University of Newcastle upon Tyne

COMPUTING LABORATORY

Constructing Reliable Distributed Applications using Actions and Objects

S.M. Wheater

TECHNICAL REPORT SERIES
No 316 June, 1990

Bibliographical details

WHEATER, Stuart Mark

Constructing reliable distributed applications using actions and objects. [By] S.M. Wheater

Newcastle upon Tyne: University of Newcastle upon Tyne: Computing Laboratory, 1990.

(University of Newcastle upon Tyne, Computing Laboratory, Technical Report Series, no. 316)

Added entries

UNIVERSITY OF NEWCASTLE UPON TYNE.
Computing Laboratory. Technical Report Series. 316

Abstract

A computation model for distributed systems which has found widespread acceptance is that of atomic actions (atomic transactions) controlling operations on persistent objects. Much current research work is oriented towards the design and implementation of distributed systems supporting such an object and action model. However, little work has been done to investigate the suitability of such a model for building reliable distributed systems. Atomic actions have many properties which are desirable when constructing reliable distributed applications, but these same properties can also prove to be obstructive.

This thesis examines the suitability of atomic actions for building reliable distributed applications. Several new structuring techniques are proposed, providing more flexibility than hitherto possible for building a large class of applications. The proposed new structuring techniques are: Serialising Actions, Top-Level Independent Actions, N-Level Independent Actions, Common Actions and Glued Actions.

A new generic form of action is also proposed, the Coloured Action, which provides more control over concurrency and recovery than traditional actions. It will be shown that Coloured Actions provide a uniform mechanism for implementing most of the new structuring techniques, and at the same time are no harder to implement than normal actions. Thus this proposal is of practical importance.

The suitability of the new structuring techniques will be demonstrated by considering a number of applications. It will be shown that the proposed techniques provide natural tools for composing distributed applications.

About the author

Dr. Wheater is a Research Associate in the Computing Laboratory.

Suggested keywords

ATOMIC ACTIONS	LONG RUNNING TRANSACTIONS
COLOURED ACTIONS	PERSISTENT OBJECTS
CONCURRENCY	RECOVERY
DISTRIBUTED SYSTEMS	RELIABILITY

Suggested classmarks (primary classmark underlined)

Dewey (18th):	001.64404	001.6425
U.D.C.	519.687	681.322.06

Constructing Reliable Distributed Applications using Actions and Objects

by

Stuart M. Wheater

Ph.D. Thesis

September 1989

The University of Newcastle upon Tyne

Computing Laboratory

Abstract

A computation model for distributed systems which has found widespread acceptance is that of atomic actions (atomic transactions) controlling operations on persistent objects. Much current research work is oriented towards the design and implementation of distributed systems supporting such an object and action model. However, little work has been done to investigate the suitability of such a model for building reliable distributed systems. Atomic actions have many properties which are desirable when constructing reliable distributed applications, but these same properties can also prove to be obstructive.

This thesis examines the suitability of atomic actions for building reliable distributed applications. Several new structuring techniques are proposed, providing more flexibility than hitherto possible for building a large class of applications. The proposed new structuring techniques are: Serialising Actions, Top–Level Independent Actions, N–Level Independent Actions, Common Actions and Glued Actions.

A new generic form of action is also proposed, the Coloured Action, which provides more control over concurrency and recovery than traditional actions. It will be shown that Coloured Actions provide a uniform mechanism for implementing most of the new structuring techniques, and at the same time are no harder to implement than normal actions. Thus this proposal is of practical importance.

The suitability of the new structuring techniques will be demonstrated by considering a number of applications. It will be shown that the proposed techniques provide natural tools for composing distributed applications.

Acknowledgments

Firstly I would like to thank my supervisor, Professor Santosh Shrivastava, who suggested this area of research. I would also wish to thank him, Dan McCue and Dr. Lindsay Marshall for reading and commenting upon the numerous drafts of this thesis. Their efforts are greatly appreciated.

I would also like to thank my fellow members of the Arjuna project, in particular Dr. Graham Parrington and Mark Little, and several staff members of the Computing Laboratory for many useful comments and discussions I have had on this work.

Finally I would like to thank my family for their support and encouragement, which they gave me during my studies.

Financial support for much of the work described in this thesis was provided by grants from the Science and Engineering Research Council.

Table of Contents

1 Introduction

With the increasing availability of powerful workstations and high performance local area networks, there has been a corresponding increase in the use of distributed systems. Distributed systems make possible the sharing of the available resources on the network between applications. This has caused a need for developing techniques and methodologies for constructing reliable distributed systems. Distribution can be an advantage when constructing reliable systems, for example, if distributed systems need to be available for a long time, then they can be constructed to take advantage of the fact that the replication of resources with independent failure modes can be used to tolerate failures. This is possible because the failure of a computer on a network is unlikely to cause the failure of other computers on the network, or failure of the network. However, distributed systems introduce additional possible failures which can occur such as: message loss, server process failure, remote machine crash and network partitioning. These failures cause problems when trying to maintain consistency within a distributed system, and can increase the complexities of the reliability mechanisms and techniques required.

One proposed technique for the construct reliable applications is the use of atomic actions. This technique tackles both the problems of inconsistency due to partial failure, and interference of concurrent parts of the applications, although when constructing systems from atomic action, the restrictions which atomic actions impose, may give rise to unreasonable restriction on how the applications can be constructed. This thesis will describe how reliable distributed applications can be constructed using atomic actions, along with the new structuring techniques which overcome some of the restrictions on how the applications can be constructed.

1.1 Fault tolerance

A fault-tolerant system is one that is designed to fulfil its desired purpose despite the occurrence of component failures. The system must detect errors produced by faults in the hardware or software of the system, and use error recovery mechanisms to convert the system state to one from which normal progress can be continued.

The precise definitions of error, fault and failure used here can be found in [Anderson 81]; In brief the definitions are that, an *error* is a part of an erroneous state. An *error* in a component or the design of a system will be referred to as a *fault* in the system. A *fault* is a cause of a *failure*. A *failure* occurs when a system's behaviour does not conform with the specification. Component failure within a system can cause the system's internal state to become erroneous such that further processing can lead to failure of the system.

A fault-tolerant system should include mechanisms to allow:

1) Error detection: for a system to be fault-tolerant it is necessary that the system should be able to detect errors.

2) Damage confinement and assessment: attempts should be made to prevent faults in one part of the system from causing errors in other parts of the system.

3) Error recovery: these mechanisms aim to change the erroneous system state to one from which normal progress of the system may continue.

4) Fault treatment and continued service: although the error recovery mechanism may have changed the system state to one from which normal progress of the system can continue, mechanisms may be needed to allow the system to fulfil the purpose for which it was constructed, despite the presence of the fault.

In the following four sections the above mechanisms will be explained in more detail.

1.1.1 Error detection

Error detection is the start of the fault tolerance process. The error detection mechanism must be able to detect errors caused by faults in the system. The detection of an error may result in an exception being raised, the type of exception raised depending on the error detected, for example, division by zero, memory access violation, or violation of array range bounds.

1.1.2 Damage confinement and assessment

Due to the delay between the manifestation of a fault causing the system state to become erroneous and the detection of the error, it is possible for erroneous data to have been passed to other parts of the system. This means that these parts of the system could contain undetected errors. The purpose of damage confinement mechanisms is to reduce the spread of erroneous data. For example, operating systems ensure that the address spaces of processes are kept isolated from each other, so preventing an error in one process's address space spreading to that of other processes. Damage assessment mechanisms are used to determine which parts of the system may contain errors produced by the fault.

1.1.3 Error recovery

Error recovery techniques are used to change the system state to one from which continued progress can be made, after an error has been detected. The error recovery operation attempted may depend on the type of exception raised by the error detection mechanism. There are two main error recovery techniques: forward error recovery and backward error recovery [Campbell 86].

Forward error recovery attempts to compensate for errors in the system, by changing the current inconsistent system state so as to remove particular errors, so that normal progress can be continued. This needs detailed knowledge of the consequences of the errors; this means that forward error recovery is application dependent.

Backward error recovery attempts to restore the system state to the state before the failure affected the system, so that normal progress can be continued from that earlier point in time. Backward error recovery can be used for coping with errors caused by unexpected failures, and where the extent of the errors produced by failures is unknown; this means that backward error recovery is application independent. Backward error recovery may involve large overheads due to the need to store previous states of the system or the changes made since those points in time.

1.1.4 Fault treatment and continued service

If the error recovery mechanism has worked, the system should be in a state which contains no errors, but the fault may remain in the system. So, to allow further progress, it is necessary to isolate the faulty component, preventing it from causing further errors.

1.1.5 Recovery blocks

To illustrate the concepts described in the previous sections, the *recovery block mechanism* [Horning 74][Randell 75], a method of constructing fault-tolerant software, will be examined. The syntax of a recovery block is given below.

```
ensure <acceptance test> by P0 else-by P1 else fail ;
```

In the example of the recovery block, P_0 and P_1 are procedures, the primary and alternative, respectively. The algorithm of the system is that P_0 is first executed, then the *acceptance test* is evaluated. If the test returns true, statement following the semi-colon will be executed, if the test is not passed, the state is recovered to the state at the beginning of

the recovery block, then P_1 is executed and the acceptance test is re–evaluated. If the test is not passed, the state is recovered as before and the fail exception is signalled.

The acceptance test is used for *detecting error*. If the acceptance test is "perfect", meaning it is able to detect all deviations from the recovery block specification, then the recovery block can detect any errors which may occur in either P_0 or P_1.

Damage confinement and assessment is simple when using recovery blocks because it is assumed that only the presently executing component (P_0 or P_1) is affected.

Error recovery is obtained using backward error recovery; the state is recovered to the state at the beginning of the recovery block.

Fault treatment and continued service is provided by the ability to specify an alternative component to the executed, if the primary component fails the acceptance test.

1.2 Distributed systems

In a distributed system the execution of programs may involve access to resources on other machines, and threads of control may pass between machines.

There are many possible reasons to construct a distributed system. For example: the need for higher performance, a requirement for higher reliability, or simply because of the nature of the application.

Distributed systems have the potential for providing higher performance than a conventional centralised system. Also, if additional performance is required, it can be provided by simply adding more machines to the system. Some centralised systems are simple to upgrade to provided the additional performance, but not to the potential scale which which distributed systems can provide.

Applications which are required to have a high degree of reliability, are unsuitable for execution on a single machine if the degree of reliability required is greater than the reliability of the machine itself. The use of a distributed system permits the exploitation of replication of system components, which enables applications to be constructed which are more reliable than the individual machines on which they are executed.

Some applications such as inter–network electronic mail, and factory automation are inherently distributed in nature.

1.2.1 Modeling failures in distributed systems

The model of distributed systems assumed here consists of nodes (machines) connected by a network. Nodes can communicate with one another by sending messages over the network.

A node is modelled as comprising a processor, volatile storage and stable storage. It is assumed that nodes only fail in one way, that is to crash (The fail–stop assumption [Schlichting 83]). If a node crashes, any information held in volatile storage will be lost, while information in stable storage will survive the crash. A crashed node is assumed not to produce any messages. When a node crashes it is assumed that after a finite amount of time the node will be repaired and then restarted.

The network may fail in many ways. It may: lose messages, corrupt messages, delay message delivery, replicate messages, deliver messages out of order or may partition (where certain groups of nodes are unable to communicate with each other for long periods of time). Message loss, corruption, delay, replication and delivery out of order, are failures for which there are well known fault–tolerance techniques which can be used in the message protocols [Tanenbaum 88]. It will be assumed that those error "do not occur" at the application level.

It will also be assumed that network partitions cannot be distinguished from node crashes until the network resumes correct operation.

1.2.2 Server - client model

One model for the construction of distributed applications is the server – client model. In this model a server process provides a particular service (for example, the management of a shared resource). The server process will receive, from client processes, requests for operations to be performed. The operations are part of the service provided by the server process.

There are two main techniques for implementing the interaction between client and server processes: the client process making a request is blocked until the server process has completed the request, or the client process may continue immediately after issuing the request and receive the results later in its computation. Therefore a client can receive results either synchronously or asynchronously. The synchronous strategy is often referred to as a remote procedure call (RPC) [Nelson 81]. In practice, the RPC model has been found to be adequate for many systems [Liskov 84] [Liskov 88] [LeBlanc 85] [Warker 83] [Warker 85] [XEROX 81] [Cooper 83], and is simpler to use and understand, than the asynchronous strategy.

1.3 Object oriented programming

The basic concept in object oriented programming languages is support of the abstraction of *objects*. An object is an entity which has an internal state, and the internal state can only can be manipulated using a set of operations provided by the object. Objects provide a form of *data abstraction* in that the internal state of the object, and the implementation of the set of operations need not concern the users of the object; they need only understand the behaviour of its operations. In most object oriented programming languages, the definition of objects is given using the *class* construct. Thus objects are instances of the corresponding classes.

A powerful concept provided in many object oriented programming languages is that of *inheritance*. Inheritance allows the creation of a new class of objects which are either a refinement or embellishment on an existing class of objects.

Many object oriented programming languages have been implemented such as: Smalltalk [Goldberg 83], C++[Stroustrup 86] [Lippman 89], Clu [Liskov 79], Trellis/Owl [Schaffert 86], Simula–67 [Dahl 70] [Birwhistle 73] and Eiffel [Meyer 88]. In the next section, one such object oriented programming language (C++) will be focused upon.

1.3.1 C++ programming language

This section will give a brief description of some of the C++ language features, along with an explanation of some C++ terminology. The language features description in this section will be used in section 3, when describing the Arjuna system.

C++ [Stroustrup 86] [Lippman 89] is an object oriented programming language, based on the programming language C [Kernighan 78]. C++ provides extensions to C to support programming in an object oriented manner. Objects in C++ are instances of a *class*; the class of an object defines the operations associated with the object and the variables that the object contains. These operations and variables are referred to as the *members* of the class. The class also defines the access allowed to these members, of which there are three types: *public*, *private* and *protected*. Public access means that the member is accessible to both members and non–members of the class. Private access means that the member is accessible only to other members of the class. Protected access means that the member is accessible to other members of the class and to an inheriting class's members.

Below is an example of the class interface for a class called *Stack*, containing private and public members. The class has private members *Top* and *Elems* (integer and integer

array respectively), and public members *Stack()*, *Stack(int MaxTop)* (the "constructors"), *~Stack()* (the "destructor"), and the operations *Push* and *Pop*.

```
class Stack
{
public :
        Stack() ;
        Stack(int MaxTop) ;
        ~Stack() ;

        int Push(int Elem) ;
        int Pop (int *Elem) ;

private :
        int    Top ;
        int    Elems[150] ;
} ;
```

C++ allows the class implementor to provide operations which can be used to initialise objects. These operations are called *constructors*. When an object is created the appropriate *constructor* is invoked to initialise the object. To allow orderly release of resources when objects are destroyed, an operation called a *destructor* can be provided; this is invoked automatically when an object goes out of scope.

Below is an example of the creation of instances of a class (*Stack*). The two instances of the class will be accessed using the names *a* and *b*, object *a* being initialised with the *Stack()* constructor, and object *b* being initialised with the *Stack(int MaxTop)* constructor.

```
Stack a, b(10) ;
```

In C++, the operation to be performed on an object is specified using a "dot" notation. Examples of the invocation of operations on objects are given below:

```
int err, val ;

err = a.Push(10) ;        // Push the value, 10 onto the stack a.
err = b.Push(30) ;        // Push the value, 30 onto the stack b.
err = a.Pop(&val) ;       // Pop the top element of stack a into
                          // the integer variable, val.
```

Inheritance is used to produce classes of objects which have the refined properties of some other class of objects. A *base class* is said to be *inherited* by the *derived class*. There are two types of inheritance in C++ *public* and *private*. In public inheritance the public and protected members of the base class are accessible to the members of the derived class and the public members of the base class are also accessible via the objects of the derived class. In private inheritance the public and protected members of the base class are accessible to the members of the derived class but the public members of the base class are not accessible via the objects of the derived class.

The two example class interfaces below are of private and public inheritance, respectively.

```
class DerivedClass1 : private BaseClass1
{
        . . .
} ;

class DerivedClass2 : public  BaseClass2
{
        . . .
} ;
```

The use of *virtual operations* in C++ allows the operations of a deriving class to be invoked by the operations of its base class. If a virtual operation of the base class has an

equivalent operation in the derived class then the operation in the derived class will be invoked instead of the operation in the base class.

Virtual operations allow a derived class to redefine the behaviour of it base class. For example, if a base class *employee* has a virtual operation *pay* which calculates the pay of a standard employee using a standard pay scale, it would be possible to derive a *manager* class which provides an alternative implementation of the *pay* operation, which uses an alternative pay scale. This alternative implementation will be invoked instead of the base class's *pay* operation, when the *pay* operation is invoked on instances of the *manager* class.

1.3.2 Distributed object oriented systems

Object oriented programming has advantages when constructing distributed systems, because objects provide convenient interfaces for services, and objects are a natural way to represent local or remote resources.

The server – client model is a suitable basis for an object oriented distributed system. The client may perform RPC to the server to invoke the operations of the object(s) which the server implements. The object also forms a natural unit of replication, so allowing the availability of the service provided by an object to be increased. To ensure the consistency of the set of objects in a system, the operations of the objects can be performed as (and from within) *atomic actions* [Lomet 77] [Gray 78]. An *atomic action* is a mechanism by which the consistency of the objects in the system can be maintained despite the concurrent invocation of object operations and failure of components within the system.

1.4 Contributions of the Thesis

It has been observed that, "The methodology of programming with actions and objects is not at present well understood compared with experience in programming with objects" [LeBlanc 85]. I have been studying how reliable applications should be constructed using actions operating on objects. I began by first investigating how a number of example applications could be constructed using atomic actions. This helped to indicate the problems involved in the use of atomic actions, and how atomic actions should be used to construct other reliable applications. As a result of these studies I have identified new structuring techniques for the construction of reliable applications. These new structuring techniques are: serialising actions, top–level independent actions, n–level independent actions, common actions, and glued actions. In addition to these structuring techniques, I propose a uniform mechanism by which most of the structuring techniques can be implemented. This mechanism is based on the use of "coloured actions". Coloured actions are a generalisation of atomic actions which separates serialisability from atomicity, so allowing their use in a more flexible manner.

1.5 Structure of the Thesis

Relevant work on the subject of constructing reliable distributed applications will be discussed in section 2. The Arjuna object oriented system which forms the basis of the experimental work is described in section 3. A reliable distributed application, constructed using the Arjuna system, is examined in section 4. New structuring techniques for atomic action systems are developed in section 5. Ideas behind coloured action and locks are described in section 6. A possible implementation for coloured actions is described in section 7. An examination of three distributed applications (make, spreadsheet and meeting arranger) is presented in section 8, and the conclusions arising from this work are given in section 9.

1.6 Working environment

Most of the work reported here has been carried out as a part of the Arjuna research project, a group effort concerned with building a fault–tolerant distributed system. The

construction of an operational distributed application, using the Arjuna system, is described in sections 4. This experience gave me ideas on more general techniques for building applications using coloured actions. A simple implementation of coloured actions has been performed to test the feasibility of the concepts. A fully-fledged distributed implementation will be undertaken in the next phase of the Arjuna project.

2 Constructing reliable applications

In this section, techniques which have been used or proposed for the construction of reliable applications will be described. These techniques are designed to tackle the problems of inconsistency due to partial failure, and interference between concurrent parts of the applications.

