Implementing Fault−Tolerant Distributed Applications Using Objects and Multi−Coloured Actions

Santosh K Shrivastava and Stuart M Wheater

Computing Laboratory
University of Newcastle upon Tyne, UK.

Abstract

This paper develops some control structures suitable for composing fault−tolerant distributed applications using atomic actions (atomic transactions) as building blocks, and then goes on to describe how such structures may be implemented using the concept of multi−coloured actions. We first identify the reasons why other control structures in addition to − by now well known − nested and concurrent atomic actions are desirable and then propose three new structures: serializing actions, glued actions and top−level independent actions. A number of examples are used to illustrate their usefulness. A novel technique, based on the concept of multi−coloured actions, is then presented as a uniform basis for implementing all of the three action structures presented here.

Key words: Atomic actions, atomic transactions, concurrency control, distributed systems, fault-tolerance, object−oriented system.

1. Introduction

Distributed computing systems provide new opportunities for developing flexible and reliable applications; at the same time, because of independent failure modes of the system components, such systems pose design and implementation problems not normally encountered in more conventional centralized systems. A computational model that has been widely advocated for constructing fault-tolerant distributed systems is based upon the concept of using nested atomic actions controlling operations on persistent objects. This paper develops some control structures suitable for composing distributed applications using atomic actions as building blocks, and then goes on to describe how such structures may be implemented using the concept of multi−coloured actions.

2. A Review of Objects and Actions in Distributed Systems

We assume that the hardware components of the system are workstations (nodes), connected by a communication subsystem (for example, a local area network). A node is assumed to possess the property of *fail−silence*, that is, it either works as specified or simply stops working (*crashes*). After a crash a node is repaired within a finite amount of time and made active again. A node can have both *stable storage*, a type of storage which can survive node crashes with high probability, and non−stable (*volatile*) storage or just non−stable storage (the former modelling a diskfull workstation and the latter a diskless one). All of the data stored on volatile storage is assumed to be lost when a crash occurs; any data stored on stable storage remains unaffected by a crash. It is also assumed that faults in the communication subsystem are responsible for failures such as lost, duplicated or corrupted messages. Well known network protocol level techniques are available for coping with such failures, so their treatment will not be discussed further. The failure assumptions made here are similar to those made in other research work reported in the literature [e.g. 1].

An object is an instance of some *type* or *class*. Each individual object consists of some variables (its instance variables) and a set of operations (its methods) that determine the externally visible behaviour of the object. The operations provided by a class have access to the instance variables and can thus modify the state of an object. Furthermore, the class of an object determines what operations may be applied to it.

In a distributed system, an operation upon an object is typically invoked via a *remote procedure call* (RPC) mechanism, which uses messages for passing parameters to the operation and returning the results of the operation to the caller. Programs which operate on objects are executed as *atomic actions (atomic transactions)* with the properties of (i) *serializability*, (ii) *failure atomicity*, and (iii) *permanence of effect*. The first property ensures that the concurrent executions of programs are free from interference (i.e. a concurrent execution can be shown to be equivalent to some serial order of execution). The second property ensures that a computation can either be terminated normally (*committed*), producing the intended results or it can be *aborted* producing no results. The abortion property may be obtained by the appropriate use of backward error recovery, which is invoked whenever a failure that cannot be masked occurs. Typical failures causing a computation to be aborted include node crashes and communication failures such as the continued loss of messages. It is reasonable to assume that once a computation terminates normally, the results produced are not destroyed by subsequent node crashes. This is the third property − *permanence of effect* − which ensures that any state changes produced (i.e. new states of objects modified in the atomic action) are recorded on *stable storage*. A *commit protocol* is required during the termination of an atomic action to ensure that either all the objects updated within the

atomic action have their new states recorded on stable storage (*committed*), or no updates get recorded (*aborted*). A variety of concurrency control techniques for atomic actions to enforce the serializability property have been reported in the literature. A very simple and widely used approach is to regard all operations on objects to be of type *read* or *write*, which must follow the well known locking rule permitting *concurrent reads but only exclusive writes*. In a classic paper [2], Eswaren *et al.* proved that a *two−phase* locking policy can ensure the serializability property. During the first phase, termed the *growing phase*, an atomic action can acquire locks on objects but not release them. The tail end of the computation constitutes the *shrinking phase*, during which time held locks can be released but no locks can be acquired. To prevent the danger of *cascade aborts*, locks are released simultaneously, at the commit time of an atomic action. (To understand how cascade aborts are possible if locks are released gradually, consider an atomic action which aborts after releasing some locks on objects which have been subsequently locked by other atomic actions, then these atomic actions could be forced to abort as well.)

The object and action model provides a natural framework for designing fault−tolerant systems with persistent objects. Objects normally reside in object stores − which are designed to be stable. Atomic actions are employed for producing state changes; the properties of atomic actions ensure that only consistent state transformations take place on objects, despite failures. The availability of objects can be increased by *replicating* them and storing them in more than one object store. Replicated objects must be managed through appropriate *replica−consistency protocols* to ensure that object copies remain mutually consistent [3].

