Abstract:

Some ideas on the construction of user applications as atomic actions are developed. Atomic actions that last a long time pose several problems if 'conventional' ideas on concurrency control and recovery are applied. What is required is some means of delaying commitment without sacrificing performance. A model is proposed in which it is possible for an action to release and process as yet uncommitted objects. The impact of this on recovery is also discussed.


A Dependency, Commitment and Recovery Model for Atomic Actions

By

S.K. Shrivastava

TECHNICAL REPORT SERIES

Editor: Mr. M.J. Elphick
Number 178
June, 1982

© 1981 University of Newcastle upon Tyne.
Printed and published by the University of Newcastle upon Tyne, Computing Laboratory, Claremont Tower, Claremont Road, Newcastle upon Tyne, NE1 7RU, England.
SHRIVASTAVA, Santosh Kumar


Newcastle upon Tyne: University of Newcastle upon Tyne, Computing Laboratory, 1982.

(University of Newcastle upon Tyne, Computing Laboratory, Technical Report Series, no. 178.)


Suggested classmarks (primary classmark underlined)
Dewey (18th): 001.64404 001.6425
U.D.C. 519.687 519.681

Suggested keywords ATOMIC ACTION
COMMITTMENT
CONCURRENCY CONTROL
DISTRIBUTED SYSTEMS
RECOVERY

Abstract

Some ideas on the construction of user applications as atomic actions are developed. Atomic actions that last a long time pose several problems if 'conventional' ideas on concurrency control and recovery are applied. What is required is some means of delaying commitment without sacrificing performance. A model is proposed in which it is possible for an action to release and process as yet uncommittable objects. The impact of this on recovery is also discussed.

About the author

Dr. Shrivastava joined the Computing Laboratory of the University of Newcastle upon Tyne in August 1975, where he is currently a Lecturer.
Abstract

Some ideas on the construction of user applications as atomic actions are developed. Atomic actions that last a long time pose several problems if "conventional" ideas on concurrency control and recovery are applied. What is required is some means of delaying commitment without sacrificing performance. A model is proposed in which it is possible for an action to release and process as yet uncommitable objects. The impact of this on recovery is also discussed.

1. Introduction

Much of the recent work in the areas of reliability and data integrity in distributed systems has been concerned with the provision of atomic transactions (atomic actions) [1-6]. Such an action is characterised by the serializable property [7]: it is the unit of concurrency control such that concurrent execution of actions is equivalent to some serial order of execution. For reliability purposes, it is also convenient to make it the unit of recovery such that an ongoing action that would have normally produced changes in the system state, can be terminated without producing any state changes - the action is then said to be recovered or aborted. By structuring user interactions with the system as atomic actions with the above recovery property (henceforth termed basic actions) a powerful tool for maintaining the integrity of the shared data of the system is obtained. If during the execution of a basic action, some erroneous situation is detected such that further meaningful progress is not possible, then the recovery capability of that action is invoked for aborting the action. References [2-5] describe numerous schemes for the construction of such actions in systems.

For basic ideas on atomic actions, recovery and reliability, the reader's attention is drawn to the much cited paper of Randall et al [7].

Given a user interface that provides the facilities for the construction of basic actions, the next question that naturally arises is, can user applications be constructed out of them with similar "clean" properties? Unfortunately this does not appear to be easily possible for the following reasons:

(1) The relatively straightforward concurrency control technique that is typically used for the construction of basic actions (essentially the two stage locking scheme [7] with locks held till the end of an action) is not always practicable for the construction of "bigger" atomic actions - actions that last for a longer time than a few seconds. Consider for example an insurance claim processing application where a client's claim can take as long as say six months to process. Logically, the claim processing can be regarded as an atomic action and yet it is absurd to assume that parts of the insurance data base will be kept locked for that long a period of time. Clearly a more sophisticated concurrency control technique is necessary.