One technique which has integrated the solutions to the above problems into a single mechanism is the use of atomic actions. This will be examined in some detail in this section, because atomic actions form the basis for the work which will be described in subsequent sections. Other approaches are summarised at the end of the section.

2.1 Atomic actions

Atomic actions have been proposed as a technique for constructing reliable distributed applications, and some systems which support atomic actions have been implemented [Liskov 84] [Marshall 80] [Nett 85] [Allchin 83] [Spector 87]. In this section the properties of atomic actions will be discussed, along with a method for constructing reliable applications using atomic actions.

2.1.1 Atomic action properties

The properties of atomic actions have been defined in many ways. The definitions of their properties used here, are the following:

1) Failure atomicity: an atomic action on completion either has performed the desired effects or has had no effect at all.
2) Serialisability: the concurrent execution of atomic actions should produce the same effects as some serial order execution of the atomic actions.
3) Permanence of effect: once an atomic action has completed successfully, all changes it has made to the system state become permanent.

There are two ways of modeling atomic actions: one where atomic actions operate on passive objects (this model has been used in database systems, banking and airline reservation) the other where concurrent processes exchange messages (this model has been used in process control, avionics and telephone switching systems). The relationship between the two models is examined in [Shrivastava 87].

In the following three sections the above properties of atomic actions will be explained in more detail with respect to actions and objects model.

2.1.1.1 Failure atomicity

An action is said to be failure atomic if, when it fails automatic backward error recovery will be performed on the objects which the action has changed. If objects were in consistent states before the action began then recovery will solve the problem of maintaining the consistency of objects after a failure, since the original states will be restored.

Backward error recovery can be performed by either restoring the state to the state at the beginning of the action, or by performing some compensating action which transforms the state to the state at the beginning of the action. For example, if the atomic actions had increased *i* by 5, the compensating action could decrease i by 5.

2.1.1.2 Permanence of effect

Permanence of effect means that once an atomic action has completed successfully, all changes it has made to the system state become permanent. This ensures that if an atomic action has *committed*, its effects will survive subsequent failures.

2.1.1.3 Serialisability

Serialisability is the system structuring mechanism which is used to control the concurrency in the system. Serialisability is a property of a concurrent execution of atomic actions such that, "the execution of concurrent atomic actions should produce the same effects as some serial order execution of the atomic actions". This means concurrent atomic actions are not permitted mutually to interfere or affect each other's results in any way. It also means that atomic actions see a consistent system state throughout their lifetime.

While several schemes have been proposed for enforcing serialisability [Moss 81] [Schwarz 84] [Eswaran 76], one of the most common schemes involves the locking of the objects used by the atomic actions. The locks could be obtained on all the objects needed by the atomic action when started, and all the locks could be released when the top–level atomic action committed. If this scheme is used, the resulting lock conflicts will reduce the concurrency in the system. To reduce the frequency of lock conflicts and so increase concurrency in the system three techniques have been proposed: late acquisition of locks, early release of locks and optimistic concurrency control.

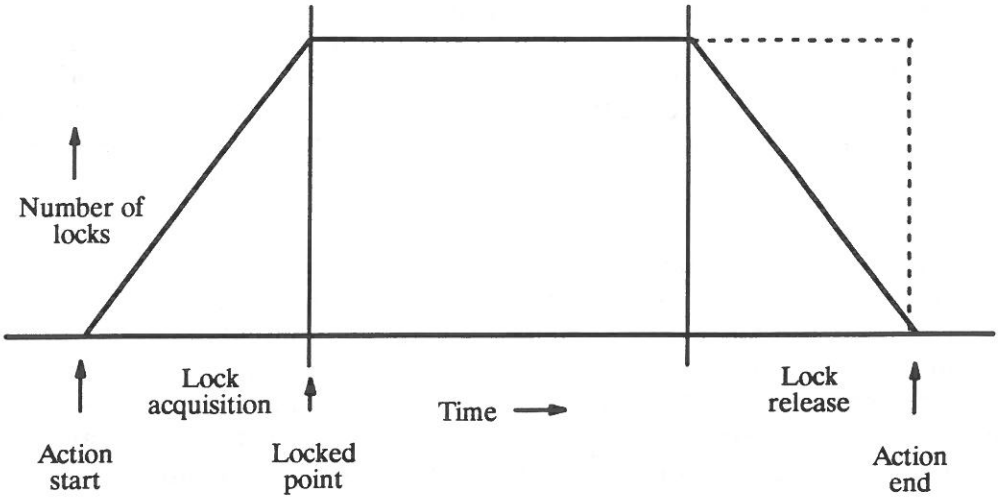
The late acquisition of locks involves waiting until the objects are needed before attempting to lock them. This means that deadlock is possible between atomic actions. If deadlock occurs one of the involved atomic actions will be aborted, so causing at lease one of the locks involved in the deadlock to be released. The atomic action to be aborted should be selected in a consistent manner. A common method is to abort the youngest of the atomic actions involved in the deadlock [Rosenkrantz 78].

The early release of locks involves releasing the locks on an object when no more use is to be made, of the object. This may cause problems if an atomic action, *A*, fails after it has released some locks, since if another atomic action, *B*, has acquired one of the locks released by *A*, then atomic action *B* will be using an object which could be in an inconsistent state, therefore *B* must be aborted to allow recovery of the object. If the atomic action which acquired the lock on the inconsistent object has also released locks this will mean that other atomic actions may have to be aborted, this could cause a cascade of atomic action abortions.

Most systems which use atomic actions have adopted a two–phase locking scheme [Eswaran 76] [Moss 81] to ensure serialisability. This can involve both the late acquisition of locks and the early release of locks. Using a two–phase locking scheme, no new locks can be acquired after any lock has been released. This results in two distinct phases, a growing phase where locks are being acquired and a shrinking phase where locks are being released. The point where the atomic action has obtained all the locks it requires is called the *locked point* (also sometimes referred to as the *lock point*).

Figure 2–1 shows the number of locks held by an atomic action over time. The continuous line represents an atomic action using a two–phase locking scheme, and the dotted line represents an atomic action using a strict two–phase locking scheme, where all locks are released together.

Figure 2–1



Optimistic concurrency control [Kung 81] does not involve locks. Instead, atomic actions maintain lists of objects they have used. When committing, the lists are checked to see if any of the atomic actions have interfered with one another, those which have are aborted and must then be retried. When the frequency of conflicts is low, this approach results in higher levels of concurrency than two–phase locking. However, as the rate of conflicts increases, this approach degrades seriously.

2.1.2 Nested atomic actions

Given that a system provides atomic actions which perform certain operations, it is sometimes necessary to combine them to form another operation, which is also required to be an atomic action [Moss 81]. The resulting atomic action's effects being some combination of the effects of the atomic actions from which it is made. The atomic actions which are contained within the resulting atomic action are said to be *nested*, and the resulting action is referred to as the *enclosing* atomic action. The enclosing atomic action is sometimes referred to as the *parent* of a *nested* or *child* atomic action. The outermost atomic action in a system is called the *top–level atomic action*.

The effect of a nested atomic action will be recovered if the enclosing atomic action aborts, even if the nested atomic action has committed.

To maintain serialisability, the locks obtained by nested atomic actions which commit must be *inherited* by their enclosing atomic action.

Figures 2–2 and 2–3 show pictorial representations of enclosing atomic actions which contain two nested actions. In figure 2–2 the enclosing atomic action, *B*, has nested within it two atomic actions, *A* and *C*, and the atomic action *A* must complete before atomic action *C* starts. In figure 2–3 the enclosing atomic action, *B*, has nested within it two atomic actions, *A* and *C*, and the atomic actions, *A* and *C*, are executed concurrently.

Figure 2–2

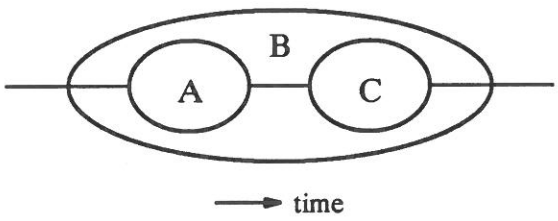
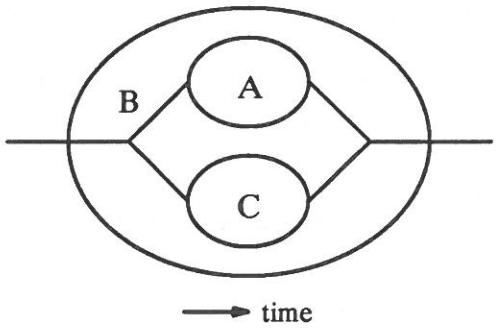


Figure 2–3



2.1.3 Constructing reliable applications using atomic actions

The atomic actions and objects model provides a natural environment for the construction of reliable applications with persistent objects. The persistent objects are normally resident in a store object, which is stable. Since the operations which are performed on the objects are atomic actions, only consistent state transformations take place, so guaranteeing their consistency. Using objects, it is also possible to increase further the reliability of an application, by increasing the availability of the objects on which the application depends. This can be done by replicating objects on different nodes. To ensure that component objects of a replicated object remain mutually consistent an appropriate *replica–consistency protocol* must be employed.

If atomic actions are to be used for structuring reliable applications, then they must be able to provide mechanisms for: error detection, damage confinement, damage assessment, error recovery, and fault treatment and continued service.

Error detection could be done by using acceptance tests just before atomic actions are committed. These would check the effects of an atomic action, making sure that acceptable results would be produced. Alternatively, objects used by an atomic action could raise exceptions if they find themselves in an inconsistent state.

Atomic actions provide excellent facilities for damage confinement and assessment. Because of the property of serialisability, the interaction between different parts of the system is very restricted, limiting the extent to which damage to the system state could have occurred. If serialisability is obtained using strict two–phase locking, the effects of an atomic action become visible only after the atomic action has committed. The set of objects write–locked by an atomic action also provides information on the parts of the system which it could have changed.

Atomic actions usually provide backward error recovery, but exception handling for forward recovery could also be used. Exception handling poses many problems when used in conjunction with atomic actions, for example: how should exceptions be handled in an atomic action which contains other atomic actions which have not been completed and how should simultaneous raising of exceptions be handled. Exception handling will be discussed briefly in section 2.2.7.

Fault treatment and continued service using atomic actions could be provided by the use of alternative atomic actions that can be used to obtain the desired effects if an attempted atomic action fails. If all the alternatives of a nested atomic action fail an exception could be raised in the enclosing atomic action, which could be handled by executing an alternative to the enclosing atomic action.

2.1.4 Node failure tolerance using atomic actions

The use of atomic actions enables the construction of applications which can tolerate the failure of the nodes on which part or all of the system of atomic actions are operating. This is particularly important when reliable distributed applications are to be constructed.

The degree to which atomic actions can continue to make forward progress in the event of node crashes will depend on the implementation of the atomic actions. The more information which the atomic actions store on stable storage, the more crash resistant the system will be. In most current systems supporting atomic actions, only the effects of top–level atomic actions are stored on stable storage. This is because accessing stable storage is relatively slow. This is satisfactory if top–level atomic actions are short, but can cause problems if atomic actions last for days, weeks or months. For such a long running atomic action, it becomes likely that a node on which it is depending will crash during its execution.

There are four ways in which the crashing of a node would affect a top–level atomic action:

1) If that node were executing a nested atomic action of the atomic action.
2) If that node had executed a nested atomic action of the atomic action.
3) If that node were to execute a nested atomic action of the atomic action.
4) If that node were executing the atomic action.

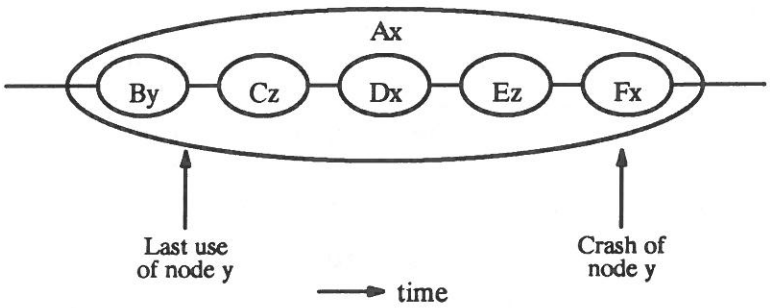
Of course a top–level atomic action may be affected in more than one of these ways by the crash of a particular node.

If a crashed node were executing a nested atomic action, the failure atomicity property of atomic actions would ensure that the node on which the nested atomic action was being executed would still be in a consistent system state, when the node was restarted. The state would be recovered to the state of the system before the atomic action was started. The atomic action which invoked the nested atomic action could retry the atomic action.

If a nested atomic action had been executed on a node where its effects were stored in volatile storage, and the node on which it was executing crashed, the effects of the atomic action would be lost. The enclosing atomic action could not commit. This could cause great problems where long running atomic actions are involved. This problem can be solved by allowing some nested atomic actions to make their effects persistent (the effects are stored in stable storage, but can be recovered by the aborting of an enclosing atomic action) but not permanent (the effects are stored in stable storage, but will not be recovered by the aborting of any atomic action). The top–level atomic action could then commit even if the nodes on which such nested atomic actions had been executed had crashed. When these nodes came up, they would discover that the top–level atomic action had committed and make the nested atomic actions' effects permanent.

This concept is illustrated in figure 2–4. The atomic action, *A*, executing on node *x*, invokes a nested atomic action, *B*, on node *y*. The nested atomic action commits. But later in the execution of atomic action, *A* node *y* crashes losing all the effects of atomic action *B*, and preventing atomic action *A* from committing.

Figure 2–4



If a node crashes which was to execute a nested atomic action of the top–level atomic action, then the top–level atomic action will only be able to proceed before the node

restarts if the resource needed on that node is replicated on other nodes. Keeping such replicated resources consistent may involve much effort; all modifications to the resource need to be done to all copies.

If the node which is executing the top–level atomic action crashes, then all the effects of the top–level will need to be recovered, thus involving recovery of effects on other nodes. This would be very undesirable for a long running atomic action. This could be made less of a problem if long running atomic actions could be checkpointed, so when the node on which a long running atomic action was running restarts, it could be continued from the last checkpoint (this could involve a small amount of recovery).

2.2 Methodologies for the use of atomic actions

Despite the implementation of many systems that support of atomic actions [Liskov 84] [Nett 85] [Allchin 83] [Spector 87], little work has been done on developing methodologies for their use in applications. There is little knowledge of where atomic actions should be used and how applications should be structured with them. Some additions to the basic concept of atomic actions have been suggested to allow easier construction of systems using atomic actions.

In the past, atomic actions have been used for performing compact and elementary operations, whose effects on the system state are small, for example, money transfer between bank accounts, and airline seat reservation. These operations are suitable of implementation as atomic actions because, in the case of failure, it is acceptable for them to produce no effect on the system.

The following sections describe atomic actions methodologies which address some of the problems which arise when constructing applications. These problems are caused by the potentially over–restrictive properties of atomic actions [Taylor 86]. For many applications the properties of *serialisability* and *failure atomicity*, which atomic actions possess, will restrict respectively: the possible concurrency within the application, and the intermediate effects produced by the application which can be made permanent.

2.2.1 Nested top–level atomic actions in Argus

It has been observed by Liskov [Liskov 84] that it is sometimes desirable for a top–level atomic action to be invoked from within another atomic action. A top–level atomic action invoked in this way is referred to as a *nested top–level atomic action*. The purpose of nested top–level atomic actions is to avoid inefficiency due to unwanted recovery of nested atomic actions caused by the abortion of their parent action. This is possible because nested top–level atomic actions can be used to perform "benevolent side effects". A nested top–level atomic action differs from a nested atomic action in that it cannot acquire from its parent atomic action any of the locks which its parent atomic action retains. Also, the committing of a nested top–level atomic action is not dependent on its parent atomic action. Its effects can be made permanent even if its parent atomic action aborts. It is not clear if it is intended that the parent of a nested top–level atomic action be blocked until the nested top–level atomic action commits or whether the parent atomic action can commit before the nested top–level atomic action. An example of a name server is given which speeds up its look–ups by copying information from remote nodes to its local node. The use of a nested top–level atomic action to copy the information ensures that the change is permanent, even if the parent atomic action aborts, so making it unnecessary for this operation to be performed again in the future.

The use of nested top–level atomic actions to perform "benevolent side effects" does mean that the invoking atomic action is not truly atomic, having caused effects which will not be recovered if the invoker aborts. In addition, nested top–level atomic actions could be used to mutually affect each other's progress, so breaking serialisability.

2.2.2 The split-transaction operation

A paper by Pu, Kaiser and Hutchinson [Pu 88] proposes a new structuring technique made possible the by use of the split-transaction operation. This operation can assist the construction of long running atomic action systems. The use of the split-transaction operation makes it possible for a transaction, A, to split into two transactions B and C. The split-transaction operation will divide all the objects accessed by the transaction A into two subsets $B_{Objects}$ and $C_{Objects}$. These sets are further divided into the objects which have been read by A, or will be read by the transactions, and the objects which have been updated by A, or will be updated by the transactions (for example, $B_{ReadObjects}$ and $B_{WriteObjects}$).

Let us assume that transaction B precedes C, and let the sets:

$B_{WriteObjects} \cap C_{WriteObjects} = C_{WriteObjectsLast}$
$B_{ReadObjects} \cap C_{WriteObjects} = 0$
$C_{ReadObjects} \cap B_{WriteObjects} = SharedSet$

If both the *SharedSet* and $C_{WriteObjectsLast}$ sets are empty, the transactions B and C are *independent*, in which case they can be serialised in either order, and the transactions B and C can commit (or abort) independently.

If either of the sets *SharedSet* or $C_{WriteObjectsLast}$ is not empty, the transactions B and C are *serial*, in which case transaction C must follow B because of the data access dependencies, and the transactions C most abort if the transactions B aborts.

The purpose of the split-transaction operation is to allow long transactions to make permanent the changes made to certain objects, and to allow the releasing of locks on these objects which are no longer required. The transaction A can achieve this by splitting into two transactions B and C, and delegating to B these objects; the transaction B would then immediately attempt to commit. If the transactions B and C are *independent*, the remaining transactions C would be able to commit even if B was unable to commit. Whereas if the transactions B and C are *serial*, the transaction C could only commit if B successfully committed.

One problem which arises from this scheme is that if transaction A used a nested transaction to perform an atomic operation on two objects b and c, it is possible that after A splits into two transactions B and C, that the changes made to b will be committed by B, while the changes made to c could be undone, due to C aborting.

A join-transaction operation is also proposed which allows a transaction to join a target transaction, so enabling the transactions to commit (or abort) the their result atomically.

2.2.3 Model for long running atomic action systems

A model of atomic actions has been put forward by Shrivastava [Shrivastava 82] based on work done by Davies [Davies 78] which can be used to aid the construction of long running atomic action systems. It is pointed out that one of the major problems with the use of long running atomic actions is that they may reduce the concurrency in the system by retaining locks on objects until they commit. The model allows an atomic action to acquire objects and later release new versions of these objects, before it commits. The objects acquired by an atomic action could therefore be from atomic actions which have not yet committed. So that it is known which objects are committable, each object has associated with it a committable status with indicates the probability that the object will become committable. Associated with each object acquired by an atomic action there is a degree of importance value; this is used to signify the importance that the atomic action gives to that object. This value can increase from its initial value during progress of the atomic action, but will not attain the highest value until the atomic action commits. The

committable statuses of the objects acquired by an atomic action used in conjunction with their degree of importance values, produces the committable status of new versions of the objects. This results in a system where atomic actions can acquire uncommitted objects and produce from them new versions of the objects which other atomic actions can acquire. These objects become committable when all their old versions become committed. An example of how this model of atomic actions can be used to construct a system to arrange meetings is given in the paper [Shrivastava 82], along with a discussion of the recovery management required.