Several enhancements to the model described above have been proposed. One enhancement, permitting a finer control over recovery and concurrency, is to permit *nesting* of atomic actions; further, nested atomic actions could be concurrent. This is illustrated in fig. 1 which shows a system of three atomic actions, *A, B*, and *C*, where atomic actions *B* and *C* are concurrent and nested within *A*. Lines connecting atomic actions indicate the flow of control along the timeline; at any time the control resides with the innermost atomic action(s) at that point. The outermost atomic action, such as *A*, is referred to as a *top−level atomic action.* The permanence of effect property is then possessed by top−level atomic actions only; so referring to fig. 1, if atomic action *A* aborts, the states of any objects modified by atomic actions *C* and *B*, if any, will be recovered to their previous stable states. If *A* commits, all of the states of persistent objects modified within *A*, *B* and *C* will be made stable. Another enhancement is to introduce *type specific concurrency control*, which is a particularly attractive means of increasing the concurrency in a system. The idea is to permit concurrent read/write or write/write operations on an object from different atomic actions provided these operations can be shown to be non interfering (for example, for a directory object, reading and deleting different entries can be permitted to take place simultaneously). Object−oriented systems are well suited to this approach, since semantic knowledge about the operations of objects can be exploited to control permissible concurrency within objects [4,5]. The idea can be taken further by introducing *type specific recovery*. For example we can exploit the fact that if some operations, say *add()* and *subtract()* of an object *commute*, then if an atomic action aborts after having performed, say an *add()* operation, then rather than recovering the state of the object, the corresponding *subtract()* operation can be performed, thereby *compensating* the effects of its prior activities.

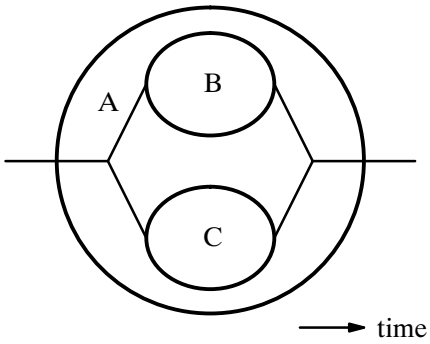

Fig. 1. Concurrent nested atomic actions

We next address the problem of composing distributed applications using atomic actions. The structuring mechanisms available are sequential and concurrent compositions of atomic actions, nested within a top−level atomic action. These mechanisms are sufficient if an application function can be represented as a single top−level atomic action. Frequently this is not the case. Top−level atomic actions are most suitably viewed as "short−lived" entities, performing stable state changes to the system state [6,7]; they are not appropriate for structuring "long−lived" application functions (running for hours, days...). Long−lived top−level atomic actions may reduce the concurrency in the system to an unacceptable level by holding on to locks (in the case of two−phase locking) for a long time; further, if such an atomic action aborts, much valuable work already performed could be undone. Much of the research on structuring applications out of atomic actions has been influenced by the ideas on *spheres of control* put forward by Davies [8]. However some of these and related ideas [e.g. 9] have not been found easy to implement.

In this paper we present some simple and practical ideas on composing applications out of nested and top−level atomic actions. A novel technique, based on multi−coloured actions, is presented as a uniform basis for implementing a variety of action structures presented here. We require an application (or its function) to be *modular,* meaning that it can be designed and implemented independent of other applications in the system. This is of course the great strength of the atomic action concept. We further require that an application builder be given control over concurrency and recovery in a much more flexible manner than is possible just by nesting of atomic actions.

3. Action Structures

Problems encountered when structuring an application using nested atomic actions can be appreciated by considering a specific example. Consider three atomic actions A, B and C (see fig. 2). B is ordered to occur before C, and both B and C are nested within A. Assume that B performs some long and complicated computation using a set of objects b, and C performs some operations on the set of objects $\{b, c\}$. It is required that the set of objects in b remain unmodified between the end of B and the start of C; that is why A is necessary. If B terminates successfully but a failure prevents completion of A, then A will be aborted, thereby undoing the effects of B and C. We can imagine situations where one requires the functionality whereby either A completes normally or if A aborts after having completed B then the effects of B survive. Such a functionality cannot be obtained by nesting B and C within A. The problem is that the only way objects in set b can be prevented from being changed between the end of B and start of C is by using an enclosing atomic action. What is necessary is for A to be 'atomic with respect to concurrency' but not with respect to failures. This and other forms of functionalities can be obtained using action structures to be described here.

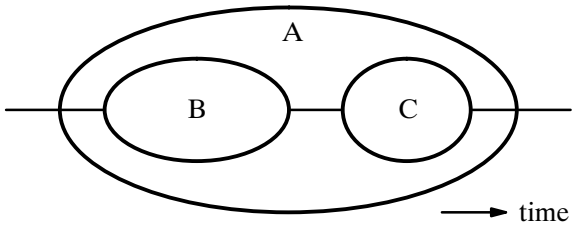

Fig. 2. Nested atomic actions

The smallest unit of work will be assumed to be an atomic action. We will describe how a number of top−level actions can be composed to provide the functionality required by many applications. We will present three types of actions: serializing, glued and top−level independent. In the rest of the paper, locking will be assumed as a means for concurrency control; this will make the discussion more specific but easier to understand.