(11) The "all or nothing" property of a basic action is often not desirable, and sometimes impossible to achieve. This is because during the execution of an application, side effects might be produced that are either difficult or expensive to "undo" or are potentially unrecoverable.

The above discussion is intended to suggest that a new set of mechanisms is needed to help construct arbitrary user applications out of basic actions. The model to be developed here is a step in that direction. The ideas to be described here are a further development (with occasional recasting) of the pioneering work on spheres of control by C.T. Davies and L.A. Bjork [9-13]. We begin by reviewing the essential ideas of their work. In the rest of the paper we shall concentrate upon "data processing" applications - those concerned with long term storage and manipulation of data (e.g. banking, office information systems); though the applicability of the ideas to other fields such as process control is not ruled out.
2. Basic Ideas on Commitment and Recovery

"As is true for the trapeze artist, so must data processing have a basis for further action, such basis being a commitment to prior action" - C.T. Davies [10].

Any atomic action can be viewed at a lower level as constructed out of more primitive atomic actions - this is illustrated in the "trace diagram" of figure 1 which also introduces the diagramatic notation that will be used.

![Figure 1: Nested Atomic Actions](image)

According to Figure 1, actions E's constituents are actions A, B, C and D; the figure also shows the causal relationships between the actions. So, the execution sequence of actions A, B, C and D was "A" followed by concurrent execution of "B" and "C" followed by "D". A line joining two actions is meant to represent the fact that outputs of the "left" action are used by the "right" action. So in Figure 1, D gets its input from B and C and outputs of E are being used by F. Assume that time has advanced up to t2 and that an error is detected during the execution of C. Under such a circumstance, it is logically possible to abort C without affecting any other ongoing action (B in this case); in other words, C can be recovered ("backed out") unilaterally. What happens after C's recovery? The question must be resolved within the scope of E - the enclosing action. This leads us to the now well understood notion of nested recovery which will not be elaborated here.

As is well known, basic actions are equipped with the unilateral recovery capability with the property that this capability is discarded when a given action ends. This is a rather limited view of unilateral recovery. Assume that time has advanced up to t3; we note that as yet the outputs of "E" have not been used. So, logically it is still possible to back out "E" unilaterally (more precisely, since "E" has terminated, the state changes produced by "E" can be undone unilaterally). The question then arises as to when is it logically incorrect to back out an action? This leads us to the notion of commitment.

In our normal conversations when we say: "I am committed to ...", or "I have commitments ...", we imply that "others are depending on the promises made by me". The same idea needs capturing when we consider commitments of outputs produced by atomic actions. Informally, the outputs produced by a terminated action get committed when they are used as inputs to other actions. So, for example, in Figure 1, at time t1, no commitments have been made by A, (and A can be backed out unilaterally); at time t2, the outputs of A are committed, implying they cannot be withdrawn unilaterally. Thus commitment guarantees "input stability"[10]. A number of observations can now be made:

(a) When an action terminates, the output values produced by it have the status "commitable" implying that other actions can use them. By embellishing the notation of Figure 1 slightly, we can illustrate the idea further:

![Figure 2: Commitable and committed objects](image)

Figure 2 shows that inputs to "A" consist of objects a, b and c (shown as labels on the arcs). Henceforth the following convention will be used: upper case letters will denote actions and lower case letters will denote objects. We will assume that the function of an action is to produce new versions of its input objects; so an action is a creator of new versions (for the sake of simplicity it is will be assumed that all objects are permanent). We shall refer to the specific version of an object by indicating its input (output) relationship to a given action:

a\rightarrow A: The version of 'a' that is input to 'A';
A\rightarrow a: The version of 'a' that has been created by 'A';

Note that in Figure 2 A→a = a\rightarrow B. So from Figure 2 we see that at time t2, the version a\rightarrow B has been committed while the versions b\rightarrow C, c\rightarrow C are still only commitable.