This model of atomic actions would certainly permit acceptable levels of concurrency in a system of long running atomic actions, but no satisfactory techniques for implementing such a model have been proposed.

2.2.4 Profemo project

The Profemo distributed operating system kernel by Nett et al [Nett 85] provides support for nested transactions operating on shared data in a distributed object oriented environment.

The concurrency control used by the Profemo project uses two levels of synchronisation: outer synchronisation which is used to synchronise atomic actions and inner synchronisation which is used to synchronise the processes participating in an atomic action. The outer synchronisation is obtained by using a two–phase locking scheme, to enforce serialisability between atomic actions. Inner synchronisation ensures the internal consistency of objects by using a locking mechanism based on read/write semantics, allowing either several participating processes to perform read operations on an object or one participating process to perform a modifying operation on that object. The processes participating in an atomic action may only use locks held by that atomic action. This means that outer synchronisation is concerned with the acquisition of a lock and inner synchronisation is concerned with the use of a lock. This scheme allows atomic actions to be constructed out of many lightweight processes.

Because the concurrency control uses a two–phase locking scheme, the recovery management used by Profemo must use a so–called "chase protocol" [Merlin 78], to establish global recovery. The information used by the "chase protocol" is held in a recovery graph. The recovery graph is a distributed data structure and consists of the set of objects used by a transaction at a site (the nodes of the graph) and the dependencies between the set of objects used by a transaction at a site (the edges of the graph). From this recovery graph and knowledge of the effect of an error it is possible to work out a *recovery line* [Merlin 78]. For recovery from the error, object modification performed after the *recovery line* must be undone.

2.2.5 Type–specific locking

In the paper [Schwarz 84], Schwarz and Spector give a formalism for specifying the concurrency properties of shared abstract types and a locking technique that allows semantic information about shared abstract types to be used to allow higher degrees of concurrency. The higher degrees of concurrency are achieved by allowing the interleaving of operations from different atomic actions on an object. To ensure that the consistency of the object is maintained and that the atomic actions are serialisable, a special locking scheme called type–specific locking is required. A type–specific locking scheme will only allow the acquisition of a lock by an atomic action on an object if it does not conflict with the locks already held by other atomic actions on that object, such conflicts being detected by the use of a lock compatibility table. Examples of the use of type–specific locking schemes are given, for a directory and two types of queues.

For example, a directory object may allow two atomic actions simultaneously to modify entries, if each atomic action is modifying a different entry. Below is the lock

compatibility table for a directory with two entries (*a* and *b*), the directory having two operations *Modify* and *Inspect*.

		Held Lock			
		Modify(a)	Modify(b)	Inspect(a)	Inspect(b)
Required Lock	Modify(a)	n	y	n	y
	Modify(b)	y	n	y	n
	Inspect(a)	n	y	y	y
	Inspect(b)	y	n	y	y

2.2.6 Setwise serialisability and compound transactions

The work done by Sha on *setwise serialisability* and *compound transactions* (atomic actions) [Sha 85][Sha 88] deliver respectively: provably consistent, correct, modular and optimal scheduling rules, and a provably higher degree of concurrency by using application dependent modular concurrency control.

2.2.6.1 Setwise serialisability

The construction (and proof) of setwise serialisable scheduling rules depends on the concept of *atomic data sets*. An atomic data set is a collection of data objects which have associated with them consistency constraints that can be satisfied independently of other atomic data sets. This means that if a system state is partitioned into atomic data sets, and each transaction in that system is consistent and correct when executed alone in the system, then each transaction in the system will be also be executed consistently and correctly when it is executed setwise serialisably. In practise it could be difficult to identify atomic data sets.

2.2.6.2 Compound transactions

Compound transactions allow higher concurrency by utilising the semantic information of a given transaction. The semantic information is used to partition the compound transactions into a partially ordered set of *elementary transactions*. This partitioning being correctness preserving, and each atomic data set being independently consistent, allows scheduling rules which provide higher concurrency. But the resulting concurrency control is application dependent, and non modular, so if a compound transaction were added to the system, all compound transactions would require repartitioning into set of *elementary transactions*, in practise this could be unacceptable.

2.2.7 Exception trees

A paper by Campbell and Randell [Campbell 86] proposes a model of error recovery in asynchronous system, using the message and process model of atomic actions. The use of error recovery in a single process system is discussed first. Then the use of forward error recovery in asynchronous systems using atomic actions is explained. Their paper proposes that if an exception is raised by one of the processes participating in an atomic action then all the processes participating in that atomic action should take part in the error recovery. The way in which such an error recovery scheme can be used is illustrated by an example of a banking service. The problems involved with concurrently raised exceptions is solved by the use of an exception tree, which combines the different exceptions into a single exception. An exception tree has exception types at its nodes, with a special exception type called an universal exception at the root. The exception raised as a result of concurrent exceptions being raised is found at the root of the smallest sub–tree which contains all the

exceptions raised. An example of the exception tree for a twin–engined aircraft is given, showing the exceptions which result from different forms of engine failure. The problems of exceptions in nested atomic actions are then discussed.

2.3 Approaches not based on atomic actions

In this section another approach to constructing reliable distributed applications will be discussed. This is to provide an environment which enables easier construction of such applications.

2.3.1 ISIS

The aim of the team which implemented the ISIS system [Birman 87][Birman 88] was to provide an environment suitable for the construction of reliable distributed systems. To achieve this goal, the ISIS system provide tools which make use of the idea of *virtual synchrony*. Virtual synchrony provides an environment in which operations (potentially replicated) will appear to be done synchronously, even thought in many cases the operations will have actually be done asynchronously. Virtual synchrony is provided in ISIS by the use of *atomic multicasting* primitives. The multicast primitives are atomic in the sense that either all the intended recipients receive the multicasted information, or none do. The ISIS system provides three multicast primitives: ABCAST, CBCAST, and GBCAST.

The ABCAST (atomic broadcast) primitive is used in situations in which a number of processes are communicating with another set of processes providing a common shared resource. To maintain consistency of that resource, the communications must be acted upon in the same order by each of the processes. For example the operations of a replicated FIFO queue must be acted upon in the same order by each of the processes providing the replicated FIFO queue.

The CBCAST (causal broadcast) primitive is used in cases in which it is only necessary that communications from related processes are acted upon in the same order by overlapping destinations. The order in which communications are acted upon, should reflect the temporal relationship between the set of processes using the service. Therefore the CBCAST primitive offers a weaker ordering requirement than the ABCAST primitive, since only communications *potentially causally related* must be acted upon in order. Two multicast communication events are defined to be potentially causally related if information about the first event could have reached the point where the second event was begun before the second event was initiated. The CBCAST primitive therefore guarantees that communications that are potentially causally related are acted upon everywhere in the order in which the CBCASTs are invoked.

The GBCAST (group broadcast) primitive causes communications between process which are ordered with respect to all communications caused by both the ABCAST and CBCAST primitives (the order in which communications caused by GBCASTs are acted upon depends upon the type of the communication, for example, GBCAST communications indicating failure of a process are acted upon after all other communications from the failed process). The GBCAST primitive is used by the ISIS system to manage group addressing, and can be used by users for configuration management.

Some of the tools provided by the ISIS system which make use of virtual synchrony allow: process groups and group RPC, different methods of co–operation to execute replicated requests, use of replicated semaphores, and detection and reaction to failure.

The user of the ISIS system develops software by interconnecting non–distributed programs using the tools provided.

2.4 Summary

In this section, techniques which have been used or proposed for the construction of reliable applications were described. The properties of atomic actions (and nested atomic actions) were discused, as well as how atomic actions can be used to construct reliable applications.

Atomic actions provided an integrated mechanism which address the problems of inconsistency due to partial failure, and interference between concurrent parts of applications. Inconsistency due to partial failure is prevented by atomic actions due to their property of *failure atomicity,* which means that "an atomic action on completion either has performed the desired effects or has had no effect at all". Interference between concurrent atomic actions is prevented by atomic actions due to their property of *serialisability*, which means that "concurrent execution of atomic actions should produce the same effects as some serial order execution of the atomic actions". However, the price paid for these properties is that atomic actions can be highly restrictive on how applications can be constructed.

After describing atomic actions, methodologies which extend the model of atomic actions were described. The new structuring techniques of nested top–level atomic actions in Argus [Liskov 84] and split–transactions [Pu 88] were described. Two new models for the atomic action systems proposed by Shrivastava [Shrivastava 82] and Sha [Sha 85] [Sha 88] were described. Extensions to the conventional atomic actions model, which provided higher concurrency, such as type–specific locking [Schwarz 84], and early lock release [Net 85], were discused. The idea of using exception trees [Campbell 86] for controlling the concurrent rasing of exceptions, was described.

Finally, an approach not based on atomic actions was discused; this approach based on message ordering is used in the ISIS system [Birman 87] [Birman 88], which provided a set of broadcast primitives, and a tool kit which used them. The relationship between ISIS like systems and transaction systems is as yet not well understood and is a topic for research.

3 The Arjuna system

The *Arjuna system* [Shrivastava 88] provides tools which allow the construction of reliable distributed object oriented applications. The applications are made reliable by the use of replication and atomic actions. The rest of this section contains: a description of the present implemention of the Arjuna design, and an explanation of the tools provided by the Arjuna system for the construction of distributed applications using atomic actions.

3.1 Introduction to structure of the Arjuna system

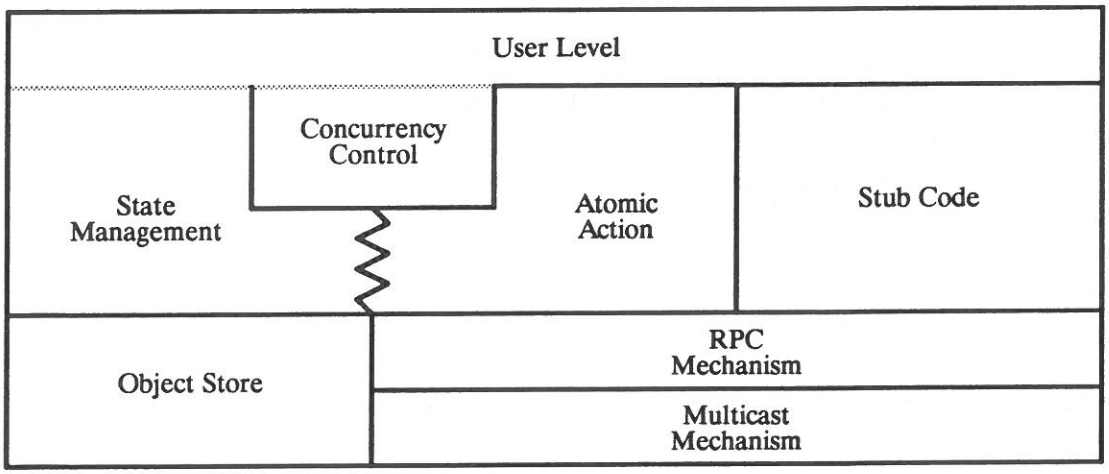
The Arjuna system assists in the construction of reliable distributed applications by providing tools and mechanisms which allow a user to build an application which is structured as atomic actions operating on persistent objects. The possibility that the objects are on different nodes is made invisible to the user application by the use of a *stub generator* tool. The stub generator generates code which enables the operations of an object to be invoked by a user application even if the object and application are on different nodes. The code provided by the stub generator performs operations on remote objects through a remote procedure call protocol. The remote procedure call protocol make use of a multicast protocol, so allowing the invocation of operations on *replicated* remote objects.

The Arjuna system is presently implemented using C++, and extensively uses the inheritance mechanism provided by C++. The present implementation is running on a number of UNIX workstations, connected by a local area network.

In keeping with the view that Arjuna is an object oriented system, the mechanisms needed to enable the construction of reliable distributed applications are presented to users in an object oriented manner. Both atomic actions and locks are objects, which can be manipulated using their operations like any other object. If the implementor of a class decides that the objects of that class need to be either persistent, recoverable, or have concurrent invocations of their operations controlled, then this can be simply achieved by the class inheriting the appropriate class provided by the Arjuna system, with very minor changes to the original class.

The Arjuna system contains many separate layers which are integrated to form the required functionality, to enable the uses of distributed atomic actions. These layers are illustrated in figure 3–1.

Figure 3–1



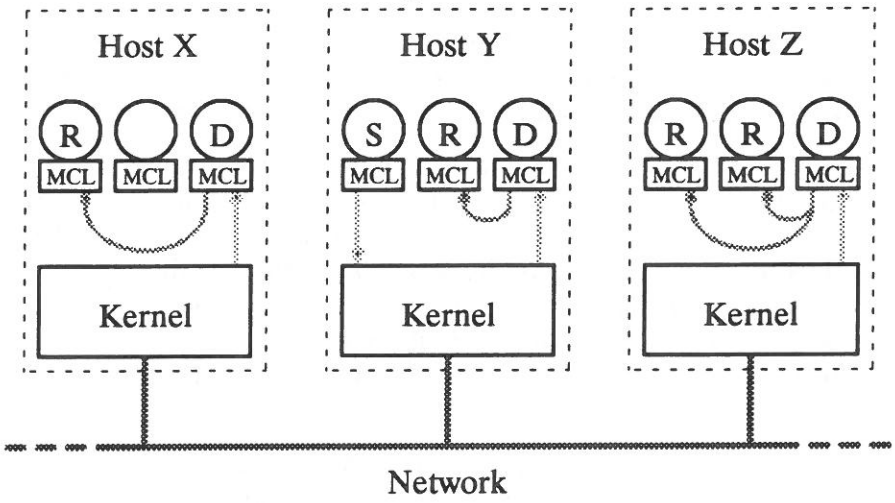
3.1.1 Multicast layer

The *Multicast layer* [Hughes 86] within the Arjuna system provides a mechanism by which a process may efficiently send a message to many receiving processes. The sending process will send the message to a *multicast group*. Processes are able to *join* and *leave* multicast groups using operations provided by the multicast layer. A message to be sent to a multicast group will be broadcast to all nodes on the network (including the sender's node). On each of the nodes the message will be received by a *multicast daemon*; this process will forward the message to all the local processes (if any) which are members of the multicast group to which the message was sent.

Use of this strategy means that only one message is sent over the network, regardless of the number of receiving processes. The use of the multicast daemon to forward the messages instead of broadcasting to all possible receiving processes means that there is no overhead caused by processes having to discard unwanted messages.

The diagram in figure 3–2 illustrates an example of the sending of a multicast message. The process marked with a *S* on host *Y* is the sending process, the processes marked with a *R* are receiving processes, the processes marked with a *D* are multicast daemons, and *MCL* is the multicast layer software.

Figure 3–2



3.1.2 RPC mechanism

The *RPC mechanism* provides a reliable remote procedure call (RPC) interface to the multicast layer (`multicall`), the mechanism used is based on the *Rajdoot* remote procedure call protocol [Panzieri 85][Panzieri 88]. The RPC mechanism also provides group management primitives to allow: the creation of a group of server processes (`initiate`), the joining of a server process to a group (`joingroup`), the leaving of a group by a server process (`leavegroup`), and the termination of a group of server processes (`terminate`).

The group management primitives also provide an orphan (server which no longer has a client) detection and killing mechanism. The orphan detection mechanism works by detecting the failure of nodes, and signalling any servers whose client was on the crashed node.

3.1.3 Stub generator

The *Stub generator* [Parrington 89] [Gibbons 87] is a tool which is used to allow certain classes of objects to be accessed remotely. That is to say that the operations of the objects can be invoked from nodes other than the node on which they are resident. This is

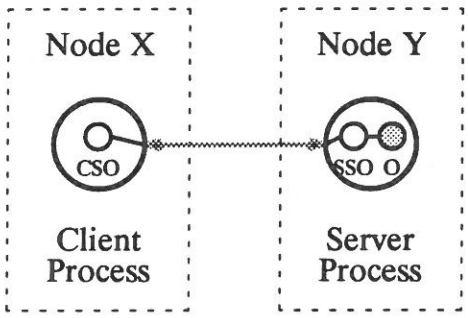
achieved by the stub generator producing code for two classes: a *client stub* class, and a *server stub* class. The stub code is generated from the class interface definition of the objects which are to be remotely accessed. The client stub object has exactly the same interface as the remote object. When a client stub object (i.e. instance) is created this causes the creation of a server stub object and remote object. Any operation invoked on the client stub object will cause the same operation to be performed on the corresponding remote object. This approach to making objects distributed is transparent to the user of the object.

The functions of a client stub object are to pack the parameters with which the operation is invoked into a form which can be transmitted between nodes (possibly between nodes with different ways of representing data), then to send the information to the appropriate server stub object, and to unpack any results that may be returned from the server stub object, and provide them to the invoker.

The operations of a server stub object will unpack the parameters sent by the client stub object and call the appropriate object operation with the parameters; any results produced are packed by the server stub object and sent to the client stub object.

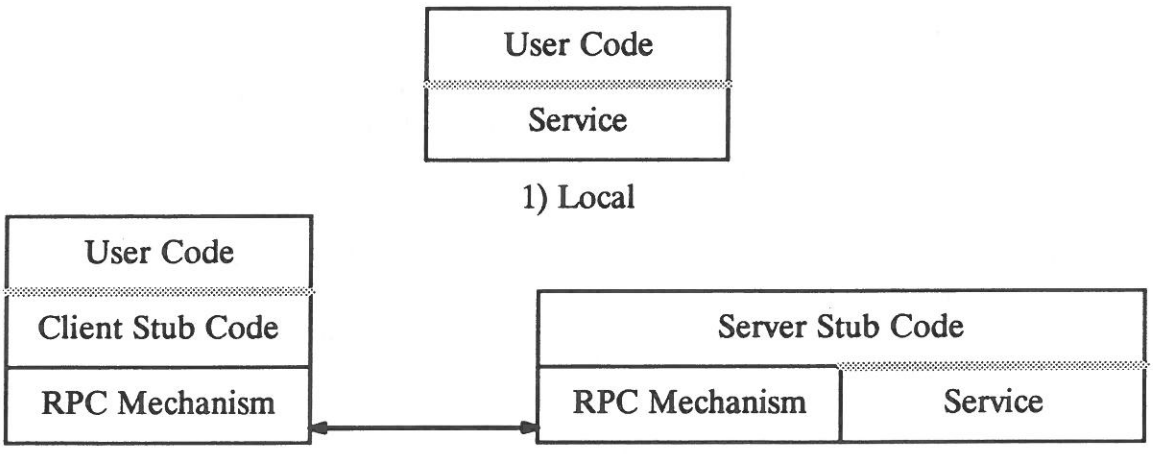
Figure 3–3 illustrates the client stub object (*CSO*), server stub object (*SSO*), and object (*O*, depicted as shaded).

Figure 3–3



The diagram in figure 3–4 shows the difference in software structure between user code accessing a local service and a user accessing a remote service. The service, in this context, is the set of operations provided by the object.