3.1. Serializing Actions

The example discussed previously hinted at the need for relaxing the failure atomicity property of atomic actions − the resulting structure will be called a *serializing action.* The use of a serializing action provides a mechanism by which locks on objects can be transferred from one of its constituent actions to another, while preventing non−nested actions from acquiring conflicting locks on them. Serializing actions follow the same locking rules as atomic actions. Thus referring to fig. 3 (which shows the system depicted in fig. 2, except that A is now a serializing action), once B terminates, the locks released by B will be *retained* by A. The locks retained by A can later be acquired by a nested action, such as C. The constituents of a serializing action (B and C in fig. 3) are treated as top−level actions as far as permanence of effect property is concerned. As a result, an execution of A can have one of the three possible outcomes (assuming B and C modify objects):

(i) No effects are produced (because B aborts);

(ii) Effects from B and C become permanent and are made visible simultaneously (B and C commit);

(iii) Effects of B only become permanent and made visible (B commits, C aborts).

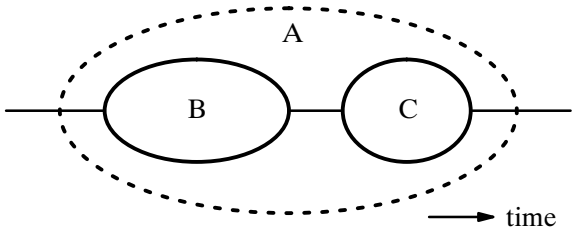

Fig. 3. Serializing action

3.2. Glued Actions

Glued actions are a generalization of the serializing actions. They provide a simple mechanism for increasing the concurrency in the system, particularly when long running actions are considered. If long running top−level actions retain locks till commit time then the degree of concurrency possible in the system can suffer. One possible method of increasing concurrency is then to permit early release of locks (during the shrinking phase of two−phase locking), but this method, as stated before, can cause a cascade of actions to be aborted if the releasing action aborts. Glued actions provide a control structure for releasing locks on objects without the possibility of the cascade aborts.

The idea behind glued actions can be explained with the help of a simple example. Imagine that an application is composed of two atomic actions A and B, where B is a long running action. The purpose of A is to modify (or inspect) a set O of objects, $O=\{O_1....O_n\}$, and select a subset P of O, $P=\{O_i....O_k\}$; this subset is passed on to B which then performs a time consuming computation on P. Objects in P should remain unchanged between the end of A and start of B. The effects of A on O should not be recovered if B fails.

The last requirement rules out the use of an enclosing atomic action. Thus, one can run A and B as two top−level actions (fig. 4[a])− but this does not meet the requirement of guaranteeing that objects in P remain unchanged in between the end of A and the start of B. An alternative is to enclose A and B inside a serializing action (fig. 4[b]). This solution is not entirely satisfactory since write and read locked objects in the set $O - P$ will remain inaccessible and unmodifiable respectively till B ends; this is clearly undesirable since B does not require those objects. The solution is to 'glue' B to A (fig. 5) which allows locks on the relevant objects to be passed from A to B, and the remaining locks to be released at A's commit time. The gluing of B to A is intended to show that certain locks on objects are passing from A to B 'atomically', without other actions being allowed to acquire them, and does not mean that B must be initiated immediately after A. If there is an interval of time between the end of A and the start B, it is not depicted in the figure.

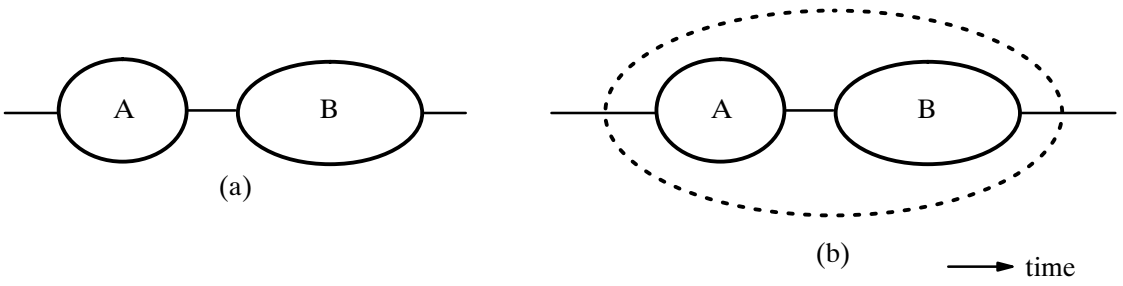

Fig. 4.

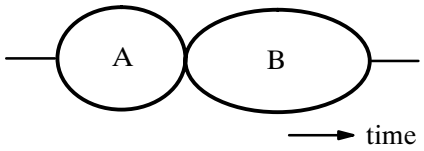

Fig. 5. Glued actions

If all of the locks held by A are passed on to B, then the system of glued actions (fig. 5) will become identical to the serializing action system (fig. 4[b]). Thus a serializing action can be seen as a special case of the use of glued actions. Nevertheless it is preferable to employ separate pictorial representations, enabling application requirements to be expressed clearly. Gluing of actions can be performed among concurrent actions, as illustrated in fig. 6.

