local variable;
      •
      •
   †  y(b);
      end z;
   •
‡  z;
   •
   end x;
```

We assume that the call (‡) to procedure "z" is executed in action "x". Both "x" and "z" modify the process state by means of, e.g. updating variables global to "x". However, "z" encounters some difficulty it cannot cope with and realizes that some of the changes made have been erroneous. It, therefore, calls "y" at (†), passing some information via argument "b". When "y" is called, all changes to variables global to "x" are erased and the local variables of "x" are re-initialized. Only those changes produced by "y" will be detectable subsequently. When "y" terminates, it returns control to the caller of "x", exactly as if "y" had been an **escape** procedure.

We should offer a word of caution concerning the argument(s) to a **reset** procedure. If an argument is a variable passed by reference, the **reset** procedure will not see its value at the moment of call but rather its value after recovery, i.e. the value the variable had when the enclosing action was entered. Passing arguments by value does not, of course, have this potential confusion.

An Example:Recovery Blocks

We illustrate the use of **reset** procedures by programming an implicit recovery mechanism, i.e. Randell's recovery blocks [10]. Basically, a recovery block is a control structure that consists of two main components.

1. an "acceptance "test" that must be satisfied on exit from the recovery block, i.e. it is a boolean expression that must evaluate to **true**.
2. a set of alternative bodies that are executed to produce the desired effects. The first alternative body is executed and the acceptance test evaluated. If **true**, the recovery block is complete. If **false**, recovery takes place, returning the block to its initial state after which the next alternative body is executed, etc.

A syntax for recovery blocks is:

3.(2)

```
ensure <boolean expr>
   by <procedure>
   {else by <procedure>} *
   else error;
   end;
```

The procedure of 3.(3) provides the recovery semantics required of a recovery block. The resetting in "ensure" prior to the execution of the first alternative can be avoided if the invocation of "recover(1)" is replaced by most of the body of "recover". This results in duplicate code. Since there are no effects to be undone at "recover(1)", little if any cost is involved in this initial resetting.

Many other applications should exist for **reset** procedure, including some in which information of a more essential nature than illustrated for recovery blocks is passed from the rejected control path to its alternative. Sussman and McDermott [11] present a cogent argument for this, though advocating a very different mechanism for accomplishing it.

3.(3)

```
ensure:action(accept:boolean function,
   alternative:array of procedure);
   recover: reset(j:integer);
      alternative(j);
      if accept then return;
      else
         if j < highbound(alternative)† then
            recover(j+1);
         else error; ‡
      end recover;
   recover(1);
   end ensure;
```

† highbound is a function that returns the upper bound of a vector, i.e. its maximum index.

‡ error might designate an **escape** or a **reset** procedure.

5. Summary

Atomic actions have been explored as a means of structuring multiple process programs. *Action* procedures were suggested as a way of introducing atomic actions into a programming interface. Synchronization was achieved by means of the **await** statement, without exposing any underlying queueing or making explicit the acquisition and release of resources. Together, **action** procedures and the **await** statement make multiple process programming very little more difficult than sequential programming.

Because atomic actions isolate a process from the rest of the system, recovery involving restoring a process to the initial state of an uncompleted atomic action is particularly simple. **Reset** procedures were introduced to provide the user with explicit control over recovery and to permit the passing of some information from a rejected control sequence to its explicitly requested alternative.

Acknowledgements

This work was immensely aided by my frequent interactions with members of the System Reliability Project at the University of Newcastle upon Tyne. Particular thanks are due to B. Randell who stimulated my interest in this area and carefully critiqued an earlier draft of this paper; and to P. M. Melliar-Smith whose many informative discussions of this subject greatly enhanced my understanding of it.

References

1. Brinch Hansen, P. *Operating System Principles,* Prentice Hall, Englewood Cliffs, N.J., U.S.A. 1973.
2. Clint, M. and Hoare, C. A. R. Program Proving: Jumps and Functions, Acta Inf. 1 (1972) 214-224.
3. Dahl, O-J., Myhnhaug, B. and Nygaard, K. The SIMULA 67 Common Base Language, Norwegian Computer Centre, Oslo, Publication S-22 (1970).
4. Dennis, J. B., and Van Horn, E. C. Programming semantics for multiprogrammed computations. Comm. ACM 9, 3 (March 1966) 143-155.
5. Dijkstra, E. W. Co-operating Sequential Processes, in *Programming Languages* (Ed. F. Genuys), Academic Press, New York, 1968.
6. Eswaren, K. P., Gray, J. N., Lorie, R. A., and Traiger, I. L. On the notions of consistency and predicate locks in a data base system. IBM Research Report RJ1487, December 1974.
7. Gray, J. N., Lorie, R. A., Putzolu, G. R., and Traiger, I. L. Granularity of locks and degrees of consistency in a shared data base. IBM Research Report RJ1654, September 1975.
8. Hoare, C. A. R. Monitors, an operating system structuring concept. Comm. ACM 17, 10 (October 1974) 549-557.
9. Landin, P. A. A correspondence between ALGOL 60 and Church's lambda-notation: part I. Comm. ACM 8, 2 (February 1965) 89-101.
10. Randell, B. System structure for software fault tolerance, Sigplan Notices 10, 6 (June 1975) 437-449.
11. Sussman, G. J. and McDermott, D. V. Why conniving is better than planning. MIT A.I.Memo No. 255A, April, 1972.