Figure 3–4

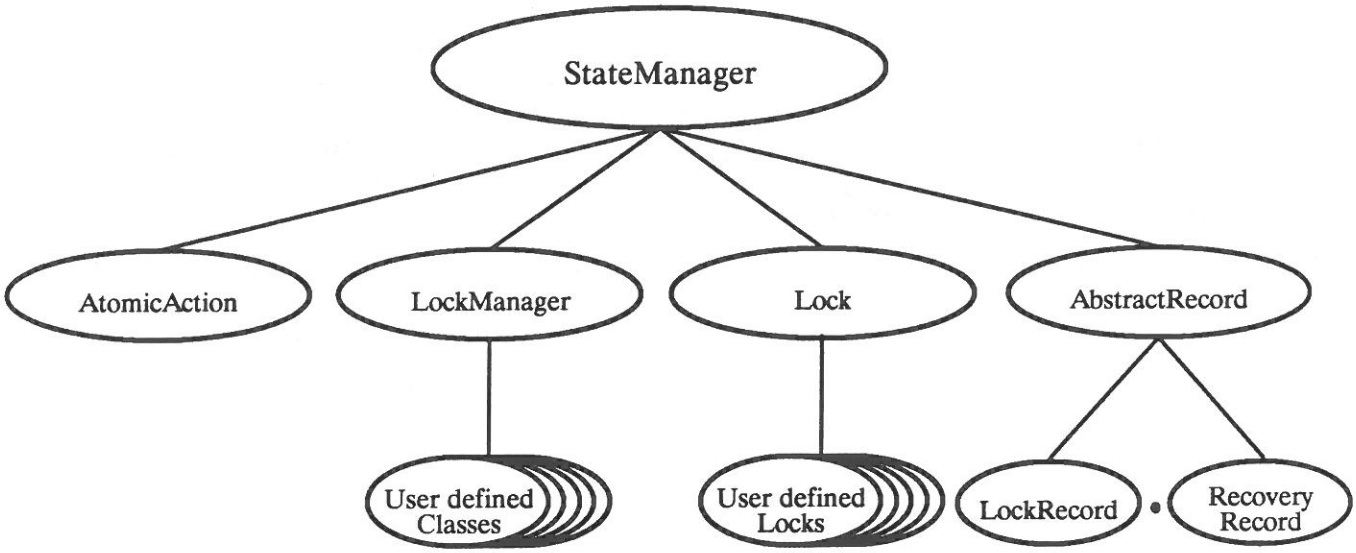


3.1.4 The Arjuna class hierarchy

The diagram in figure 3–5 shows the class hierarchy [Dixon 87a] for the Arjuna system. All the classes are derived from the class `StateManager`, which provides an interface to

the Arjuna object store. This means that instances of all the classes in the hierarchy can be stored in the object store. The user defined classes which are accessed using atomic actions, are derived from class `LockManager`. The class `lock` provides the standard two-phase locking facilities. If the user requires any special forms of locking, then this may be achieved by deriving classes from the class `Lock`.

Figure 3–5



3.1.5 Object Store

To manage the states of persistent objects stored at a node, the Arjuna system provides a class called *ObjectStore*, an instance of which can be used to manipulate the object store on that node. In the Arjuna system, each node in the network possesses its own object store. Because the object store must be able to survive the failures of the node on which it resides, the current implementation of the object store makes use of the UNIX file system, which is assumed to survive crashes (that is, file information not resident in the volatile store will not be affected by a crash). The state of each persistent object is stored in an individual file. These files are divided into directories depending on the class of the object. The directory name in which the state of a persistent object resides is determined by the name of the class and the names of the classes it inherits. For example, the objects of a user defined class `SpreadSheet` would be stored in the directory `StateManager/LockManager/SpreadSheet` (see figure 3–5) within the `ObjectStore` directory. The individual objects of the `SpreadSheet` class in the directory are named using unique identifiers, which are instances of the class `Uid`.

3.1.5.1 State management mechanism

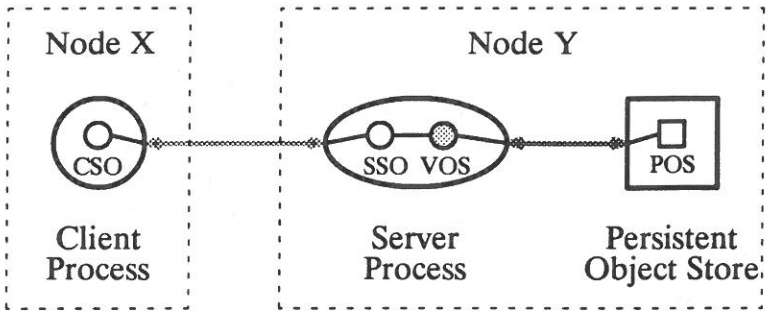
The *State management mechanism* [Dixon 87b] [Dixon 88] [Dixon 89] provides support for the use of both persistent and recoverable objects, so allowing a process to load the state of a persistent object into an instance of the object's class, and later, save the new state of the object back into the object store. The state management mechanism also enables the implemention of recoverable objects, allowing modifications to both persistent and non-persistent objects to be undone.

If the implementor of a class decides that the objects of that class need to be either persistent or recoverable, then this can be simply achieved by the class inheriting the class *StateManager*.

The state management mechanism has been integrated with the stub generator, so allowing remote objects to be persistent and recoverable, also allowing the user (or client) of a remote object to invoke operations of the `StateManager` object, from which the

remote object is derived. This is illustrated in figure 3–6, where the client process contains a client stub object (CSO), the server process contains both a server stub object (SSO) and a volatile object state (VOS (the remote object state)), and the persistent object store contains the persistent object state (POS).

Figure 3–6

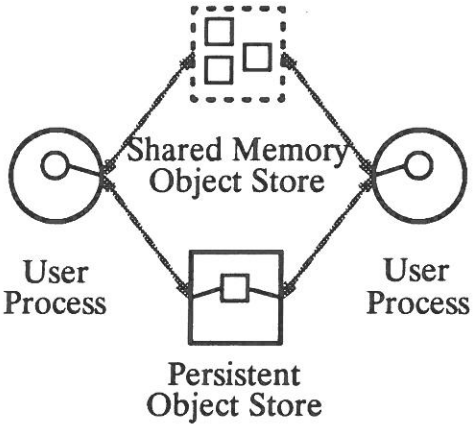


3.1.5.2 Concurrency control mechanism

The *Concurrency control mechanism* [Parrington 88a] [Parrington 88b] provides support for the placing of locks on instances of objects, so ensuring that only one process is modifying an object at any one time, and ensuring that an object is not being modified by one process while it is being inspected by another process. The present implementation of the concurrency control mechanism supports two–phase locking scheme, with (shared) read and (exclusive) write locks. If the implementor of a class decides that the member operations require concurrency control, then this can be simply achieved by the class inheriting the class *LockManager*.

In the Arjuna system, locks are objects; the *Lock* class being derived from *StateManager* means that lock objects can be saved in the object store. Because of the time overheads involved in using the persistent object store, the concurrency control mechanism uses a shared memory object store, for storing locking information. This is illustrated in figure 3–7; a persistent object is being accessed by two user processes, the state of the object is stored in the persistent object store, and the locking information is stored in a shared memory object store.

Figure 3–7



3.1.5.3 Atomic action mechanism

The atomic action mechanism co-ordinates the state management and concurrency control mechanisms to allow operations to: begin a new atomic action, end the current atomic action, or to abort the current atomic action. It has been implemented to allow nesting of atomic actions, and for nested atomic actions to be created on remote nodes, their effects to be undone even if their enclosing atomic action aborts.

In the Arjuna system, atomic actions are provided by instances of the class `AtomicAction`. The atomic actions are begun, committed and aborted, by calling operations on the objects.

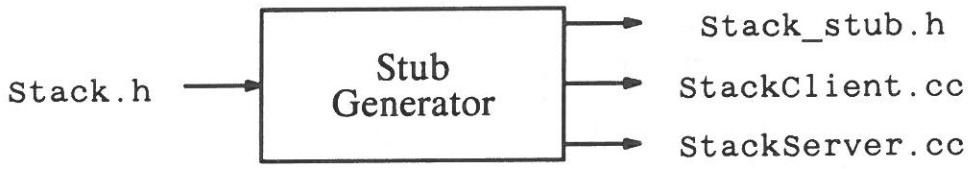
3.2 Introduction to the application level

In this section, details of how the Arjuna system can be used to construct reliable distributed applications will be presented. The main facilities provided by the Arjuna system, to aid the construction of reliable distributed applications are: a stub generator to allow remote invocation of operations, a state management mechanism, a concurrency control mechanism, and an atomic actions mechanism which co-ordinates the state management and concurrency control mechanisms, to allow the operations of objects to be performed as atomic actions. In the next section, the use of the Arjuna stub generator will be explained.

3.2.1 Stub generator

The stub generator provided as a part of the Arjuna system allows the operations of C++ objects to be invoked from nodes other than the one on which the object is located. The operations of the remote objects are invoked by a user application in an identical manner to that used if the objects were local to the application. This is achieved by the stub generator providing a replacement object for the user application, which is implemented so as to invoke the operations of the remote object; this replacement object will be referred to as a *client stub object*. The stub generator uses the information from a standard C++ header file to generate: a replacement stub header file, client stub class code, and server stub class code. This is illustrated in figure 3–8, with a header file for a class called `Stack`.

Figure 3–8



The C++ header file will contain the interface definition of the class whose operations will be invoked remotely. An example of such an interface definition, stored in a file called `Stack.h`, is given below.

```
#include "boolean.h"

class Stack
{
      int top;
      long values[100];
public:
      Stack();            // constructor
      ~Stack();           // destructor

      bool Push(long val);
      bool Pop(long *val);
};
```

Let the implementation of the class `Stack` be in the file called `Stack.cc`, and of the form:

```
#include "Stack.h"

Stack::Stack()
{
      . . .
```

```
}

Stack::~Stack()
{
        . . .
}

bool Stack::Pop(long val)
{
        . . .
}

bool Stack::Push(long *val)
{
        . . .
}
```

And, let the user application code be in a file called `UserAppl.cc`, and of the form:

```
#include "Stack.h"

int main(int argc, char *argv[])
{
        int v;
        bool err;

        Stack StackObj;

        err = StackObj.push(10);

        . . .

        err = StackObj.pop(v);
}
```

To compile the above simple application, the following command would be required:

```
CC -c UserAppl.cc                    — Compile user application
CC -c Stack.cc                       — Compile stack
CC -o UserAppl UserAppl.o Stack.o    — Link user application and stack to form an
                                     — executable program
```

To allow the above user application to create, and invoke operations on remote stack objects, the following must be done: The file `Stack.h` must be passed through the stub generator, this would produce the files: `Stack_stub.h`, `StackClient.cc` and `StackServer.cc`. The header file `Stack_stub.h` should be used instead of the `Stack.h`, by user application and the implementation of the stack object, because `Stack_stub.h` contains the class interfaces for server stub objects, the client stub objects, and the remote objects. To ensure that the correct class interface is used, the correct preprocessor variables must be set, before including `Stack_stub.h`.

For example, the implementor of user application above would replace:

```
#include "Stack.h"
```

with:

```
#define STACK_CLIENT
#include "Stack_stub.h"
```

And, the implementor of stack object would replace:

```
#include "Stack.h"
```

with:

```
#define STACK_SERVER
#include "Stack_stub.h"
```

If a stack server program is to be created, then the code in `StackServer.cc` should be compiled with the compiler option `-DSTACKSERVER_MAIN`.

To compile the above files, to form the user application and a stack server program, the following command would be required:

```
-- Run stub generator

tatsu Stack.h -s /usr/servers/stack
```

The `-s` option is used to indicate the path to the stack server program, which will be produced by the following commands. In the above example the location of the stack server program is intended to be `/usr/servers/stack`.

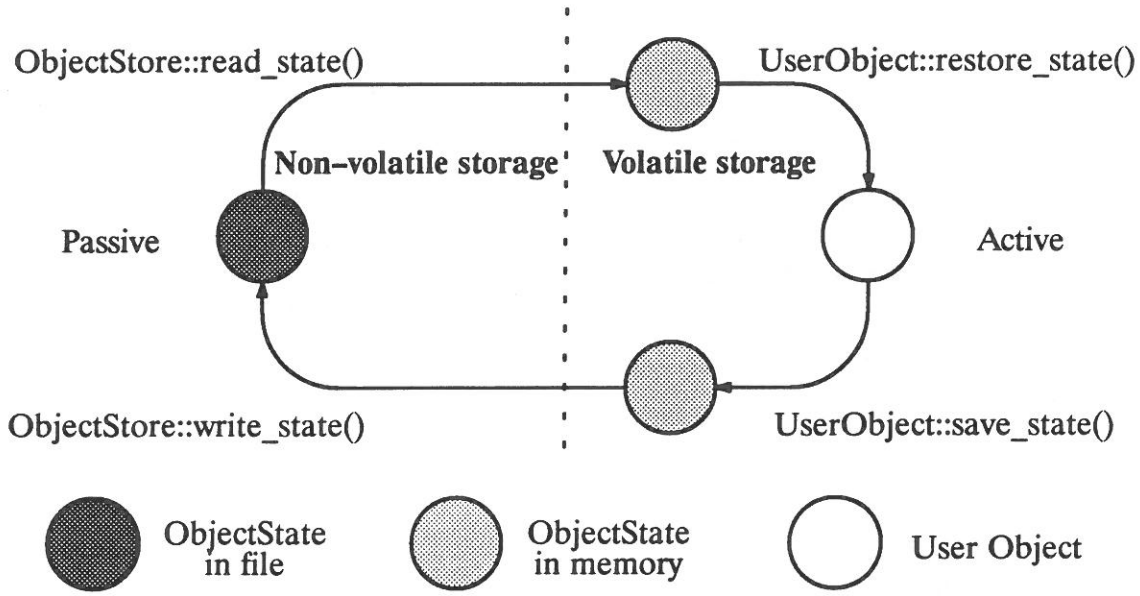
```
-- Server Program

CC -c Stack.cc                                  — Compile stack
CC -c -DSTACKSERVER_MAIN StackServer.cc         — Compile server stack stub code
CC -o /usr/servers/stack StackServer.o Stack.o  — Link stack and server stack stub
                                                — to form the server program

-- User application Program

CC -c UserAppl.cc                               — Compile user application
CC -c StackClient.cc                            — Compile client stack stub code
CC -o UserAppl UserAppl.o StackClient.o         — Link user application and client
                                                — stack stub to form user appliction
                                                — program
```

3.2.2 Persistent object life cycle

Before explaining how the Arjuna system provides state and lock management, along with their management to form atomic actions, the life cycle of persistent objects in Arjuna will be explained.

In the life time of a persistent object, it may be made active then passive many times, as illustrated by figure 3–9.

Figure 3–9



The times when an object is made active or passive, are managed by the state management mechanism.

The state of a user object, when it is to be made passive, must be first converted into an instance of the class `ObjectState`, (this conversion is performed by an operation

provided by the implementor of the user object), the instance of the class `ObjectState` is suitable for storing in the ObjectStore (or even transmission between nodes with different architectures).

When an object is made active, the instance of the class `ObjectState` is retrieved from the object store, and converted into an instance of the user object (this conversion is performed by an operation provided by the implementor of the user object). Operations may now be performed on the user object.

3.2.3 State management class

The state management mechanism of the Arjuna system enables objects to be persistent and recoverable. The Arjuna class which manages the states of object is `StateManager`, and provides operations which allow objects to be made active or passive. Below are the relevant parts of the `StateManager.h` header file:

```
enum object_type { RECOVERABLE, ANDPERSISTENT, NEITHER };

class StateManager
{
        . . .
protected:
        void modified();
        void terminate();
        void persist(Uid* =0);
public:
        StateManager();
        virtual ~StateManager();

        virtual bool save_state(ObjectState*, object_type);
        virtual bool restore_state(ObjectState*, object_type);

        bool activate(ObjectStore* =0);
        bool deactivate(ObjectStore* =0, WriteType =OVERWRITE);

        . . .

        Uid* get_Uid();

        . . .

        virtual TypeName type();
        virtual void destroy();
};
```

The naming of persistent object states by the state management mechanism is performed using unique identifiers. A unique identifier is an instance of the class `Uid`. Such objects are guaranteed to be unique, even if produced on different nodes.

In the following sections, the modifications required to a class whose objects are to be made persistent and recoverable, are explained, along with the operation provided to manage the state of the resulting object.

3.2.3.1 Modifications required to the class

If the implementor of a class wishes its objects to be persistent or recoverable, then this can be achieved by deriving the class from the Arjuna class `StateManager`. If the objects of a class also require concurrency control, then the class should be derived from `LockManager`, because `LockManager` is derived from `StateManager`, and so the operations of `StateManager` can also be accessed via `LockManager`.

Along with the deriving the class from `StateManager` or `LockManager`, the implementor of the class must also redefine the virtual operations `save_state()`,

restore_state() and type(), of StateManager. The save_state() and restore_state() are used to transform the object's state into/from an ObjectState object. The ObjectState object is of a form which is suitable for saving in (or retrieving from) the object store.

An example of the structure of a save_state() operations is given below. Note that it allows the implementor to specify different behaviour when saving the object's state into an ObjectState object, depending on whether the ObjectState object is to be recoverable (in this case the object_type parameter t will be RECOVERABLE), or recoverable and persistent (in this case the object_type parameter t will be ANDPERSISTENT). The implementor of a class can use this facility, for example, in the case of recovery to save the pointer to an object, but in the case of persistence to save the uid of the object. The structure of a restore_state() operation is similar to the structure of the save_state() operation.

```
bool RecObj::save_state(ObjectState *S, object_type t)
{
        bool res = FALSE;

        if (t == ANDPERSISTENT)
        {
                . . .
        }
        else if (t == RECOVERABLE)
        {
                . . .
        }

        return res;
}
```

The other operation which must be provided is the type operation, this operation tells the state management mechanism were the object should be saved in the object store. Below is an example of the structure of a type operation.

```
TypeName RecObj::type()
{
        return " . . . /RecObj";
}
```

The string which is returned will depend on the classes which the class RecObj is derived from. For example, the returned string would be "StateManager/RecObj", if RecObj was derived directly from StateManager, and would be "StateManager/LockManager/RecObj" if RecObj was derived directly from LockManager.

In this section, the modification required to a class to provide recovery and persistent were explained, in the next section the operations provided by the state management mechanism will be explained.

3.2.3.2 State management operations

The operations described in this section are the operations which are provided by the class StateManager, which are used for controlling the state management mechanism.

If an object is required to be persistent, the operation persist() must be called by that object, to inform the state management mechanism. If the object is to be used to manipulate an existing persistent object, then the Uid of the persistent object should be specified, by passing it as a parameter to the operation. Because this operation is protected, it can only be performed by the object itself.

The get_Uid() operation is used by the user application of an object to find the Uid of the persistent object to which it corresponds; this Uid can be regarded as the persistent object's "name", and can be used for accessing the object in the future.

If an object is modified by one of its operations, the `modified()` operation should be called to inform the state management mechanism, so that the new state of the object can be saved when the object becomes passive. The `modified()` operation being protected means that only the object itself can indicate that it has been modified.

The `destroy()` operation when performed on an object, will remove the corresponding persistent object state from the object store.

The `activate()` and `deactivate()` operations, are used to inform the state management mechanism that an object should be made *active* or *passive*, respectively.

If an object is making use of the concurrency control mechanism (described in the next section), many of the above operations will be called automatically, by the concurrency control mechanism. For example, when a lock is set on an object, the concurrency control mechanism will call `activate()` on the object, followed by `modified()` if the lock was a write lock.

The state management mechanism has been integrated into the atomic actions mechanism, registering, with the currently running atomic action, information about the states of objects before they are modified.

3.2.4 Lock management class

The concurrency control mechanism of the Arjuna system provides control over concurrent invocation of operations on an object. The concurrency control mechanism is based on the operations of an object, acquiring a lock (of mode read or write) on the object before accessing the state of the object.