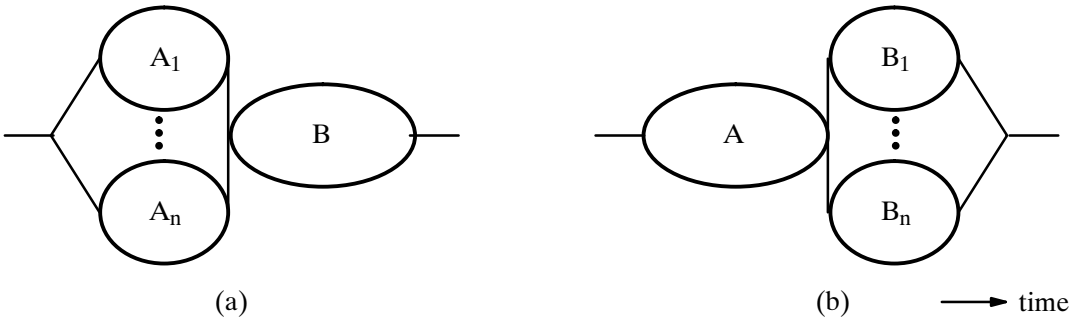

Fig. 6. Concurrent glued actions

3.3. Top−Level Independent Actions

It is sometimes desirable to provide a mechanism for invoking a top−level action from within an action [1]. So, if the invoking action aborts, this does not lead to automatic abortion of the invoked action − which can commit or abort independent of its invoker. Such an invoked action will be referred to as a *top−level independent* action. Such actions could be invoked either synchronously (fig. 7[a]) or asynchronously (fig. 7[b]); in both the cases, B (an independent action) can commit or abort independent of A. When a top−level independent action has been invoked synchronously, the invoking action (A in fig. 7[a]) will continue only after B has committed or aborted; subsequent activities of A can be made to depend upon the outcome of B (for example, if B aborted, A might abort as well).

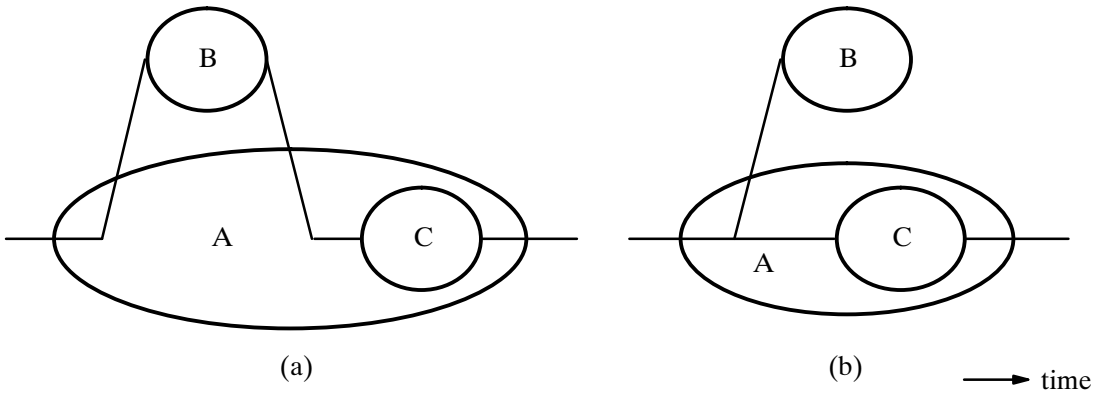

Fig. 7. Top−level independent actions

3.4. Summary

Three structures to complement nested atomic actions were proposed. Serializing actions provide control over concurrency, but do not possess failure atomicity property. They provide a means of composing an application out of top−level actions; a generalization of serializing actions − glued actions − was proposed to gain additional flexibility over concurrency control. Finally, synchronous and asynchronous top−level independent actions were proposed permitting an action to schedule top−level actions. The uses of such action structures will be illustrated with the help of several examples in the next section. Note that once a top−level action commits, its effects can only be 'undone' by running one or more application specific *compensating* actions [8]. Developing mechanisms for compensation within the framework proposed here is left as a topic for further research.

4. Examples

This section contains five simple examples to illustrate the applications of action structures proposed previously. More examples can be found in [10].

(i) *Bulletin Board:* Posting and retrieving information from bulletin boards can be performed via synchronous or asynchronous top−level independent actions invoked from applications structured as actions. While it is desirable for bulletin board operations to be structured as atomic actions, if these actions are nested within the actions of an application, then bulletin information can remain inaccessible for long times. Top−level independent actions give the desired functionality. Of course, if the invoking action aborts it may well be necessary to invoke a compensating top−level action; this is consistent with the manner in which bulletin boards are used.

(ii) *Name Server Access:* Most distributed systems employ a *name server* for maintaining a variety of information pertaining to objects. For the sake of availability and consistency it is desirable that a name server be replicated and operations on it (such as add, delete, lookup) structured as atomic actions. Such atomic actions can be invoked as top−level independent actions from within distributed applications. Thus an application level action, upon finding out that certain objects are unavailable due to a node crash, can invoke a top−level independent action to update the name server asynchronously, while carrying on with the main computation. There is no reason to undo the name server updates should the invoking action abort.

(iii) *Billing and accounting resource usage:* If a service is accessed by an action and the user of the service is to be charged, then the charging information should not be recovered if the action aborts. Top−level independent actions again provide the required functionality.