(b) The notion of commitment is hierarchic in an obvious manner; at the level of abstraction of action E (Figure 1) no commitments have been made at time t2; yet at a lower level, outputs of A have been committed.

(c) Next we illustrate the idea of dynamic control over commitment. So long as the criteria of serializability is observed, it is possible for an action to release versions of objects before the action ends (if two phase locking is used then objects can be released during the shrinking phase). Needless to say, this achieves a greater degree of concurrency at the cost of making...
recovery more complex. The early release of an object is illustrated in Figure 3. Looking at Figure 3(a) first, it is clear that even though "B" atomically uses "a" at time t2, a> B at t2 has not committed, since action A has not yet terminated.

![Figure 3. Early release of objects](image)

A different situation is illustrated in Figure 3(b) where action B ends before action A. Here again, A>a will become committable only after time t3. The dotted lines in Figure 3 are intended to show action A's "sphere of influence" over commitment — only outside of this sphere are commitments allowed. From the point of view of recovery, "A" can be recovered so long as no commitments have been made; however, a recovery after time t2 will also include the back out of "B".

Making uncommitted objects available to other actions in a controlled manner for performance reasons is the central theme of this paper and is explored further in the subsequent sections. In the rest of the paper we will assume that basic actions are the lowest level actions and concentrate upon the mechanisms suitable for the construction of actions composed out of basic actions.

3. Degrees of Dependencies and Commitment

Dynamic control over commitment, as discussed in the last section, is a technique that can be used to obtain a greater degree of concurrency than would otherwise be possible. We generalise that idea further here by introducing the concept of degrees of dependency. In our everyday life, it is common for us to make tentative decisions, inform the concerned parties of those decisions and later on either to cancel or to confirm those decisions (take the example of booking a seat on a plane). By making our tentative decisions public, we are essentially speeding up the process of achieving our objective, since the concerned parties can now perform some tentative processing that can await final confirmation. So far we have assumed that the status of an object is binary - comittable or noncomittable; by making it many we can model the "tentative" processing in a convenient manner.

3.1. Creator to user dependency

With each version of an object we associate an abstract value that reflects the comittable status of that version: whether it is comittable and if not the chances of that version attaining the comittable status. This value is returned by a function "Sc" when applied to a given version at a given time:

$$Sc(a, A, t) = \begin{cases} 
C1 : & \text{the version of object } a \text{ as created by action } A \text{ will probably attain a comittable status at some time after } t; \\
C2 : & \text{the version of object } a \text{ as created by action } A \text{ will most probably attain a comittable status at some time after } t; \\
Cm : & \text{the version of object } a \text{ as created by action } A \text{ is comittable.}
\end{cases}$$

![Figure 4. Dependencies](image)

The intuitive meanings associated with the values are given above. Formally, a given version (say A>a) is said to be comittable at time t if the following condition is satisfied:

$$Sc(A>a, t) = Cm$$

Values C1 and C2 represent from minimum to maximum confidence in the fact that the given version of an object will eventually be comittable (it is possible to have many values between the interval C1 to Cm, but assuming only one value, C2, seems adequate). This is illustrated in Figure 4. Assume that the version of "a" at t0 is comittable and after time t1, "a" can be released by "A". Since the processing of "A" has not yet finished, "a" can only be released with a low C1 value: C1. As "A" nears the end of its processing, the chances of it being aborted decrease and hence the chances of A>a attaining the comittable status increase, so far for example, at t3, the status of "a" can be C2, while at t4, it is Cm. The values C1 are useful to the user of a given object in deciding how much reliance it can place on that version of the object; this leads us to the complementary idea of user-creator dependency.