The Arjuna class which manages the locks on an object is `LockManager`, and provides operations which allow locks to be set on objects. Below are the relevant parts of the `LockManager.h` header file:

```
class LockManager : public StateManager
{
        . . .
public:
        LockManager(bool oneserver = FALSE);
        LockManager(const Uid *, bool oneserver = FALSE);
        ~LockManager();

        status setlock (Lock *, int timeout = 100);

        . . .
};
```

If the implementor of a class wishes that its objects can be locked, then this can be achieved by deriving the class from the Arjuna class `LockManager`. Each operation of the object sets a lock on that object, by calling the `setlock` operation of the `LockManager` object.

The `setlock` operation can be parameterised with the type of lock required (Read / Write), and the number of attempts to acquire the lock before giving up (the default is 100 attempts, each attempt being separated by a 5 second delay.) If the lock requested is obtained, the `setlock` operation will return the value `GRANTED`, else `REFUSED`. Below are examples of the use of the `setlock` operation.

```
res = setlock(new Lock(WRITE), 10);

res = setlock(new Lock(READ), 0);

res = setlock(new Lock(WRITE));
```

The concurrency control mechanism has been integrated into the atomic actions mechanism, registering, with the currently running atomic action, what locks have been set on objects.

3.2.5 Atomic action class

The atomic action mechanism of the Arjuna system co-ordinates the state management and concurrency control mechanisms to allow the operations of objects to: begin a new atomic action, end the current atomic action, or abort the current atomic action.

The Arjuna system being object oriented, atomic actions are provided by instances of the class `AtomicAction`. The atomic actions are begun, committed and aborted, by calling operations of the objects. Below are the relevant parts the interface to the `AtomicAction` class:

```
class AtomicAction : public StateManager
{
      . . .
protected:
      PrepareOutcome Prepare();
      void Commit();
public:
      static AtomicAction *Current;

      AtomicAction ();
      AtomicAction (Uid*);
      virtual ~AtomicAction ();

      virtual bool save_state(ObjectState*, object_type);
      virtual bool restore_state(ObjectState*, object_type);
      . . .
      virtual Action_Status Begin();
      virtual Action_Status End();
      virtual Action_Status Abort();
      . . .
      bool add(AbstractRecord*);
      AtomicAction* Parent();
};
```

The operation which the `AtomicAction` class provides to begin a new atomic action is `Begin()`. The `AtomicAction` class also provides the `End()` operation to commit the current atomic action, and the `Abort()` operation to abort the current atomic action. Each of the operations returns a status, which can be used to ensure that the operation was successful. An example of the use of these operations is given below.

```
AtomicAction A;                  // Create an instances of atomic action

A.Begin();                       // Begin the atomic action

      . . .

if (OK)
      Result = B.End();          // End (Commit) the atomic action
else
      Result = B.Abort();        // Abort atomic action
```

The operations described above are all that are required to use atomic actions, but in addition, the `AtomicAction` class also provides a variable `Current`, which points to the current atomic action, along with an operation `Parent()` which returns a pointer to the atomic action's parent.

3.3 Summary

This section described, how the Arjuna system can be used to construct reliable distributed applications. The mechanisms provided by the Arjuna system to aid the

construction of reliable distributed user applications are: a stub generator, a state management mechanism, a concurrency control mechanism, and an atomic action mechanism.

In the next section, an example application will be developed using the Arjuna system; this application will be a spreadsheet. The spreadsheet will be first constructed as a persistent object, using the operations provided by the Arjuna system. The modifications required to allow the spreadsheet to be accessed using atomic actions will be described, then modifications required to form a remote spreadsheet, and finally, how the spreadsheet itself can be distributed.

4 Use of actions and objects

In the previous section, the basic concepts of the Arjuna system were described. In this section the use of these concepts will be illustrated in practice, by the designing and implementing a simple application. The purpose of this section is not only to illustrate how to use the Arjuna system, but also to show how atomic actions and objects can be used to provide the reliability required by an application. The example application chosen is that of a spreadsheet. The spreadsheet will be implemented as an object. For simplicity, the spreadsheet object consist of a two–dimensional array of elements, each element containing a value.

The operations provided by the spreadsheet class allow the value of an element of the spreadsheet to be either set or examined. Certain elements in the spreadsheet cannot have their values set, because their values are calculated from the values of the other elements in the spreadsheet. These formulas, for simplicity, are that the last entry of each row (and column) contains the sum of the preceding elements of the row (column). The interface definition of this simple spreadsheet class is given below:

```
class SpreadSheet
{
        int Elements[SprShSize][SprShSize];
public:
        SpreadSheet();
        ~SpreadSheet();

        int Set(int x, int y, int v);
        int Get(int x, int y, int* v);
};
```

The *Set* and *Get* operations change/return the value of the element x, y of the spreadsheet, respectively. (The *Set* operation cannot change the value of the last element in a row or column.)

In the following sections, the simple spreadsheet class described above will be modified to form:

i) A persistent spreadsheet class. Such a spreadsheet object should be able to survive the failure of the node on which it is located.

ii) A persistent spreadsheet class, whose operations are performed as atomic actions. Such a spreadsheet object should be able to survive the failure of the node on which it is located, and remain consistent despite concurrent operations being performed upon it.

iii) A persistent spreadsheet class, whose operations are performed as atomic actions, and are remotely invocable. Such a spreadsheet object should be able to survive the failure of both the node on which it is located and the node on which its user is located. The spreadsheet object should also remain consistent despite concurrent operations (potentially remote in origin) being performed upon it.

iv) A distributed spreadsheet class, where the elements of the spreadsheet are objects. Both the spreadsheet object and the element objects will be persistent objects, whose operations are provided as atomic actions, and are remotely invocable. Such a spreadsheet object should be able to survive the failure of the node on which it is located, and operations which do not involve elements which are on failed/inaccessible node should be successful. The spreadsheet object and element objects should also remain consistent despite concurrent operations being performed upon them.

4.1 Persistent spreadsheet class

In this sections, the implementation of a persistent spreadsheet class will be explained which uses the Arjuna system, and the reliability obtained from such objects described. Below is the interface definition of the persistent spreadsheet class:

```
class SpreadSheet : public StateManager
{
      int Elements[SprShSize][SprShSize];
public:
      SpreadSheet(Uid *UOld, int *res);
      SpreadSheet(int *res, Uid *UNew);
      ~SpreadSheet();

      int Set(int x, int y, int v);
      int Get(int x, int y, int* v);

      virtual TypeName type();
      virtual void save_state(ObjectState *, object_type);
      virtual void restore_state(ObjectState *, object_type);
};
```

The structure of the application is illustrated in figure 4–1, the process which uses the persistent spreadsheet object, obtaining the persistent object state from the object store.

Figure 4–1

Spreadsheet client — Spreadsheet Object State

The major differences between the interface for the persistent spreadsheet class and the interface for the simple spreadsheet class are: the class is derived from the `StateManager` class, the constructors are parameterised with the unique identifier (`Uid`) of the existing/created persistent object, and operations must be provided for the spreadsheet to be converted into/from an `ObjectState` object. These modifications, plus those required by the operations of the spreadsheet class will be described in the following sections.

4.1.1 Constructors

To access existing persistent objects, and to create new persistent objects, requires the spreadsheet class to have special constructors. If an existing persistent object is to be accessed, then the constructor is required to take the `Uid` of the persistent object, and pass it to the `StateManager` object, using the `persist()` operation. This informs the state management mechanism which object state to activate. The implementation of such a constructor is:

```
SpreadSheet::SpreadSheet(Uid *UOld, int *res)
{
      StateManager::persist(UOld);
      *res = OperDoneCorrectly;
}
```

If the constructor is to create a new persistent spreadsheet object, the implementation of such a constructor is:

```
SpreadSheet::SpreadSheet(int *res, Uid *UNew)
{
      StateManager::persist();              // Indicate new persistent object
```

```
        if (! StateManager::activate())       // Try to make object activate
              *res = CannotActivate;
        else
        {
              StateManager::modified();       // Indicate object is modified

              // Initialise spreadsheet elements

              for (x = 1; x <= SprShSize; x++)
                    for (y = 1; y <= SprShSize; y++)
                          Elements[x-1][y-1] = 0;

              if (! StateManager::deactivate())     // Try to deactivate object
                    *res = CannotDeactivate;
              else
              {
                    *UNew = *StateManager::get_Uid();   //Return Uid of object
                    *res = OperDoneCorrectly;
              }
        }
}
```

4.1.2 Operations

The operations of a persistent spreadsheet object are required to activate and deactivate the object, and also indicate if the object will be modified. Below is the structure of a set operation of a persistent spreadsheet object:

```
int SpreadSheet::Set(int x, int y, int v)
{
      int res = OperDoneCorrectly;

      if (! StateManager::activate())       // Try to make object activate
            res = CannotActivate;
      else
      {
            StateManager::modified();       // Indicate object is modified

            // Check parameters

            if ((x < 1) || (x => SprShSize) || (y < 1) || (y => SprShSize))
                  res = InvalidElement;
            else
            {
                  int OldValue = Elements[x-1][y-1];    // Save old value

                  Elements[x-1][y-1] = v;               // Change values
                  Elements[SprShSize-1][y-1] += v - OldValue;
                  Elements[x-1][SprShSize-1] += v - OldValue;
                  Elements[SprShSize-1][SprShSize-1] += v - OldValue;
            }

            if (! StateManager::deactivate())  // Try to deactivate object
                  res = CannotDeactivate;
      }

      return res;
}
```

4.1.3 Additional operations required

To allow the state of a spreadsheet to be saved/restored to/from the object store, the following operations must be provided: `Save_State`, `Restore_State` and `Type`.

An example of how the `Save_State`, `Restore_State` and `Type` operations could be implemented is given below. The result returned form `Save_State` and `Restore_State` indicates if the operations have been completed successfully

```
bool SpreadSheet::save_state(ObjectState *S, object_type t)
{
      int x, y;
      bool res = TRUE;

      // Pack each element into the ObjectState S

      for (x = 1; (x <= SprShSize) && res; x++)
            for (y = 1; (y <= SprShSize) && res; y++)
                  res = S->pack(Elements[x-1][y-1]);

      return res;
}

bool SpreadSheet::restore_state(ObjectState *S, object_type t)
{
      int x, y;
      bool res = TRUE;

      // Unack each element from the ObjectState S

      for (x = 1; (x <= SprShSize) && res; x++)
            for (y = 1; (y <= SprShSize) && res; y++)
                  res = S->unpack(&(Elements[x-1][y-1]));

      return res;
}

TypeName SpreadSheet::type()
{
      return "/StateManager/SpreadSheet";
}
```

Another approach to providing the `Save_State` and `Restore_State` operations would be to only save the elements of the spreadsheet which cannot be recalculated, and when the spreadsheet object is restored to recalculate the values of the remaining elements. This could reduce the size of the persistent object state when stored.

4.1.4 Reliability of the persistent spreadsheet

The spreadsheet object being persistent, means that the state of the object will survive the failure of the node on which it is located. The state of the object which will survive is the state in the object store, this being the state after the last operation which was performed on the object. But this could be inconsistent with the applications intentions, if for example, the application wished to perform two set operations, and either both or neither, should have appeared to have occurred. In addition concurrent operation on such a persistent object could cause inconsistencies in the object state. To overcome these problems with the reliability of spreadsheet class, it could be implemented using atomic actions. The implementation of such a spreadsheet will be described in the next section.

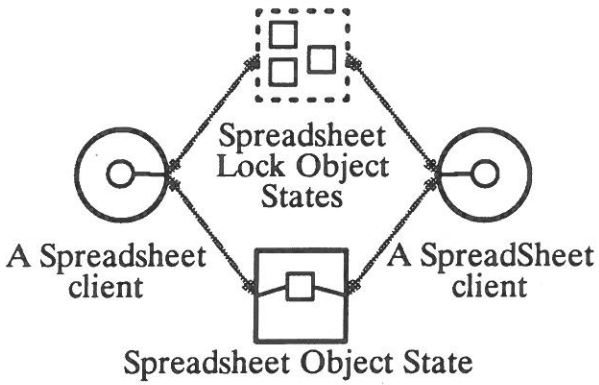
4.2 Spreadsheet using atomic actions

In this sections, a persistent spreadsheet class will be implemented, whose operations will be performed as atomic actions, and the reliability of such objects described.

To allow the spreadsheet class to make use of both the state management mechanism and the concurrency control mechanism provided by the Arjuna system, the spreadsheet class must be derived from a class called `LockManager`.

The way in which application processes access a spreadsheet object is illustrated in figure 4–2, the application processes requiring both access to the object state in the object store, and access to the locks which are held on the object.

Figure 4–2



In the following section, the modifications to the persistent spreadsheet class to allow the use of atomic actions will be described.

4.2.1 Constructors

To use the existing persistent objects requires the use of a special constructor; this constructor is required to take the `Uid` of the persistent object, and pass it to the constructor of the `LockManager` object it is derived from. The implementation of such a constructor is:

```
SpreadSheet::SpreadSheet(Uid *UOld, int *res) : (UOld)
{
      *res = OperDoneCorrectly;
}
```

The constructor which creates the new persistent object of the spreadsheet class will return the `Uid` of the persistent object created. The implementation of such a constructor is:

```
SPreadSheet::SpreadSheet(int *res, Uid *UNew)
{
      int res = UnknownError;

      AtomicAction A;

      StateManager::persist();        // Indicate new object is persistent
      if (A.Begin() == RUNNING)       // Try to start atomic action
      {
            if (setlock(new Lock(Write), 0) == GRANTED) // Try to set lock
            {
                  res = OperDoneCorrectly;

                  // Initialise spreadsheet elements

                  for (x = 1; x <= SprShSize; x++)
                        for (y = 1; y <= SprShSize; y++)
                              Elements[x-1][y-1] = 0;

                  // Get the persistent object's Uid

                  *UNew = *StateManager::get_Uid();

                  if (A.End() != COMMITTED)       // Commit
                        res = UnableToCommitAction;
            }
```

```
            else  // Lock refused so abort the atomic action
            {
                  res = UnableToSetLockOnSpSh;
                  if (A.Abort() != ABORTED)       // Abort
                        res = UnableToAbortAction;
            }
      }
      else          // Unable to start atomic action
            res = UnableToStartAction;

      return res;
}
```

4.2.2 Operations

If the operations of the spreadsheet class are to be performed is atomic actions, then the *Set* operation would be modified to have the structure given below:

```
int Set(int x, int y, int v)
{
      int res = UnknownError;

      AtomicAction A;

      if (A.Begin() == RUNNING)              // Try to start atomic action
      {
            if (setlock(new Lock(Write), 0) == GRANTED) // Try to set lock
            {
                  // Check parameter

                  if ((x<1) || (x=>SprShSize) || (y<1) || (y=>SprShSize))
                        res = InvalidElement;
                  else
                  {
                        int OldValue = Elements[x-1][y-1];  // Save old value

                        Elements[x-1][y-1] = v;              // Change values
                        Elements[SprShSize-1][y-1] += v - OldValue;
                        Elements[x-1][SprShSize-1] += v - OldValue;
                        Elements[SprShSize-1][SprShSize-1] += v - OldValue;

                        res = OperDoneCorrectly;
                  }

                  // If operations performed successfully commit the
                  // atomic action, else abort.

                  if (res == OperDoneCorrectly)
                        if (A.End() != COMMITTED)  // Commit
                              res = UnableToCommitAction;
                  }
                  else
                  {
                        if (A.Abort() != ABORTED)  // Abort
                              res = UnableToAbortAction;
                  }
            }
            else  // Lock refused so abort the atomic action
            {
                  res = UnableToSetLockOnSpSh;
                  if (A.Abort() != ABORTED)        // Abort
                        res = UnableToAbortAction;
            }
      }
      else          // Unable to start atomic action
            res = UnableToStartAction;
```

```
        return res;
}
```

The *Set* operation above must first create and begin an atomic action, the operation must then set a lock on the object (in the case of *Set* a write lock must be acquired), then the main body of the *Set* operation can be performed, if this is successful the atomic action can be committed, otherwise aborted.

For both the constructors and operations of the spreadsheet class, the activation and deactivation of the spreadsheet object is performed invisibly by the atomic action and concurrency control mechanisms.

4.2.3 Other operations

Because the spreadsheet class is derived from the `LockManager` class, the operation `type()` should be modified to be:

```
TypeName SpreadSheet::type()
{
        return "/StateManager/LockManager/SpreadSheet";
}
```

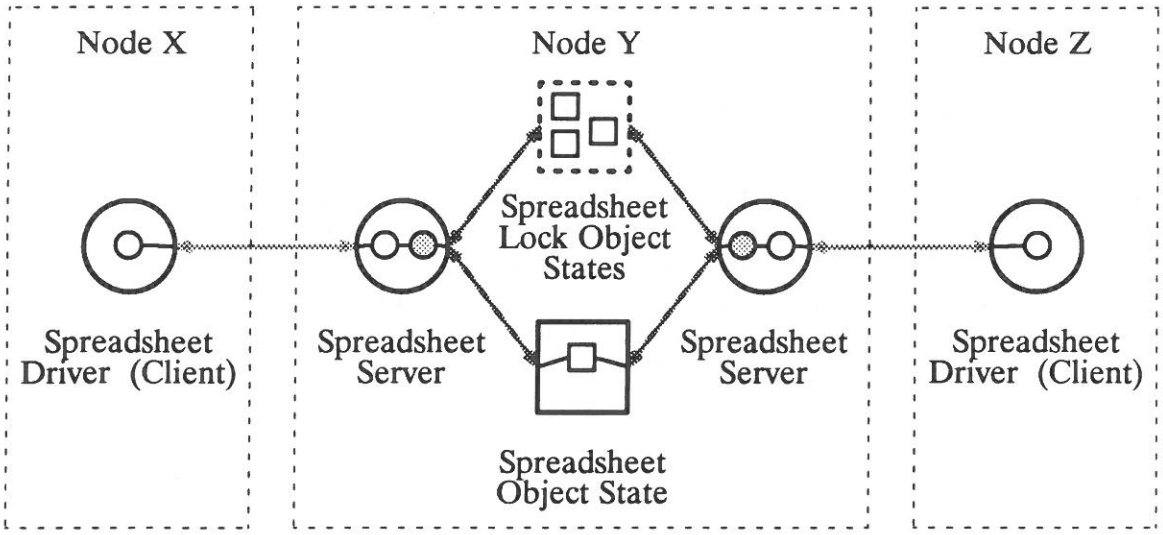
4.2.4 Reliability of the spreadsheet object using atomic actions

The spreadsheet object being persistent, means that the state of the object will survive the failures of the node on which it is located. The state of the object which will survive is the state produced by the last top–level atomic action performed on the object, which committed. So, if it was the intention of an application, to perform two set operations atomically, this can be done be nesting the set operations in an enclosing atomic action. In addition, concurrent operations on such a persistent object will be serialised, thereby preventing inconsistencies in the state of the object. But because the elements of the spreadsheet objects are not individually lockable, certain combinations of concurrent operation invocations will be executed serially, where they could be executed concurrently: for example, the setting of the value of two different elements in the spreadsheet.

4.3 Remotely invocable spreadsheet

In this section, the implementation of a spreadsheet class whose operations can be invoked remotely will be described; this class will be implemented with the help of the stub generator provided by the Arjuna system. The structure of the resulting system is illustrated in figure 4–3, the persistent object state being accessed by server processes created by the application (Client), the server process performing operations on the persistent object state using an atomic action, whose access is controlled using locks.

Figure 4–3



If the operations of the spreadsheet objects are to be invoked remotely, the class interface definition will have to be used by the stub generator to produce the appropriate stub code.