(iv) *Distributed make:* This example will illustrate the uses of serializing actions. *Make* is a well known Unix utility program used mainly for maintaining the consistency of programs which have been constructed from many modules. It ensures that a program is recompiled if its constituent modules have been changed. A file (*target file*) is regarded as being consistent if all the files it depends upon are consistent and were last changed earlier than the target file. The files that the target file depends upon are call *prerequisite* files. Each file has a timestamp associated with it, which is updated automatically every time the file is changed. The user must specify, in a file called a *makefile*, the dependencies of a target file, along with the commands necessary to reestablish the consistency of the target file. An example of a *makefile* is given below:

```
Test: Test0.o Test1.o
      cc -o Test Test0.o Test1.o

Test0.o: Test0.h Test1.h Test0.c
      cc -c Test0.c

Test1.o: Test1.h Test1.c
      cc -c Test1.c
```

In this example, the target file `Test` depends upon prerequisite files `Test0.o`, `Test1.o`. These two files in turn depend upon `Test0.h`, `Test1.h`, `Test0.c` and `Test1.h`, `Test1.c` respectively. If *make* were to be run with the above makefile and files `Test0.o` and `Test1.o` were consistent but had more recent timestamps that `Test`, then the command '`cc -o Test Test0.o Test1.o`' will be executed to make `Test` consistent. Note that *make* is recursive. We will consider the design of a fault−tolerant distributed *make* for a distributed system.

Within a distributed environment, a *distributed make* program should possess the following three characteristics: (i) it should take advantage of potential concurrency available: (ii) it

should employ proper concurrency control, such that while *make* is using a makefile, the relevant files are not manipulated by other programs; and (iii) *partial success:* if *make* fails, any files that have been made consistent should remain so (i.e., there is no reason to undo the work already performed). The above characteristics suggest that serializing actions would provide the necessary functionality. The process of making a target file consistent splits naturally into four phases: (i) ensuring the consistency of prerequisite files; (ii) obtaining the timestamps of prerequisite files; (iii) obtaining the timestamp of the target file; and (iv) execution of the commands to reestablish consistency (if necessary). The last three phases can be performed as one or more atomic actions, enclosed by a serializing action. Each prerequisite file can be made consistent concurrently with others. Thus, the serializing actions resulting from the makefile discussed previously are depicted in fig. 8 where it has been assumed that both `Test0.o` and `Test1.o` need to be made consistent.

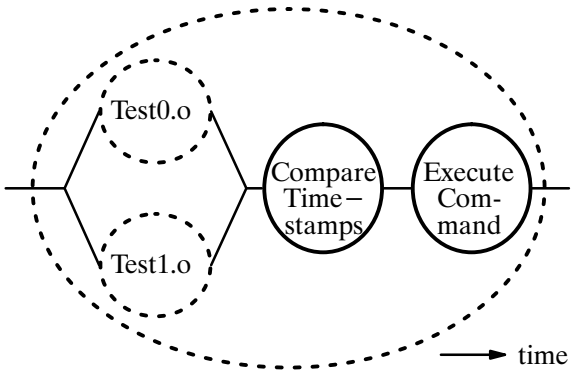

Fig. 8. Distributed make

(v) *Arranging a meeting*: This last example illustrates an application of glued actions. The requirement is to arrange a date for a meeting between a group of people. The application program starts off by informing people of a forthcoming meeting and then receiving from each member of the group a set of preferred dates. Once all this information has been gathered, it will be analysed to find out the set of acceptable dates for the meeting. This set is then broadcast to the group, to get a more definitive idea for preferred date(s). This process is repeated till a single date acceptable to all is determined. It will be assumed that each user has a *personal diary* object which records the dates of meetings and other appointments for a user. A personal diary is made up of diary entires (or slots) each of which can be locked separately.

Glued actions are useful in structuring such applications, since locks on diary entries can be passed from one top−level action to the other. The structure of the resulting system of actions is shown in fig. 9. Action I_1 locks all the relevant diary entries and selects some possible slots. Some time later, these slots are examined by I_2 which narrows the choice down to fewer than I_1, and passes them on to I_3 and so on. Each I_i is a top−level action, so its results survive crashes; at the same time meeting slots not found acceptable are released (and not handed over to I_{i+1}) thereby ensuring that entries in diaries are not unnecessarily kept locked.

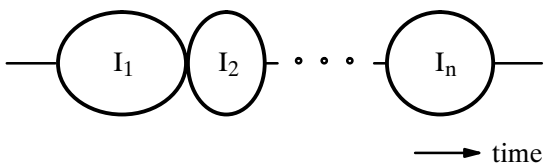

Fig. 9. Glued actions

These examples illustrate that the structuring concepts presented here do provide convenient means for composing distributed applications. Next we will describe how such structures can be implemented.

5. Implementation Using Colours

It is of course possible to implement each of the structures presented in the last section separately, using special techniques. However such techniques taken together are likely to prove unattractive for implementation. Instead we present a novel mechanism − based on coloured actions − which can be used to implement all the structures discussed here in a uniform manner. The implementation will be discussed in terms of locking. The basic idea is to assign an attribute − a colour − to each action. Coloured actions of the same colour possess properties similar to those of conventional atomic actions, but not necessarily with respect to actions of different colours. The power of this approach derives from allowing an action to possess multiple colours; this enables a system of actions to be constructed to match the concurrency and recovery requirements of a given application.

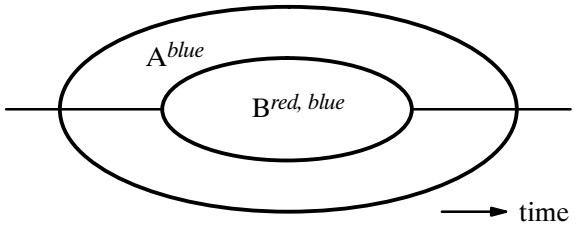

Fig. 10. Coloured actions

We will first give a simple example to hint at the properties of coloured actions and then present a more formal description. Fig. 10 shows two actions: A, coloured *blue* is enclosing action B, which is coloured both *red* and *blue*. Action B locks a set of objects, say O_r, with colour *red* and another set of objects O_b with colour *blue*. After B commits, all of the locks on O_b − (*blue locks*) are retained by A (which is coloured *blue*), but all the *red* locks on O_r are released. Action B behaves like a top−level action as far as objects in O_r are concerned but like a nested action as far as objects in O_b are concerned. So if A aborts after B has committed, only the effects of objects in O_b will be undone. As we will see, by judicious choice of nesting and colours, we can obtain a variety of behaviours from a system of coloured actions.