3.2. User to creator dependency

Let C1 be the dependency value on some version A>a. If action A releases this object
before termination, then in order that some control can be exercised over the dependency value of the version \( A_0 \), we introduce the concept of user dependency. With each input object an action \( 'A' \) in this case is using, we associate a variable \( 'U' \) that can take on the following values (with their associated meanings):

\[
\begin{align*}
& \text{Di: } U_1: \text{Ci is the maximum dependency value that can be placed upon } A_0; \\
& \text{U2: } C2 \text{ is the maximum dependency value that can be placed upon } A_0; \\
& \text{Um: } Cm \text{ is the maximum dependency value that can be placed upon } A_0.
\end{align*}
\]

Informally, \( 'U' \) values can be taken to represent the "degree of importance" an action is attaching to its inputs since these values determine the commitability of the corresponding outputs.

The two dependency values \( (\text{Ci}, U_j) \) are related to each other in an important way; we can appreciate this by referring to Figure 4 and asking the question what should be the value returned by \( Sc(aB, e, t) \), given that the fact that \( 'a' \) used \( 'a' \) with a dependency value \( U_m \)? Intuition tells us that it would be wrong for this value to be \( C_m \), since \( 'a' \) as supplied to \( 'b' \) has not yet attained a commitable status. Before the relationship between them can be described formally, a few underlying assumptions about our model will be stated.

(a) Every output produced by an action is a function of all the inputs to that action.

(b) An action can assign in the beginning either \( U_1 \) or \( U_2 \) values to its inputs; as the action progresses these values can be changed. However, \( U_m \) values can be assigned only at the termination time of the action.

(c) At the termination time of an action, all of the inputs to that action must have \( U_m \) dependencies. That is, every action ultimately must have the capability of producing commitable versions of objects.

(d) An operation \( 'n' \) is defined on the values \( \text{Ci} \) and \( U_j \) of an object:

\[
\begin{align*}
& \text{Cl} \ast U_1 = \text{Cl} \\
& \text{Cl} \ast U_2 = \text{Cl} \\
& \text{Cl} \ast U_m = \text{Cl} \\
& \text{Cl} \ast U_1 = \text{Cl} \\
& \text{Cl} \ast U_2 = \text{Cl} \\
& \text{Cl} \ast U_m = \text{Cl}
\end{align*}
\]

The above equations are to be interpreted as follows: if an action acquires an object that has a dependency value \( \text{Ci} \) and uses it with a dependency value \( U_j \), (the terms on the L.H.S. of the equation) then the value \( C_m \) (the term on the R.H.S.) represents the upper bound on the the creator-user dependency on the new version of the object when it is released (made available to other actions). So, referring back to Figure 4, and remembering that \( 'b' \) has used \( 'a' \) with \( U_m \), and \( Sc(\lambda_a, t_3) = C_2 \), then \( Sc(B_0, t_3) = C_2 \). In words: action \( B \) has put maximum dependency on its input \( 'a' \), so the output version of \( 'a' \) as produced by \( 'b' \) can have as much chance of attaining a commitable status as the input version, namely, \( C_2 \). Following this, it should be clear that \( Sc(B_0, t_4) = C_m \).

(e) The following three status values are of interest: when an object \( 'a' \) is being used by an action \( 'A' \): (i) the creator-user dependency value \( \text{Ci} \) - this is the value associated with \( a_0A \); (ii) the user-creator dependency value \( \text{Um} \) which is under the control of \( 'A' \); and (iii) the creator-user value \( \text{Cr} \) - this is the value placed upon the the released version of the object \( (a_0A) \).