From the class definition in the file `SpreadSheet.h` the stub generator will produce three files `SpreadSheet_stub.h`, `SpreadSheetClient.cc` and `SpreadSheetServer.cc`. The file `SpreadSheet_stub.h` contains the class definitions of the client stub objects, server stub objects, and remote objects. This header file should be included by both by the users of the remote objects, and the remote objects implementation. The following must be placed in the spreadsheet class implementation before the `SpreadSheet_stub.h` is included, to ensure that the remote object definitions are used.

```
#define STATEMANAGER_SERVER
#define LOCKMANAGER_SERVER
#define SPREADSHEET_SERVER
```

And the following must be placed in the program which uses spreadsheet objects before `SpreadSheet_stub.h` is included, to ensure that the client stub object definitions are used.

```
#define STATEMANAGER_CLIENT
#define LOCKMANAGER_CLIENT
#define SPREADSHEET_CLIENT
```

The `SpreadSheetClient.cc` and `SpreadSheetServer.cc` code should be compiled, and the resulting object code being linked in to the remote users of spreadsheet objects, and the spreadsheet implementations, respectively. It should be noted that no changes to the class are required (since the stub generator automatically generates the code for client and server processes).

4.3.1 Reliability of remotely invocable spreadsheet object

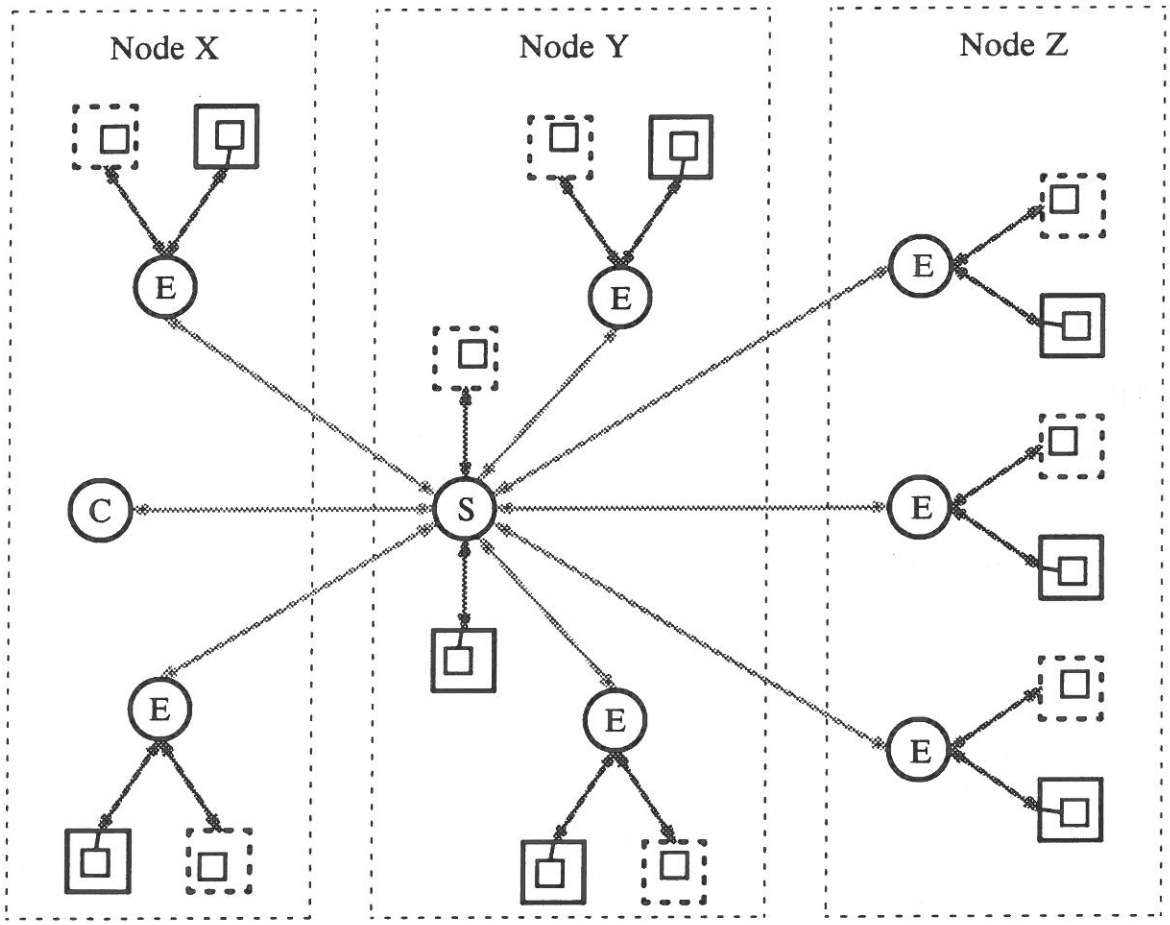
The introduction of distribution into an application in this way does not increase the reliability of the application. It does however increase the availability of the the spreadsheet object beyond the node on which it is located. Distribution of the spreadsheet does present additional reliability problems in that: the node on which the spreadsheet object (server) is located and the node on which the user (client) is located may fail independently, or a failure in the network could cause the failure of an invocation. The use of atomic actions to perform operations on the spreadsheet will however ensure the consistency of the spreadsheet object. But if the node on which the user (client) is located

fails, the spreadsheet object will still retain locks for the user. This could mean that the spreadsheet object becomes inaccessible, until the failure of the node is detected, and the atomic action is aborted.

4.4 Internally distributed spreadsheet

The spreadsheet classes outlined in the previous sections have their entire state in one place. In this section a spreadsheet whose elements are distributed (on many nodes) will be examined. The resulting structure of such a spreadsheet is illustrated in figure 4–4, where an application process, *C*, is using the spreadsheet object managed by the server process, *S*. The elements of the spreadsheet are distributed amongst the nodes *X*, *Y* and *Z*. The processes which contain elements are labelled *E* (as before, boxes drawn by dotted lines represent the locks' object store, and the boxes drawn by solid lines represent the persistent object store).

Figure 4–4



To allow the spreadsheet objects to contain elements which are distributed on other nodes, and which are individually persistent, the elements of the spreadsheet must be implemented as objects. Below is the class interface definition for such a class. Note that this class is derived from `LockManager`.

```
class Element : public LockManager
{
        int Value;
public:
        Element(Uid *UOld, int *res);
        Element(int *res, Uid *UNew);
        ~Element();

        int Set_Value(int v);
```

```
        int Get_Value(int* v);

        virtual TypeName type();
        virtual bool save_state(ObjectState *S, object_type t);
        virtual bool restore_state(ObjectState *S, object_type t);
};
```

Below is the modified form of the class interface definition for the element oriented spreadsheet.

```
class ElemSpreadSheet : public LockManager
{
        Element      *Elements[SprShSize][SprShSize];
public:
        ElemSpreadSheet(Uid *UOld, int *res);
        ElemSpreadSheet(int *res, Uid *UNew);
        ~ElemSpreadSheet();

        int Set_ElemValue(int x, int y, int v);
        int Get_ElemValue(int x, int y, int* v);
        int Set_ElemUid(int x, int y, Uid U);
        int Get_ElemUid(int x, int y, Uid* U);

        virtual TypeName type();
        virtual bool save_state(ObjectState *S, object_type t);
        virtual bool restore_state(ObjectState *S, object_type t);
};
```

The following must be placed in the `ElemSpreadSheet` class implementation before the `ElemSpreadSheet_stub.h` is included, to ensure that the remote object definition for the `Element` is used.

```
#define STATEMANAGER_SERVER
#define STATEMANAGER_CLIENT
#define LOCKMANAGER_SERVER
#define LOCKMANAGER_CLIENT
#define ELEMSPREADSHEET_SERVER
#define ELEMENT_CLIENT
```

Because the states of the element objects are now remote from the spreadsheet object, the `save_state` and `restore_state` operations must be modified. Below is the outline for the structure of such a `save_state` operation (`restore_state` would have a similar structure).

```
bool ElemSpreadSheet::save_state(ObjectState *S, object_type t)
{
        bool res = TRUE;

        if (t == ANDPERSISTENT)
        {
                // Persistence
                // For each element save its uid

                for (x = 1; (x <= SprShSize) && res; x++)
                        for (y = 1; (y <= SprShSize) && res; y++)
                                res = (Elements[x-1][y-1]->get_Uid())->pack(S);
        }
        else if (t == RECOVERABLE)
        {
                // Recovery
                // For each element save the pointer to the element

                for (x = 1; (x <= SprShSize) && res; x++)
                        for (y = 1; (y <= SprShSize) && res; y++)
                                res = S->pack((long) Elements[x-1][y-1]);
        }
```

```
        return res;
}
```

When saving the state of the spreadsheet to allow recovery is it is sufficient to save the local state of the spreadsheet object. But when saving the state of the spreadsheet to make it persistent what is required is the saving of the unique identifiers (`Uid`) of the remote persistent objects, because these unique identifiers all that is be required to recreate the remote objects.

4.4.1 Reliability of an internally distributed spreadsheet

Changing the elements of the spreadsheet object to individually lockable objects, permits operations which could not have been performed concurrently in the previous implementations of the spreadsheet, to be performed concurrently.

Introduction of internal distribution into the spreadsheet object means that operations which require elements on a node which are inaccessible will not be performed successfully, although other operations will be successful.

Another problem which could arise from this implementation is that element objects could be accessed directly by a user application. This would not cause the element object's state to become inconsistent, because its operations are performed by atomic actions. But it could cause the state of the spreadsheet object to become inconsistent.

4.5 Summary

In this section, a series of applications were described and implemented using the Arjuna system. The fault–tolerance capabilities of these implementations were also described.

As shown by the difference between the persistent spreadsheet object, and the persistent spreadsheet object whose operations were performed by atomic actions, the ability of the user applications to perform "consistent transformations" on an object is greatly increased by use of atomic actions. The use of of atomic actions also ensures that concurrent operations performed on objects will be free from interference.

The incorporation of distribution into an application does not necessarily increase the reliability of the application, because it may make the application susceptible to new kinds of failure previously absent. If the operations of a remote object are performed as atomic actions, then this will ensure the consistency of the remote object state. It is however possible that failure could occur after a successful invocation; this problem can be overcome by the client performing the remote invocations from within an atomic action.

The decomposition of an object into lockable sub–objects, is an important part of the design of an application, because it offers the possibility of increasing the concurrency in the application. This section has shown the object oriented approach provides a convenient means of structuring distributed applications.

5 Actions

In the following sections, new structuring techniques for actions will be introduced; these structuring techniques are designed to be used in conjunction with atomic actions, and their purpose is to provide mechanisms by which a system of long running actions can be made more efficient and easier to construct.

The term *action* will be used to denote a unit of work. This unit of work will have a time of starting and completion. An action may be constructed from a system of other smaller actions. In the following sections it will be assumed that the smallest unit of work will be an atomic action, and that locks will be used to provide concurrency control. Locks are not essential as the form of concurrency control for the ideas discussed in the following sections, but are used to make the explanations more specific.

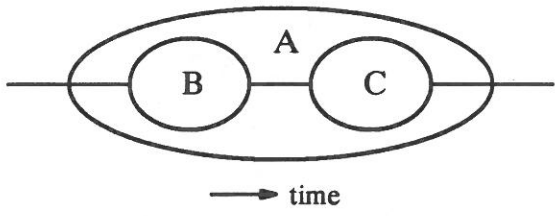
The structuring techniques for actions which will be introduced are: serialising actions, top–level independent actions, glued actions and common actions. But first, the problems which arise when using nested atomic actions will be considered.

5.1 Problems with nested atomic actions

The reason why new structuring techniques in addition to nested atomic actions are needed is that when constructing systems from only nested atomic actions, certain problems are encountered. These problems stem from the properties of failure atomicity and serialisability which atomic actions possess.

Failure atomicity means that all the effects of an atomic action which fails are undone. This can present problems when constructing long running atomic actions, which may exist for days or weeks, because it is undesirable to waste a possibly large amount of work which a long running atomic action could have performed if a failure occurs. An example of this problem is shown in the figure 5–1. This figure shows three atomic actions *A*, *B* and *C*. *B* is ordered to occur before *C*, and both *B* and *C* are nested within *A*. *B* performs some long and complicated operations on a set of objects *b*, and *C* performs an operation on a set of objects *c*, this operation depending on the new values of the objects in *b*. The set of objects in *b* must not change between the end of *B* and the start of *C*; this is why atomic action *A* is required. Suppose *C* aborts due to some hardware failure. It is an undesirable property of nested atomic actions that the failure of *C* causes the entire atomic action, *A*, to fail. All the operations performed by *B* must be undone to preserve the failure atomicity of *A*. In this case the problem arises because the only way in which objects could be prevented from being changed between the end of *B* and the start of *C* is by using an enclosing atomic action.

Figure 5–1



Failure atomicity also requires that all objects be made recoverable. This may not always be possible; much of the I/O done by a system with the outside world will not be recoverable, for example: the marks made by a plotter on a piece of paper, a servo opening a valve or the request for information from the user. So systems which contain irrecoverable objects are not suitable for construction from atomic actions.

Serialisability means that concurrent atomic actions cannot affect the progress of each other. This implies that no two way communication can be allowed between atomic

actions. It is therefore necessary to give a great deal of thought to the arrangement and order of the atomic actions, so as to maximise the concurrency, and minimise the possibility of deadlocks. In systems which contain long running atomic actions, the necessity for preserving serialisability can mean that objects which are being accessed by such long running atomic actions will be unavailable to other atomic actions for long periods of time.

Consider the example where if a long running atomic action requires access to an object which provides the interface to a hard disc, and the hard disc is required to have at least a certain number of free blocks. The long running atomic action must initially search for a suitable hard disc object which has the required amount of free blocks (this could not be performed by an atomic action before the long running atomic action, because there would be no guarantee the hard disc would still have the required number of free blocks when the long running atomic action started). The searching will require the inspecting of parts of the state of candidate objects. If serialisability is to be maintained, the parts of all the candidate objects which were inspected may not be changed by other atomic actions until the long running atomic action completes. This could cause large delays to other atomic actions requiring access to the object which provide the interface to hard discs.

The above discussion suggests that to enable easier construction of reliable distributed applications, the new structuring mechanisms and techniques are needed.

5.2 Serialising actions

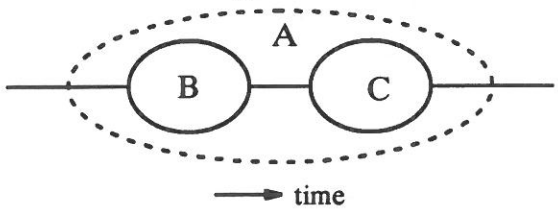
It is often desirable for a system of actions to be able to share objects, whilst denying access to other actions, and it is not always desirable for such a system to have the property of failure atomicity. If such a system is required then it can be constructed using *serialising actions*.

The use of serialising actions provides a means by which locks on objects can be acquired by a system of actions and released in such a way as to prevent actions not in the system from acquiring them. This is achieved by nesting the system of actions within a serialising action. The locks which are released by the system of actions will be retained by the serialising action until the system of actions has completed or has failed. If the system of actions fails, the serialising action will perform no additional recovery to that which was performed by the system of actions.

To ensure that the effects produced by actions at the top–level of the system are not lost in the event of a failure, it is necessary to make the effects permanent. This is possible because the enclosing serialising actions will not attempt to recover these effects.

Figure 5–2 shows such a system of actions nested within a serialising action *A*. In this case the system comprises two top–level atomic actions *B* and *C*. The atomic action *B* must complete before the atomic action *C* can start.

Figure 5–2



In this case the invocation of such a system should appear to have one of three results visible to the actions outside action *A*:

(i) The effects of neither *B* or *C* are never seen.

(ii) The effects of *B* are seen.

(iii) The effects of *B* and *C* are seen and the effects appear simultaneously.

One possible method for introducing such a construct into a system such as Arjuna would be to provide a class *SerialisingAction*, which provides an interface similar to the class *AtomicAction*:

```
SerialisingAction A;

A.Begin();
    . . .
A.End();
```

If such a class were available, the system of figure 5–2 would be programmed as follows:

```
SerialisingAction A;

AtomicAction B, C;

A.Begin                                 // Serialising action A start
      B.Begin();                        // Atomic action B start
          . . .
      B.End();                          // Atomic action B end
      C.Begin();                        // Atomic action C start
          . . .
      C.End();                          // Atomic action C end
A.End()                                 // Serialising action A end;
```

5.3 Top–level independent actions

To allow the construction of systems which contain actions (especially long running actions), it is desirable to provide a mechanism which allows actions to be invoked from within other actions, and for the effects of these invoked actions not to be recovered if failure occurs in the invoking action.

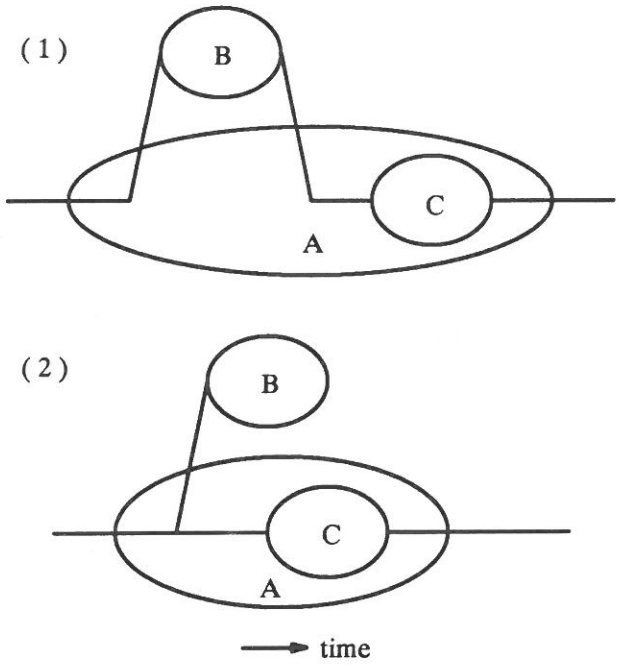
One method of providing such a mechanism is by the use of special actions called *top–level independent actions*. Top–level independent actions are not like normal nested actions in that their effect cannot be recovered if the action within which they are invoked fails. Top–level independent actions are based on the idea of *nested top–level transaction* which are provided as a part of Argus system [Liskov 84].

Another method which could be used to provide irrecoverable effects is the the use of irrecoverable objects. The advantage that top–level independent actions give over irrecoverable objects is that a top–level independent action if aborted will recover its effects so maintaining the consistency of the system.

Top–level independent actions could be used in either a synchronised or unsynchronised manner. The following sections will explain these two different methods, and give examples of where each is appropriate. It will also be shown that top–level independent actions are useful in systems which are not long running.

Figure 5–3 shows two action systems. In both systems, action *A* invokes two actions *B* and *C*, *B* being a top–level independent action. In (1), *B* is synchronised and in (2), *B* is unsynchronised. The invocation of B in both systems is followed by the invocation of *C*. In (1), this occurs after *B* has finished. In (2), it may occur before the completion of *B*, so *B* and *C* may run concurrently. In such systems, even if *C* fails causing *A* in turn to fail, the effects of B are not undone.

Figure 5-3



5.3.1 Synchronised top-level independent actions

When a top-level independent action is used in a synchronised manner, the invoking action will continue only when the invoked top-level independent action has completed. The invoking action could detect if the top-level independent action has failed, then decide if the failure should cause it to abort.