5.1. Properties of coloured actions

The properties of coloured actions are intended to permit more flexible concurrency and recovery than conventional atomic actions. We assume that actions are statically assigned colours. Coloured actions have properties which correspond to the atomic action properties of serializability, failure atomicity and permanence of effect, as described below:

1. *Failure atomicity:* a coloured action on completion either has performed the desired effects or has had no effect at all, on the objects accessed using the colours of that action.

2. *Serializability:* the execution of concurrent coloured actions with identical colours should produce the same effects as some serial order execution of the actions, given that no information is communicated between actions of the same colour using nested actions with a different colour.

3. *Permanence of effect:* once an outermost coloured action (a coloured action which is not nested within a coloured action of the same colour) of a particular colour has committed, all changes of that colour produced become permanent.

It should be noted that if all the actions in a coloured system possess the same single colour then the system reverts to being just a normal atomic action system.

5.2. Locking rules

To show the difference between atomic actions and coloured actions, it is necessary to compare their locking rules. In the case of coloured actions, it will be assumed that an action can specify one of its colours while making a lock request; this colour is assigned to the lock if the request succeeds. We will assume three modes in which a lock may be acquired: *read*, *write* and *exclusive read*. A read lock on an object allows the holder to read the object (reading can be shared, permitting several different actions to hold read locks on an object). A write lock on an object allows the holder to write to the object exclusively. An exclusive read lock on an object allows the holder to read the object exclusively. In a non−coloured system, the holder of an exclusive read lock on an object can always *convert* that lock to a read lock or *acquire* a write lock on that object; in a coloured system this is only possible subject to the read and write lock rules given below. Exclusive read locks are necessary purely to enable a coloured system to implement the action structures of section 3, and are not necessary for the conventional atomic action systems; however, for the sake of completeness, there existence will be assumed for both types of systems.

The locking rules specify the conditions under which locks may be granted on objects that are already locked (obviously, locking an unlocked object is always possible), and also the behaviour of atomic actions/coloured actions when they commit or abort. In the conventional atomic action systems [11], the locking rules are as described below:

- An atomic action may acquire a lock on an object in a read mode if all holders have read locks on the object or if all holders of write locks and exclusive read locks on that object are ancestors of the requesting atomic action.

- An atomic action may acquire a lock on an object in a write mode (or exclusive read mode) if all holders of locks on that object are ancestors of the requesting atomic action.

- When an atomic action commits, all the locks held by the atomic actions are inherited by its parent (if any). This means that the parent will hold each of the locks in the same mode as the child held them.

- When an atomic action aborts, all locks it holds (in all modes) are discarded. If any of its ancestors hold the same lock, they continue to do so, in the same mode as before the abortion.

When using coloured actions and locks, the locking rules are as follows:

- A coloured action may only hold locks of its own colours.

- When acquiring locks, a coloured action may only use the colours which it possesses.

- A coloured action may acquire a lock on an object in a write mode using one of its colours α if all the holders of the locks (of all colours and modes) on that object are ancestors of the requesting coloured action, and all the write locks held on the object are coloured α. (This means that if an ancestor of a coloured action has a write lock of colour α on an object, then the coloured action may only acquire a write lock on that object using colour α.)

- A coloured action may acquire a lock on an object in a read mode using one of its colours if all the holders have read locks on the object or if all the holders of write locks and exclusive read locks on that object are ancestors of the requesting coloured action.
- A coloured action may acquire a lock on an object in an exclusive read mode using one of its colours if all the holders of locks (of all colours and modes) on that object are ancestors of the requesting coloured action.
- When a coloured action with colours $\alpha_1..\alpha_n$ commits, its locks of colour α_i, $1 \leq i \leq n$, are inherited by the most recent ancestor (if any) with colour α_i. This means that the ancestor retains the lock in the same mode and colour as the coloured action held them.
- When a coloured action aborts, all the held locks (of all colours and modes) are discarded. If any of its ancestors hold the same locks, they continue to do so, in the same mode and colour as before the abortion.

The following sections will illustrate the applications of the locking rules in implementing the action structures of section 3.