Let \( Z = \{ a, b, c, \ldots, n \} \) be the set of input objects to action \( 'A' \) and let \( U_0, U_1, U_e, \ldots, U_m \) be the dependencies placed on them by \( 'A' \) (see figure 5).

\[ \text{Figure 5. An Atomic Action} \]

At some time \( t_j \), \( t_j < t_e \), "A" releases some object \( 'c' \); then the \( \text{Cr} \) value associated with \( 'c' \) must satisfy the following two conditions:

\[ \text{Cr} = \min \{ Sc(k\ast A, t_j) \ast U_e \} \quad \text{for all } k \in Z \]

\[ \text{Cr} < C_m \]

\[ \text{Cr} \text{ is derived from the minimum of the the input CI values and the user dependency value placed on the given object } (U_e \text{ in this case}); \text{ further, as the action has not yet terminated , } \text{Cr} \text{ must be less than } C_m. \]

Once an action ends, the dependency values on the versions created is given by:

\[ \text{Cr} = \min \{ Sc(k\ast A, t_e) \} \quad \text{where } t \rightarrow t_e \]

That is, the dependency value is determined by the smallest of the immediately preceding dependency values. So, from Figure 4, at \( t_3 \), \( Sc(B, t_3) \) will be, using (3), the same as \( Sc(aB, t_3) \) (or using the "output" notation, \( Sc(\lambda_a, t_3) \)); this value itself will be either \( C_1 \) or \( C_2 \) as determined by action \( A \).
We can now see that the model allows a fine degree of control over controllable objects. An action can acquire uncomittable objects and release them, still uncomittable, with a lower or same dependency value (as determined by condition (1)). As actions terminate, these objects attain comittable status (as determined by condition (3)).

3.3. Commitment

It is possible now to define "commitment" formally in terms of the model: let a version of object "a" be related to two actions A and B such that \( A \alpha = aB \); then this version is said to be committed at time "t" if the following two conditions hold:

(i) \( Sc(A \alpha, t) = C \alpha \);

(ii) Action B has terminated on or before t or "B" has placed a dependency value \( UI > UL \).

We can make several observations now:

(a) The commitment of a version of an object is only possible when that version acts as an input to some other action (subject to constraints stated above).

(b) The serializability property of atomic actions ensures that if "a" at time \( t_j \) is committed, then all the versions of "a" prior to \( t_j \) will also be committed.

(c) A UI dependency cannot cause a commitment. In many applications, a user has only a vague idea about the objects needed at the start of the application; more precise information becomes available as the application progresses in time. In such a case, the user would clearly wish to commit only those input objects that are strictly needed. This can be achieved by first acquiring the objects without causing any commitments (i.e., with UI dependencies on them) and later on committing the dependencies on the required objects to higher ones, while cancelling the remaining dependencies. It is indeed possible for an as yet uncommitted version of an object to be made available to a number of actions simultaneously - all of which have placed a UI dependency; however, only one of the actions will be allowed to increase its dependency value and the rest of the actions must eventually cancel their dependencies (generalization for read locks is certainly possible). Delaying the commitment of objects as long as possible is a natural requirement for control over recoverability - we shall discuss this topic in the next section.

To conclude this section, a simple example will be used to illustrate the ideas introduced so far.

3.4. Example: Scheduling a Meeting

Designing a distributed calendar system represents a significantly challenging task [14]. We will consider the task of scheduling a meeting in the distant future between a group of people. Following the description given in [14], this task - an atomic action - can be seen to have the following phases: (a) the meeting organizer gets an agreement - without any obligations - for some possible meeting times; (b) after a while the organizer selects a possible meeting time - he would now certainly want to give the assurance that a meeting will be held at the specified time; (c) as the meeting time nears, the organizer confirms the meeting.

![Figure 6. Scheduling a meeting](image)

We shall assume some form of a distributed database system where each person has a personal calendar system for keeping track of appointments and free time slots. So, making an appointment for a meeting essentially involves finding a few free slots on the various calendars as proposed options for a meeting and later on to fix one such time slot. Every user occasionally checks his calendar for meeting proposals - if a proposal is not acceptable the user can reject that option.