5.3.1.1 Synchronised top-level independent actions using locks

The invoking action and the top-level independent action it invokes are serialised, so synchronised top-level independent actions do not receive any privileges over other actions when attempting to acquire locks which are held by their invoking action. Synchronised top-level independent actions should not attempt to acquire locks on objects which are held by the invoking action, or by any action within which the invoker is nested. An attempt to acquire such a lock would cause a deadlock to occur.

In some cases it may be desirable that some of the locks acquired by a top-level independent action could be inherited by the invoking action when the top-level independent action commits. This would prevent other actions from changing the objects used by the top-level independent action before the invoking action has a chance to lock them.

5.3.1.2 Examples of synchronised top-level independent actions

1) A *bulletin board* could be accessed using top-level independent actions for the communication of information between systems of actions. The top-level independent actions would be used for the posting and retrieving of information on the bulletin board. Nested atomic actions are unsuitable for such a purpose because their effects are only made available to other actions when the top-level atomic action they are nested within is committed. Synchronised top-level independent actions are appropriate for this purpose because the invoking action will need to know when the top-level independent action is completed and if it was successful.

2) *Caching* information from a remote node onto the local node could be performed by a synchronised top-level independent action. Moving frequently accessed information closer to its accessor allows increased performance. The use of a top-level independent

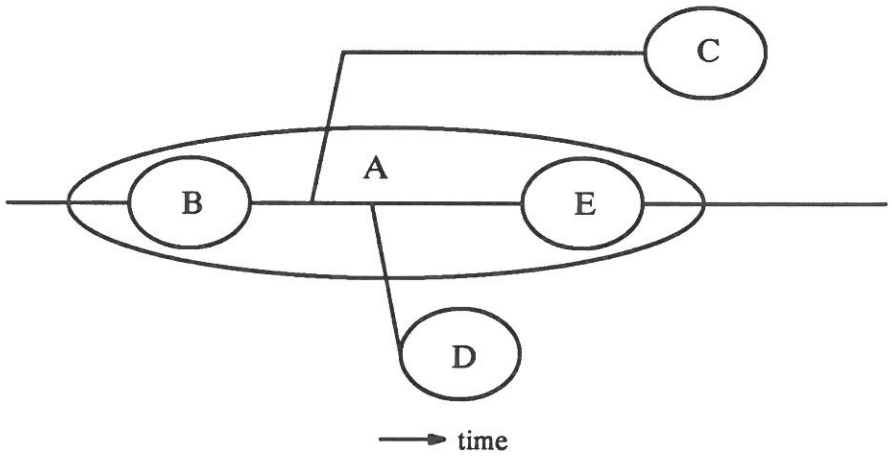
action to cache the information means the information will remain cached locally even if the action which initiated caching fails. Synchronised top–level independent actions are appropriate for this purpose because the action which invoked the caching must not continue until the caching has completed.

3) The use of lock inheritance from a synchronised top–level independent action as suggested earlier means that a top–level action can be used to acquire locks for the invoking action. The top–level independent action may be invoked to inspect an object, and if the object is suitable, to return a lock on the object to the invoking action; this would mean that the object could not be changed between the inspection and the use by the invoking action. The other advantage of the method would be that objects which were unsuitable would not remain locked, so being available to other actions.

5.3.2 Unsynchronised top–level independent actions

When top–level independent actions are used in an unsynchronised manner, the invoking action will continue immediately after the invocation and will not be informed of the completion of the invoked top–level independent actions, so will not know if it completed successfully or failed. The top–level independent actions may be executed after the invoking action has completed or be executed concurrently with it. This is illustrated in figure 5–4.

Figure 5–4



5.3.2.1 Unsynchronised top–level independent actions using Locks

Unsynchronised top–level independent actions, unlike synchronised top–level independent actions, may attempt to acquire a lock on an object which is held by the invoking action (or any action within which the invoker is nested). A lock will be released when the top–level action within which the invoker is nested completes, at which point the top–level independent actions will be able to acquire the lock.

5.3.2.2 Examples of unsynchronised top–level independent actions

1) *Logging*: If a resource or a service is accessed by an atomic action and the user of the service is to be charged, the charging information should not be undone by error recovery, if the atomic action accessing the service aborts. The use of unsynchronised top–level independent actions to register the charging information means that the information cannot be recovered, and the process of registering the charging information will not delay the atomic action accessing the service.

2) *Garbage collection*: If a system requires a garbage collection to be performed, it would be undesirable for the garbage collection to be undone if the invoking atomic action aborted. It is also desirable that the application should proceed when enough resources are available, and not wait for the completion of the garbage collection. Thus

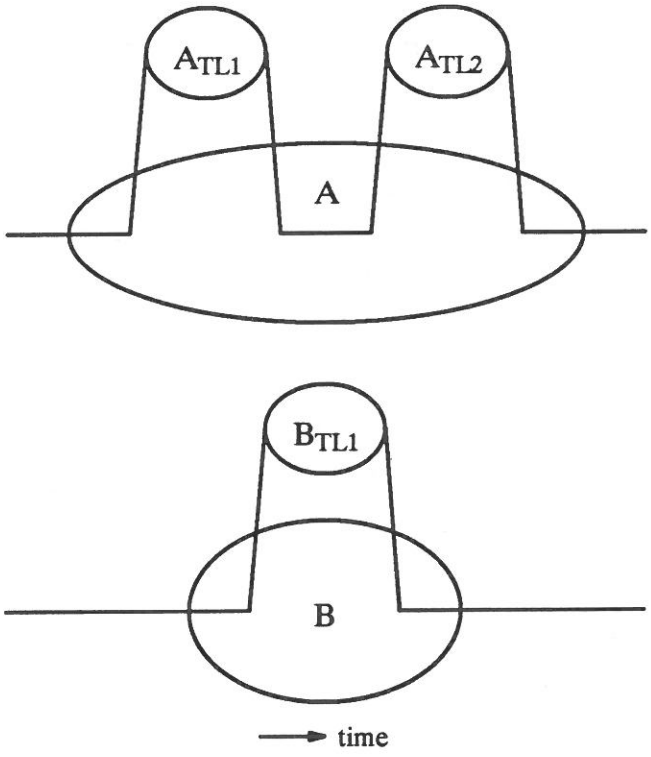
the garbage collection activity should be run as an asynchronous top–level independent action.

5.3.3 Top–level independent actions from within atomic actions

The invocation of a top–level independent action from within an atomic action is likely to mean that the invoking "atomic action" is no longer has the serialisable or failure atomic properties. An atomic action is no longer truly serialised if it invokes a top–level independent action because top–level independent action could be used to pass information between atomic actions.

For example, consider the system of actions illustrated in figure 5–5, in which two "atomic actions" A and B, create a number of top–level independent actions. The action A invokes the top–level independent actions A_{TL1} and A_{TL2}, and action B invokes the top–level independent actions B_{TL1}. If the execution of the system resulted in the top–level independent action B_{TL1} being executed after A_{TL1}, then action A should be serialised to occur before action B. But if the execution of the top–level independent action B_{TL1} occurs before A_{TL2}, then action B should be serialised to occur before action A. Both of these conditions can occur during an execution: this contradiction means that action A and B are not truly serialisable.

Figure 5–5



The invocation of top–level independent actions which will cause irrecoverable effects from within an atomic action clearly makes the atomic action not failure atomic.

But a form of "modular" serialisation and failure atomicity can be maintained if the top–level independent actions access objects in "modules" not accessed by the system of actions invoking them. For example, if a service were provided by a "module" of objects and the billing information for those objects were contained in another "module" of objects, then "module" serialisation and failure atomicity could be maintained by the atomic actions accessing the service. This is not dissimilar to the idea of *atomic data sets* and *setwise serialisability* developed by Sha [Sha 85][Sha 88].

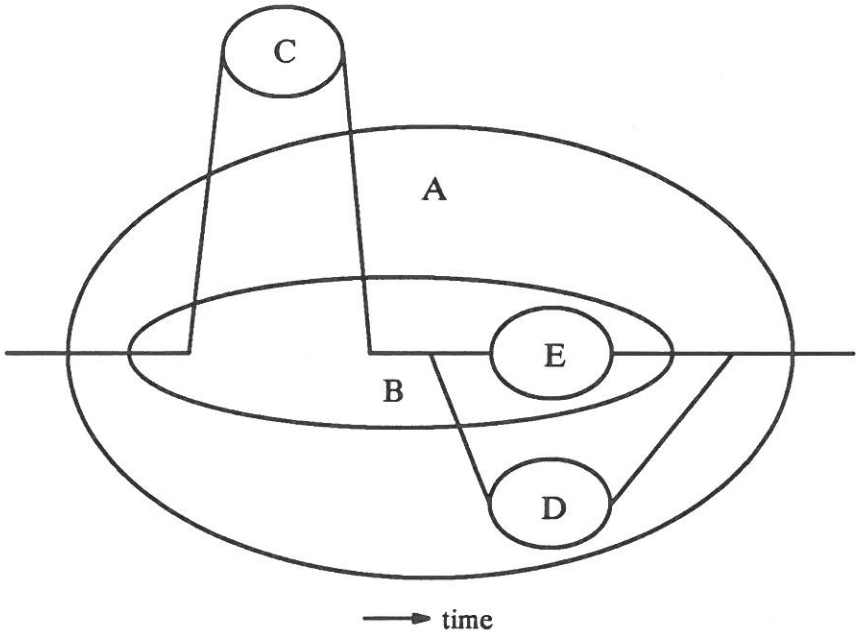
5.3.4 N-level independent actions

An obvious extension to the idea of top-level independent actions is to allow the action to be invoked at any level in the hierarchy of nested actions, instead of just at the top-level. These actions are called n-level independent actions. These n-level independent actions can be either synchronised or unsynchronised.

The effects of n-level independent actions (which are not invoked at the top-level) can be undone by aborting an enclosing atomic action, causing recovery.

An example of a system of actions using n-level independent actions is illustrated in figure 5-6. The system contains two n-level independent actions *C* and *D*. The n-level action *C* has been invoked at the top-level. The n-level independent action *D* has been invoked at one level below the top-level enclosed by the atomic action *A*. Aborting the action *A* would cause the recovery of the effects from the n-level independent action, *D*.

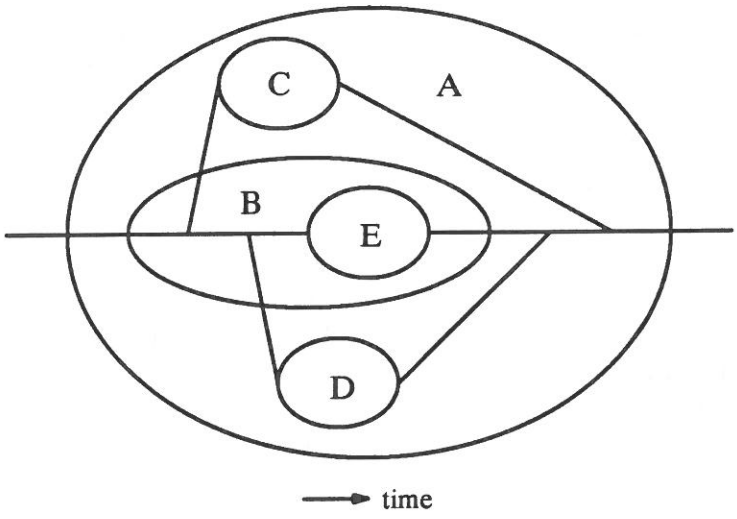
Figure 5-6



5.3.4.1 External Serialisability

The use of n-level independent actions within an action allows the action to be serialisable with other actions at the same level, but to be internally unserialised. This is illustrated in figure 5-7, where action *A* contains unserialisable activities within it.

Figure 5–7



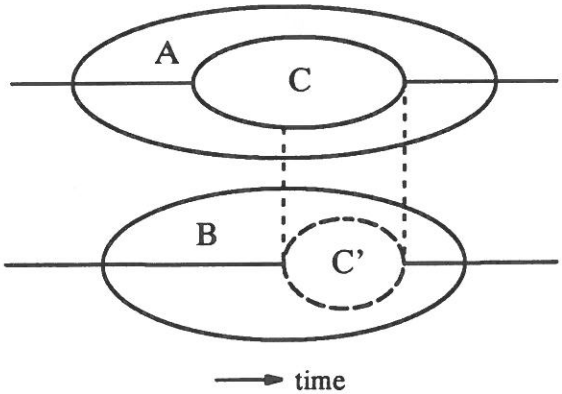
5.4 Common actions

The use of common actions permits efficient use of objects with idempotent operations. Common actions allow one invocation of the action performing an operation to be shared by many invokers, so preventing the operation having to be performed repeatedly. Such idempotent operations are for example: creating or removing a directory entry, marking an object as "temporarily invalid" or "dirty". The use of common actions is most advantageous when the idempotent operation involves changing the state of the object, because normally this would require each action in turn to acquire a write lock on the object.

Because the effect of a common action can be considered to have been caused by any of the actions which were involved in its invocation, the effect should be made permanent if any of the actions involved have their effects made permanent.

An illustration of a common action is given in figure 5–8. Action *B* shares the invocation of *C* by action *A*, via the action *C'*. This means that action *C* need only be performed once. The common action *C* will commit if either the action *A* or action *B* commits. The common action *C* will only be aborted if both action *A* and action *B* abort.

Figure 5–8



To maintain serialisability, copies of each of the locks on an object acquired by a common action are inherited by each of the invokers of the common action upon the completion of the common action. But write locks on objects inherited from the common action do not provide the inheriting actions with the right to modify the objects, unless all other actions involved in the invocation have completed. Because a common action can be

considered to have multiple parent actions, it should not receive any privileges over other actions when attempting to acquire locks which are held by any of its parent actions.

5.4.1 Parameterised common actions

Parameterised common actions are an extension to the idea of common actions in which the common action which is implementing an operation can take a parameter which will govern which other common actions it should be common with. Only invocations of common actions with the same parameter will be common.

5.4.2 Examples

In this section example of the use of common actions will be given.

5.4.2.1 Marking objects "temporarily invalid" using common action

Common actions can be used for marking objects as temporarily invalid. Common actions are suitable because this operation is idempotent. The code below shows how it could be implemented.

```
class ExampleObject {
        . . .
public
        void MarkAsInvalid();
        void MarkAsValid();
        . . .
};
void ExampleObject::MarkAsInvalid()
{
        CommonAction A;

        A.Begin();
                . . .
        A.End();
}
void ExampleObject::MarkAsValid()
{
        CommonAction A;

        A.Begin();
                . . .
        A.End();
}
```

In the above implementation, if any actions wished to mark the same object of class *ExampleObject* as invalid (or valid) at about the same time, then the action which performs the operation can be shared. If other actions later wish to mark the object as invalid (or valid) they could if desired, be considered to have taken part in the original marking of the object as invalid (or valid).

5.4.2.2 Directory objects using parameterised common action

An example of the uses to which parameterised common actions can be put is given in the create and remove entry operations of a directory. If two actions attempt to create or remove the same entry in a directory, the action which performs the task can be common. If common actions are used to implement the operation, it must be parameterised with the entry which is to be created or removed. The definition of such a *Directory* class and its operations are given below. The *Directory* class provides operations which can be either shared or not for the created and removed entry. This lets an application specify whether it wishes share its invocation of an operation or not.

```
class Directory {
        . . .
```

```
public:
        void Create(NameType Name);
        void Remove(NameType Name);

        void SharedCreate(NameType Name);
        void SharedRemove(NameType Name);

        . . .
};
void Directory::SharedCreate(NameType Name)
{
        CommonAction A;

        A.Begin(Name);
                . . .
        A.End();
}
void Directory::SharedRemove(NameType Name)
{
        CommonAction A;

        A.Begin(Name);
                . . .
        A.End();
}
```

The *SharedCreate* and *SharedRemove* operations of the *Directory* objects are performed by common actions, so for example if two action wished to add the entry "MyFile" to the same directory, the common action which performed the operation could be shared. But if two actions wised to add the entries "MyFile1" and "MyFile2" respectively to the same directory, the parameters to the common action being different would mean that a single action could not be shared.

5.5 Glued actions

Glued actions provide a mechanism for increasing the concurrency in systems which contain long running actions.

The locks held by long running actions can decrease the potential concurrency in a system. One method for preventing this is the early release of locks, but this method can cause a cascade of actions to abort, if the releasing action later fails. Glued actions provide a mechanism which allows locks on objects to be released without the possibility of causing the cascade of action aborts.

5.5.1 The problem which glued actions solve

The problem glued actions can solve is shown by a simple example: imagine an atomic action system which contains two atomic actions *A* and *B*, where *B* is a long running atomic action. The purpose of *A* is to modify objects *O1* .. *On*, then choose one of the objects *Oi*. The object *Oi* is passed to *B* which then performs a long and complicated operation on it. This operation does not involve any of the objects *O1* .. *On* other than *Oi*, but may depend on other objects. Object *Oi* should be unchanged between the end of atomic action *A* and the start of atomic action *B*. The effects of atomic action *A* on the objects *O1* .. *On* should not be recovered if atomic action *B* fails, and the objects *O1* .. *On* except *Oi* should be available for use by other atomic actions while atomic action *B* is being performed.

Two possible atomic action systems which could be used to perform this task are given in figures 5–9 and 5–10.

Figure 5–9 shows atomic action *A* simply followed by atomic action *B*, with no enclosing atomic action. This does not satisfy the first requirement because object *Oi* can be changed between the end of atomic action *A* and the start of atomic action *B*.

Figure 5–9

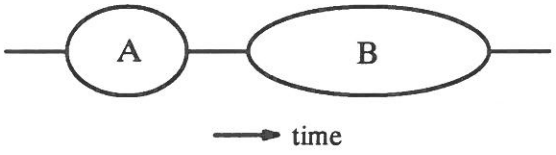
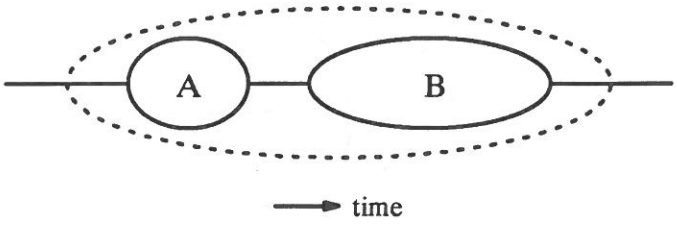


Figure 5–10 shows atomic action *A* simply followed by atomic action *B*, with an enclosing synchronising region. This does not satisfy the second requirement because the locks on the objects *O1* .. *On* except *Oi* cannot be released until the enclosing synchronising region terminates.

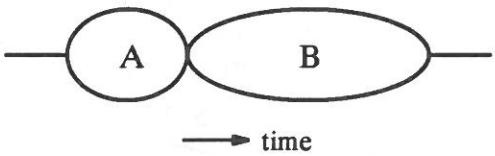
Figure 5–10



The solution is to use *glued actions*, which allow locks to be transferred between an action and the action which immediately follows it. If an action *A* is glued to the action *B*, which follows it, then some of the locks which *A* holds can be transferred to *B*, so preventing the objects being changed between the two actions.

The behaviour of the two glued actions appears no different to some other action in the system, except that an action could never observe the effect of action *A* on certain objects until action *B* has completed. The apparent transition between action *A* finishing and action *B* starting appears "atomic" with respect to some objects. The figure 5–11 illustrates glued actions.

Figure 5–11



A problem arises if the action *B* fails or is involved in deadlock. Then the locks it holds cannot be released and the action *B* retried, because the objects may be changed before the locks are regained. This means that if action *B* fails it should not be retried. Note that the effects of the action *A* still exist.