5.3. Implementing Serializing Actions.

To provide the behaviour exhibited by serializing actions using coloured actions is simple. We exploit the fact that the behaviour of an action performing *read only operations* is equivalent to a serializing action. Thus to implement the system shown in fig. 3, what is required is to colour the actions as shown in fig. 11. Let *W* and *R* be the set of objects to be updated and read by *B* respectively. To prevent objects in *W* from being modified or inspected by any action during the interval *B* is active and the interval starting from the end of *B* to start *C*, action *B* write locks them in colour *blue* and also exclusive read locks them in colour *red*. Further, all the objects in *R* are read locked in colour *red* as well as in *blue*. When *B* commits, *blue* write locks on objects in *W* are released (but not retained by *A* which is of a different colour) thereby making state changes produced on objects in *W* stable; at the same time *A* retains read locks on objects in *R* and exclusive read locks on objects in *W*. Thus while objects in *R* remain accessible in read mode, those in *W* remain inaccessible. Action *C* can acquire (without possibility of lock conflict) a *blue* read lock on any of the objects upon which action *A* retains locks (see the locking rule for such locks). *C* can also acquire a *blue* write lock on any of the objects upon which *A* retains an exclusive read lock. When *C* commits, all the blue locks are released and the states of updated objects made stable. It can be seen that *A* is acting purely as a controlling mechanism for passing objects from *B* to *C*. Further, action *A* does not perform any write operations; thus the behaviour of the coloured action system shown in Fig.11 is identical to the serializing action system shown in Fig. 3; in particular they possess identical concurrency and failure atomicity properties.

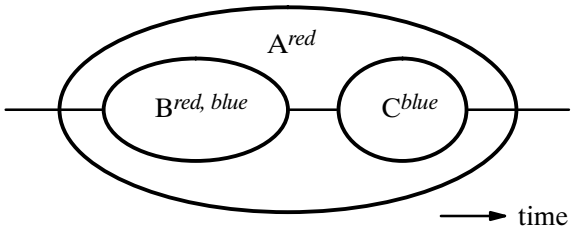

Fig. 11. Implementing a serializing action

5.4. Implementing Glued Actions

The same action colouring technique can be used to implement glued actions. Consider the example discussed with reference to fig. 5. We require that out of the set *O* of objects, a

subset P is to be passed on from A to B. For the sake of simplicity, assume that all the objects in P have been modified by A. Then, the colouring and locking scheme necessary is depicted in fig 12. An additional *red* coloured action G is required to pass on the locks from A to B. Action A locks all the objects in O in the appropriate modes with colour blue; in addition, objects in P are read exclusive locked in colour red. Action B can automatically acquire write locks in colour blue on the objects in P, due to the exclusive read mode locks retained by action G. The colouring scheme can be used in a straightforward manner to implement concurrent glued actions, such as those depicted in fig. 6. Thus the scheme shown in fig. 6(a) can be implemented by giving $A_1..A_n$ colours red and blue and enclosing them within a red coloured action (similar to G in fig. 12).

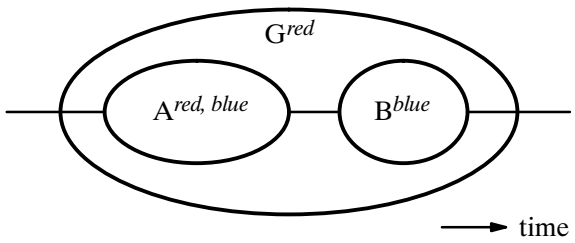

Fig. 12. Implementing glued actions

5.5. Implementing independent top−level actions.

Such actions can be implemented easily using coloured actions by giving them colours which are different from those of the invoking actions. Thus to implement the system shown in fig 13(a), A and B will be coloured as shown in fig. 13(b). Note that the two systems will behave identically provided B does not need conflicting access to objects locked by A. If this were the case, then A and B in the system depicted in fig 13(a) would be deadlocked, whereas this would not happen for the coloured system (but of course, B then would not be strictly speaking a top−level independent action).

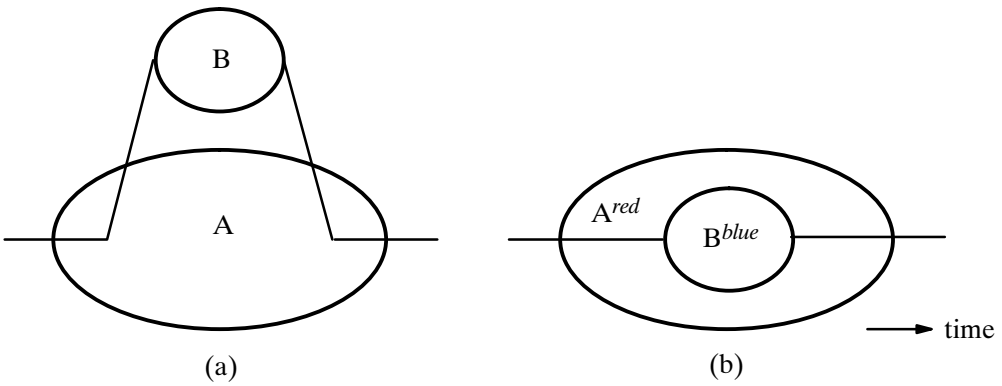

Fig. 13. Implementing top−level independent actions

5.6. Further Observations

Coloured actions provide a powerful yet simple technique for constructing arbitrarily complex action structures, and not just the structures discussed in section 3. For example, they can be used quite easily to implement a generalization of top−level independent actions, giving rise to *n−level independent actions*. Such actions are illustrated in fig. 14, where C and F are top−level independent and E a second level independent action. If A aborts, any effects of D, B and E will be undone; on the other hand if B aborts after invoking E, the effects of E will not be undone. Such a system can be implemented using coloured actions using the same principle employed for the construction of top−level independent actions. Fig. 15 shows the colouring scheme which will implement the action structure of fig. 14.