The figure above shows how the scheduling of a meeting (in its simplest form) can be modelled. The organizer’s activities are modelled as an action \( M \) (for meeting) which is composed of - in this particular case - three basic actions \( T_1, T_2 \) and \( T_3 \). The task of \( T_1 \) is to select a number of possible meeting times, a UI dependency is placed on the appropriate objects representing the time slots by action \( M \) (no commitment has as yet occurred). As time progresses, the meeting organizer wants to narrow down the choice to one “most probable” meeting time - this is done by \( M \) trying to increase UI dependencies to UI. Not all such attempts will succeed - users might have rejected some options (a user can reject an option by marking that time slot as occupied - committed - hence invalidating UI dependencies).

Assuming a time slot agreeable to everyone is found, (UI dependencies can be placed upon them), \( T_2 \) is scheduled to run - its task is to update the calendars appropriately - to indicate the chosen time; at the same time any superfluos dependencies are cancelled. After the termination of \( T_2 \), \( M \) can release the meeting time which will have the dependency values C2 (assume that inputs to \( M \) are comittable). Finally, as
the meeting time nears, T3 is run to confirm it. M then ends having placed its dependencies on the chosen time slots and the meeting time is now released with status "commitable" (dependency value Cm). Running the entire activity as an atomic action has the advantage that recovery requirements are known exactly (see the next section). In the absence of the surrounding action H, it will be difficult to maintain recoverability over the three basic actions.

The simple example discussed here is intended to demonstrate that interesting aspects of calendar management can be modelled using the ideas on dependencies and commitment developed in this paper. It is also worth noting that the state changes on objects are performed by the basic actions - the so-called "atomic transactions" (measuring stable storage and two phase commit termination algorithm) and these actions can be combined into bigger actions by exercising control over dependencies and commitment.

4. Recovery Management

It has been assumed throughout that a given action is constructed out of basic actions. If a basic action aborts, it raises a failure exception which must be handled by the enclosing action. There can be several such reasons - we shall enumerate them subsequently - for the detection of exceptions (or errors) during the execution of an action.

Following Randell [8], any recovery actions taken after the detection of an error can include the steps of damage confinement and damage assessment; the former is concerned with limiting the effects of erroneous state information from spreading further into the system and the latter provides guidelines on the selection of a particular recovery strategy. In a distributed system the lack of centralised control makes the task of damage confinement and assessment quite difficult [15,16]. The recovery management scheme to be proposed here builds upon the work of Davies and Bjork [9,10,11,12,13] and Merlin and Randell [15]. We shall make a number of assumptions:

1. A given application is structured as an atomic action (which itself is composed out of basic actions); within a computer system there may be a number of related applications.

2. Associated with an application are a number of "audit routines" for consistency checking and determining - perhaps interactively with the user - the extent of backward recovery needed (in-process or post-process, see later).

3. The dependencies - who has used whose outputs - are being recorded and available for later inspection during recovery.

Let an error be detected during the execution of action C (see Figure 7). If the audit routines assess the cause of the error (fault) to be within action C then any recovery action undertaken is known as in-process recovery; on the other hand, if it is assessed that the error has been caused owing to wrong inputs being supplied to C (i.e. outputs from A and B) then the recovery action undertaken is known as post-process recovery [12]. In-process recovery involves the action in which the error has been detected and possibly other actions that have used the outputs of this action. Post-process recovery in its simplest form, involves in addition, the actions that have supplied inputs to the action in which the error has been detected.

4.1. In-Process Recovery

Let an error be detected in action C (Figure 7) at time t3 (current time) and let the recovery required be in-process. The audit routines might indicate Tj as the cause (e.g. Tj should not have been scheduled in the first place) in which case "C" can be backed out up to Tj without affecting the dependent action E. However, if a back out beyond Tj is required, then it will be necessary to "recall" side dependencies which in this particular case consists of cancelling the version Cn (making it as invalid). What happens to action E? There are two situations under which a back out of "E" is required: (1) "E" has placed a dependency value UDUU on Cn; or (2) "E" has placed a dependency value DU on Cn and DU exists. We make it the responsibility of an on going action to make sure that any of its as yet uncommitable input object versions (i.e. versions with CI values less than Cm) are still valid and to take the appropriate recovery action. By its very nature, it is always possible in the case of in-process recovery to back out the action such as C and all its dependent actions - in-process recovery represents the case when no commitments have been made.

4.2. Post-Process Recovery

Let current time be equal to t1 when an error is detected in C and further, the audit routines assess the output from action A as incorrect. Since Aa is as yet uncommitable,
this version is cancelled (or marked as invalid) and ‘C’ is backed out (as discussed before). We make it the responsibility of on-going actions to make sure that any uncommitable versions released by them are still valid. If for an action it is detected that one of its output has been marked invalid, then this represents the detection of an error which could lead to the invocation of either in-process or post-process recovery (within ‘A’ in Figure 7).

Assume the same situation as before except that time has advanced up to 2 (Figure 7) and that A μ A is still not commitable (because inputs to A have not yet attained a commitable status). The fact that ‘A’ has terminated makes no logical difference to the recovery actions required; indeed a suitable mechanism might involve making ‘A’ ‘active’ again—purely for the purpose of invoking in-process or post-process recovery within ‘A’.

Implicit in the discussion above was the assumption that ‘C’ has placed a dependency on A μ A that is greater than UI. We now consider the case whereby the ‘wrong input’ of ‘C’ enjoys the status ‘committed’. Under such a situation recovery—in addition to the back out of ‘C’—is possible if the following conditions are met: (a) ‘C’ is the only user of A μ A (the case considered in Figure 7; note that if read locks have been placed on A μ A then there could be multiple users); and (b) all the other outputs of ‘A’ are commitable only (not yet committed). It is possible then to perform recovery of ‘A’.

If however the above two conditions are not met—thus implying that the outputs of ‘A’ have been committed by actions in addition to ‘C’—then apart from backing out ‘C’ very little else can be achieved as far as backward recovery is concerned. Any further recovery actions are very much application dependent and involve running compensating actions [9]. Whether recovery by compensation can be modelled within the framework of this model remains to be seen.

A suitable mechanism of the recovery actions outlined above (determining whether conditions (a) and (b) hold) might involve chasing as described by Bjork [9] and Merlin and Randall [15]: an attempt is made to prevent the as yet uncommitted outputs of ‘A’ from getting committed.

We conclude this section by enumerating various exceptions that could be raised during the execution of an action: (1) a basic action aborts; (ii) the user explicitly aborts the action; (iii) a request to increase one of the UI dependencies fails; (iv) some of the inputs to the action become invalid; (v) some of the outputs produced by the action become invalid; and (vi) user supplied consistency checks reveal an error.

5. Concluding Remarks

In a recent paper [17] J.N. Gray stressed the need for ‘nested transactions’ for modelling long running applications. It is argued here that in addition to allowing actions to be nested, it is also necessary to provide some means of delaying commitement as far as possible without reducing the degree of concurrency. A model was then proposed in which it is possible for an action to release (as well as process) as yet uncommitable objects. The following is a list of suggested areas for further research: (a) to test the suitability of the model by applying it to various applications (so far I have considered only simple test cases); (b) consider implementation issues—the work of Bjork [18] on ‘audit trails’ and that of Needham and Herbert on ‘sequences’ [19] appears to be particularly helpful here in deciding how to record dependencies and schedule basic actions within an action; (c) lastly, any relationship to other models such as those for office systems [20] needs to be explored.

Acknowledgements

Acknowledgement is due to Brian Randell with whom I have had many conversations on this subject. I had the pleasure of meeting Charles Davies on two occasions (first when he visited Newcastle in the summer of 1978 and secondly when I went over to the USA in the summer of 1979) during which time he was kind enough to explain many of his ideas on spheres of control to me. Earlier versions of this paper were read critically by Flavio Cristian and Brian Randell. This work was supported partly by the Science and Engineering Research Council (UK).

References