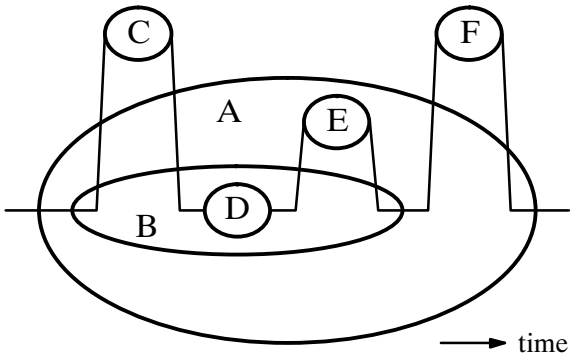

Fig. 14. N−level independent actions

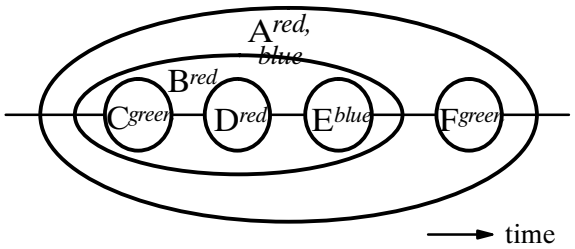

Fig. 15. Implementing n−level independent actions

6. Concluding Remarks

Coloured actions provide a remarkably powerful yet easy to implement technique for controlling concurrency and recovery properties of a system of (top−level) actions. The locking rules of coloured actions require minor modifications to the 'conventional' rules. In object−oriented systems, such as *Arjuna* [12], which permit type specific locking, such modifications are relatively straightforward to incorporate. The approach that we are adopting in our research is to let the application builder think in terms of the action structures of section 3 and to generate colour assignments automatically, thus ensuring that coloured actions are used in a controlled manner. A trial, non−distributed version in *Arjuna* has been implemented to test our ideas [10]. This exercise has given us enough confidence to embark on building a distributed version and then to evaluate the merits of the scheme by implementing some of the applications discussed earlier.

During the course of our research we came across some recent work which deserves a brief mention here. The split−transaction structure for database systems proposed recently [13] bears some resemblance to glued actions. This work indicates that we are not alone in thinking along the lines suggested in section 3. The *Acta* model of transactions [14] is an attempt to develop a unified framework for precisely specifying the salient features of action structures. Such a model may well be used for comparing and contrasting action structures proposed here and elsewhere.

Acknowledgements

This work has been supported in part by grants from the UK Science and Engineering Research Council and ESPRIT project 2267 (Integrated Systems Architecture). Discussions

with Graham Parrington, Graeme Dixon, Irvin Shizgal, Krithi Ramamritham and John Warne have been helpful in presenting the ideas put forward here.

References

[1] B. Liskov and R. Scheifler, *"Guardians and actions: linguistic support for robust distributed programs"*, ACM TOPLAS, Vol. 5, No. 3, July 1983, pp. 381−404.

[2] K.P. Eswaren et al., *"On the Notions of Consistency and Predicate Locks in a Database System"*, Communications of the ACM, Vol. 19, No. 11, pp. 624−633, 1976.

[3] P.A. Bernstein, V. Hadzilacos, and N. Goodman, *"Concurrency Control and Recovery in Database Systems"*, Addison−Wesley, 1987.

[4] P.M. Schwarz and A.Z. Spector, *"Synchronizing Shared Abstract Types"*, ACM Transactions on Computer Systems, Vol. 2, No. 3, pp. 223−250, August 1984.

[5] G.D. Parrington and S.K. Shrivastava, *"Implementing Concurrency Control in Reliable Distributed Object−Oriented Systems"*, Proc. of ECOOP88, European Conf. on Objected Oriented Programming, Norway, August 1988, Lecture Notes in Computer Science, Vol. 331, pp. 209−227.

[6] J.N. Gray, *"The transaction concept: virtues and limitations"*, Proc. of 7th VLDB Conf., September 1981, pp. 144−154.

[7] D.J. Taylor, *"How big can an atomic action be?"*, Proc. of the Fifth Symp. on Reliability in Distributed Software and Database Systems, Los Angeles, January 1986, pp. 121−124.

[8] C.T. Davies, *"Data processing spheres of control"*, IBM Systems Journal, Vol 17, No. 2, 1978, pp. 179−198.

[9] S.K. Shrivastava, *"A dependency, commitment and recovery model for atomic actions"*, Proc. of the Second Symp. on Reliability in Distributed Software and Database Systems, Pittsburgh, July 1982, pp.112−119.

[10] S.M. Wheater, *"Constructing reliable distributed applications using actions and objects"*, PhD Thesis (forthcoming), Computing Laboratory, University of Newcastle upon Tyne.

[11] J.E.B. Moss,*"Nested transactions: an approach to reliable distributed computing",* PhD Thesis 260, Massachusetts Institute of Technology, Cambridge, MA, April 1981.

[12] S.K. Shrivastava, G.N. Dixon and G.D. Parrington, *"An overview of Arjuna: a programming system for reliable distributed computing"*, IEEE Software (to appear).

[13] C. Pu, G. Kaiser and N. Hutchinson, *"Split−transactions for open−ended activities"*, Proc. of 14th VLDB Conf., Los Angeles, September 1988, pp. 26−37.

[14] P.K. Chrysanthis and K. Ramamritham, *"Capturing the structure and behaviour of complex transactions"*, Proc. of Third Workshop on Large Grained Parallelism, SEI, Carnegie Mellon University, Pittsburgh, October 1989.