5.5.2 Concurrent glued actions

Gluing can be performed among concurrent actions, as illustrated in figures 5–12 and 5–13. If concurrent glued actions are used, the concurrent actions (A_1 .. A_n/B_1 .. B_n) must be mutually disjoint in the locks which they obtain, or deadlock will result.

Figure 5–12

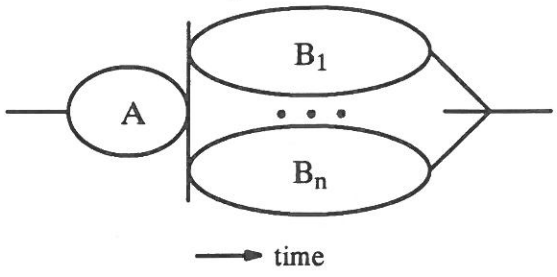
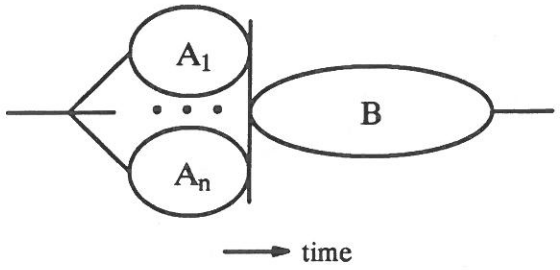


Figure 5–13



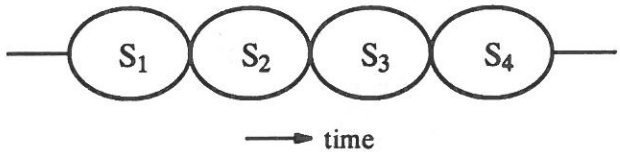
5.5.3 An example: Arranging a meeting

In this section an example of the use of glued actions will be given.

To arrange a meeting, one must select a time which is most suitable for the people taking part in the meeting, after a number of stages where preferences are stated. The problem with implementing this with a conventional atomic action system is that times which are regarded as unsuitable could be locked until the final decision is made, so hindering the arrangement of other meetings.

Using glued actions, action *S1* can lock all the possible times at which the meeting could occur. Phase *S1..S3* select from the available times the more preferred times and pass them on to the the next atomic action. The locks on the other times are released. *S4* selects one of the times it is passed as the time of the meeting. This is illustrated in figure 5–14, which assumes that the problem is broken into actions.

Figure 5–14




```
GluedAction S1, S2, S3, S4;

S1.Begin();                        // Start of glued action S1
      Lock_Initial_Preferred_Times
      Evaluate_Possible_Times
      Totally_Release_Locks_On_Unsuitable_Times
      S1.Transfer( . . . )              // Indicate which locks are to be
                                        // transfered to the next action
S1.End();                               // End of glued action S1
S2.Begin();                             // Start of glued action S2
      Lock_Possible_Times
      Evaluate_Possible_Times
      Totally_Release_Locks_On_Unsuitable_Times
      S2.Transfer( . . . )              // Indicate which locks are to be
                                        // transfered to the next action
S2.End();                               // End of glued action S2

. . .
```

```
S4.Begin();                              // Start of glued action S4
      Lock_Possible_Times
      Decide_On_Meeting_Times
      Totally_Release_Locks_On_Unsuitable_Times
      Set_Meeting_Time
S4.End();                                // End of glued action S4
```

5.6 Summary

In this section it was demonstrated that nested atomic actions are not suitable for the construction of all applications, due to the properties of serialisability and failure atomicity, in some cases being over restrictive on the structure of the application.

To overcome some of these problems, new structuring techniques of: serialising actions, glued actions, common actions, parameterised common actions, top–level independent actions and n–level independent actions, were introduced.

The structuring techniques described in this section could be implemented using a different mechanism for each of the techniques. But in the following section, an uniform mechanism which can be used to implement serialising actions, glued actions, top–level independent actions and n–level independent actions, will be described.

6 Coloured actions

It has been shown in the previous section that new structuring techniques are required to solve some of the problems involved with the implementation of systems composed of atomic actions as basic entities. Rather than design various mechanisms to implement each of these structures, the following section will describe the concept of coloured actions, and show how they can be used to implement the structures.

Coloured actions are formed by slightly changing the properties of atomic actions to create a form of action which can be used in a more flexible manner.

6.1 Introduction

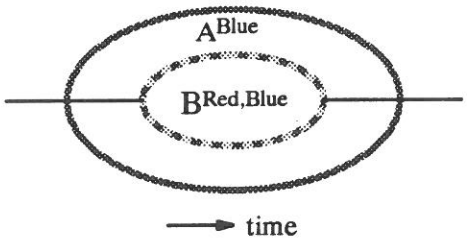
The basic unit of work in a *coloured system* is a *coloured action*. The use of coloured actions gives the implementor of a system a uniform and more flexible mechanism for structuring the systems than using atomic actions. The coloured system can be structured so that its component parts are either serialised or non-serialised with respect to each other, as required.

The operations performed by a coloured action are controlled so that they are serialised with respect to all operations performed by coloured actions of the same colour.

The modifications performed by coloured actions to objects are made permanent when the outermost enclosing coloured action which is the same colour as the modifying action commits. This means that the effects of nested coloured action may be made permanent even if enclosing coloured actions are aborted.

Before specifying the behaviour of coloured actions formally, a simple example will be given to hint at their behaviour. Figure 6–1 shows two coloured actions, coloured action A which is coloured *Blue* and coloured action B which is coloured both *Red* and *Blue*. Coloured action B is nested within coloured action A (The colours of the actions being represented by different levels of grey).

Figure 6–1



Suppose the coloured action B modifies the set of objects O_{red} using the colour *Red*, and also modifies the set of objects O_{blue} using the colour *Blue*. After B commits all modifications to objects in set O_{red} will become permanent, but the modifications to the objects in set O_{blue} will only become permanent when (and if) coloured action A commits. Coloured action B behaves as a top-level action as far as objects in O_{red} are concerned but like a nested action as far as objects in O_{blue} are concerned. So if the coloured action A aborts after coloured action B has committed, only the modifications to objects in O_{blue} will be undone.

In the following sections, how coloured actions can be used to construct systems which match the concurrency and recovery requirements of applications, will be discussed.

6.2 Properties of coloured actions

The properties of coloured actions are intended to permit more flexible concurrency and recovery than atomic actions. Coloured actions have properties which correspond to

the atomic action properties of serialisability, 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 (So in a coloured action system modifications performed can be thought of as having colours).

2) *Serialisability*: the execution of concurrent coloured actions should produce the same effects as some serial order execution of the actions with respect to each of the colours possessed by 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 by that action to the system state 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.

6.3 Coloured actions implemented using locks

The properties stated in the previous section do not force coloured action systems to be implemented using locks, but in the rest of this section this will be assumed to make the discussion more specific.

Coloured actions implemented using locks are similar to atomic actions with strict two phase locking, except for the way in which locks are inherited by an action. The inheritance mechanism is affected by the colour of the lock which is being released. In a coloured system locks would not only have modes (shared and exclusive) but also have a colour which must be specified when they are acquired. A lock must be acquired with one of the colours of the requesting action. In a coloured system many locks of differing colours and modes may be held on an object at an instance in time.

The power of coloured actions comes from allowing a coloured action to have more than one colour, along with the manipulation of the colours and modes of the locks which are set on objects. This enables a system of coloured actions to be constructed which will match the concurrency and recovery requirements of an application.

6.3.1 Locking rules

To show the difference between atomic actions and coloured actions, it is necessary to compare their locking rules.

The locking rules for atomic actions and coloured actions will be examined for three types of locks: read, write and exclusive read. A read lock on an object allows the holder to read the object. A write lock on an object allows the holder to write the object. An exclusive read lock on an object allows the holder to read the object, and no other action can acquire a lock on that object in any mode until the exclusive read lock is released. The holder of an exclusive read lock on an object can acquire (or convert the lock to) a write lock on that object without the possibility of a lock conflict.

The locking rules will specify the conditions under which locks may be granted on objects, and also the behaviour of atomic actions/coloured actions when they commit or abort. In conventional atomic action systems [Moss 81], the locking rules are as described below:

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

- An atomic action may acquire a lock on an object in a read mode 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 an 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 action 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 one of 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 holders of the locks (of all colours and modes) on that object are ancestors of the requesting coloured action, and all write locks held on the objects are of the colour 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 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 coloured action.
- A coloured action may acquire a lock on an object in an exclusive read mode if all holders of locks (of all colours and modes) on that object are ancestors of the requesting coloured action.
- When a coloured action with colours α_1 .. α_2 commits, the locks of colour α_i, $1 <= i <= n$ are inherited by the most recent ancestor (if any) with colour α_i. This means that the ancestor retains the locks in the same mode as the coloured action held them.
- When a coloured action aborts, all locks held (of all colours and 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.

6.3.2 Example of lock inheritance

This example shows how locks are inherited in a coloured action system, where an action can possess more than one colour. In the example, coloured action *B* is nested within the coloured action *A*. Both actions acquire locks, action *A* being coloured *red* and *blue*, and action *B* being coloured *red* and *green*.

For the sake of illustration a simple notation (similar to an Arjuna program) will be used to specify the coloured action systems. In this notation colours and coloured actions are objects. The coloured action objects possess operations *Begin*, *End* and *Abort* which respectively start, commit and abort the coloured action. The *Begin* operation of coloured action is parameterised with the colours the action will possess. Colour objects can also be passed as parameters to operations to specify the colour that locks should be obtained with.

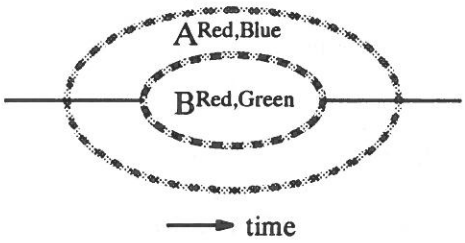
```
Data a, b, c, d, e;
```

```
Colour Red, Blue, Green;

ColouredAction A, B;
. . .                              // Point #1

A.Begin(Red, Blue);                // Start of coloured action A

      a.Change(Red);               // Operation causing Red write lock on a
      b.Inspect(Red);              // Operation causing Red read lock on b
      c.Change(Blue);              // Operation causing Blue write lock on c

      . . .                        // Point #2

      B.Begin(Red, Green);         // Start of coloured action B
            a.Change(Red);         // Operation causing Red write lock on a
            d.Inspect(Green);      // Operation causing Green read lock on d
            e.Change(Red);         // Operation causing Red write lock on e

            . . .                  // Point #3

      B.End();                     // End of coloured action B

      . . .                        // Point #4

A.End();                           // End of coloured action A

. . .                              // Point #5
```

In the above code the operations of *Change* and *Inspect* of the class *Data* will set either a write or a read lock respectively, of the colour which is passed as a parameter to the operation. The lock on an object will be set using the *Lock* operation which all classes will be assumed to possess (This operation will be used explicitly in later examples), the operation takes as a parameter a mode (read, write or exclusive read), and a colour.

The diagram in figure 6–2 illustrates the computation invoked by the above program.

Figure 6–2



The table below shows the status of the locks at certain points in the program.

Point #1	Held	:	None	
	Released	:	None	
Point #2	Held	:	a(Red)	by Coloured Action A
			b(Red)	by Coloured Action A
			c(Blue)	by Coloured Action A
	Released	:	None	
Point #3	Held	:	a(Red)	by Coloured Action A
			b(Red)	by Coloured Action A
			c(Blue)	by Coloured Action A
			d(Green)	by Coloured Action B
			e(Red)	by Coloured Action B
	Released	:	None	
Point #4	Held	:	a(Red)	by Coloured Action A
			b(Red)	by Coloured Action A
			c(Blue)	by Coloured Action A

```
                                e(Red)      by Coloured Action A
            Released    :       d(green)    by Coloured Action B committing
Point #5    Held        :       None
            Released    :       a(Red)      by Coloured Action A committing
                                b(Red)      by Coloured Action A committing
                                c(Blue)     by Coloured Action A committing
                                d(green)    by Coloured Action B committing
                                e(Red)      by Coloured Action A committing
```

Note that the coloured action *A* did not inherit the *green* lock on the object *d* which was held by the coloured action *B*.

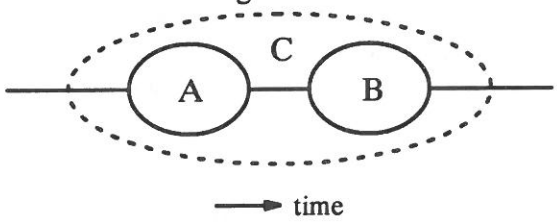
6.4 Structuring using coloured actions

In the following sections it will be shown how coloured actions can be used to implement: serialising actions, glued actions, top–level independent actions and n–level independent actions. It will be assumed that an object's interface specification will indicate the mode (read or write) of the lock required on the object, by each of the object's operations.

6.4.1 Implementing serialising actions

In the action system shown in figure 6–3, actions *A* and *B* are nested within a serialising action *C*; this means that the effects of the action *A* will be made permanent when action *A* commits, and will not be recovered even if the action *B* is unable to commit. The locks acquired by actions *A* and *B* are not be made available to other actions until serialising action *C* finishes.

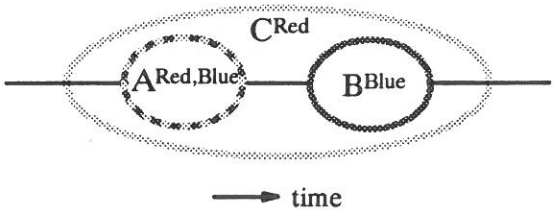
Figure 6–3



To provide the behaviour obtained from serialising actions using coloured actions is simple and straightforward. What is required is an enclosing coloured action, and for objects which were locked by the first coloured action to be additionally locked in the colour of this enclosing action. The purpose of the additional lock is to: prevent objects which were read locked by the first coloured action from being modified between the end of the first coloured action and the start of the second coloured action, and to prevent objects which were write locked by the first coloured action from being modified or read between the end of the first coloured action and the start of the second coloured action. To ensure that the objects read locked by the first coloured action are not modified, an additional read lock is sufficient, but objects write locked by the first coloured action are required not to be modified or read (by anyone other than *B*), so they must be additionally locked with an *exclusive read* lock.

Figure 6–4 illustrates such a system, where the coloured action *C* is coloured *red*, coloured action *A* is coloured *red* and *blue*, and *B* is coloured *blue*. The objects used by action *A* are locked with a lock coloured *blue*, in addition any objects modified by action *A* are also locked with a *red* exclusive read lock, and any objects read by action *A* are locked with a *red* ordinary read lock. The locks coloured *red* will be inherited by the coloured action *C*, so preventing other actions from modifying them. In addition, the inherited exclusive read locks prevent the corresponding objects from being read. The write locks obtained on objects by action *A* will be released, so causing the modifications to become permanent. Thus action *A* appears 'top–level' as far as objects with only *blue* locks are concerned.

Figure 6-4



The code below shows such a system:

```
Data a, b, c, d;

Colour Red, Blue;

ColouredAction A, B, C;
. . .
C.Begin(Red);                       // Coloured action C start

      A.Begin(Red, Blue);           // Coloured action A start
            a.Inspect(Blue);        // Operation causing Blue read lock on a
            b.Change(Blue);         // Operation causing Blue write lock on b
            c.Change(Blue);         // Operation causing Blue write lock on c
            d.Inspect(Blue);        // Operation causing Blue read lock on b

            // Acquire the locks on the object which are to be passed
            // to B unchanged.

            a.Lock(read, Red);      // Set Red read lock on a
            b.Lock(xread, Red);     // Set Red exclusive read lock on b
            c.Lock(xread, Red);     // Set Red exclusive read lock on c
            d.Lock(read, Red);      // Set Red read lock on d
      A.End();                      // Coloured action A end

      // New states of b and c made stable; a and d available in
      // read mode for other actions

      B.Begin(Blue);                // Coloured action B start
            a.Change(Blue);
            c.Inspect(Blue);
      B.End();                      // Coloured action B end

C.End();                            // Coloured action C end
```

The above code shows how coloured actions can be used to implement serialising actions. In this example the colour *blue* can be regarded as the colour used by "atomic actions" and the colour *red* as an additional colour used to provide the serialising actions.

Explained above is how serialising actions can be provided for atomic actions using coloured actions, but serialising actions are also very useful for structuring systems of coloured actions. A coloured serialising action would allow coloured actions of a particular colour to be serialised. Coloured serialising actions could be implemented by introducing another colour, in a similar manner in the example above, or the implemention of coloured action could also provide coloured serialising action, so making the writing of code simpler. For example, the implemention of coloured actions could provided a coloured serialising action using an additional *ColouredSerialisingAction* class.

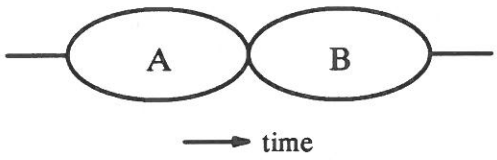
```
ColouredSerialisingAction C;

Colour Red, Blue;

ColouredAction A, B;

C.Begin(Blue);                      // Coloured serialising action C start
```

```
A.Begin(Blue);                  // Coloured action A start
      a.Inspect(Blue);          // Operation causing Blue read lock on a
      b.Change(Blue);           // Operation causing Blue write lock on b
      c.Change(Blue);           // Operation causing Blue write lock on c
      d.Inspect(Blue);          // Operation causing Blue read lock on d
A.End();                        // Coloured action A end

B.Begin(Blue);                  // Coloured action B start
      a.Change(Blue);
      c.Inspect(Blue);
B.End();                        // Coloured action B end

C.End();                        // Coloured serialising action C end
```

6.4.2 Implementing glued actions

In the action system shown in figure 6–5, the action *A* is glued to action *B*, this means that locks can be transferred from action *A* to action *B*, without other actions being able to acquire these locks.

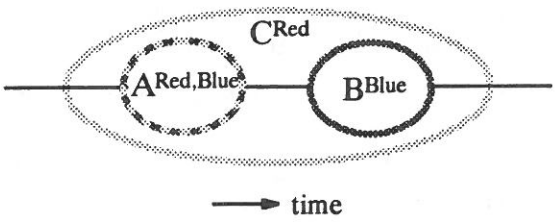
Figure 6–5



To implement the same behaviour using coloured actions, all one requires is an additional enclosing coloured action, and for the objects whose locks are to be transferred to be read locked (in particular exclusive read locks for transferring write locks) in the colour of this enclosing action.

Figure 6–6 illustrates such a glued system, where the coloured action *C* is coloured *red*, coloured action *A* is coloured *red* and *blue*, and *B* is coloured *blue*. The objects used by action *A* are locked with a lock coloured *blue*. Any objects accessed by action *A* which are to be transferred from action *A* to action *B* unchanged should be locked with a read lock coloured *red* if an object has not been modified, and an exclusive read lock coloured *red* if an object has been modified. The locks coloured *red* will be inherited by the coloured action *C*, so preventing other actions form acquiring them. The write locks obtained on objects by action *A* will be released, so causing the modifications to become permanent.

Figure 6–6



The code below could be used to implement such a system of two glued actions:

```
Data a, b, c, d;

Colour Red, Blue;

ColouredAction A, B, C;
. . .
C.Begin(Red);                   // Coloured action C start

    . . .
    A.Begin(Red, Blue);         // Coloured action A start
```

```
            a.Change(Blue);          // Causes Blue write lock on a
            b.Inspect(Blue);         // Causes Blue read lock on b
            c.Change(Blue);          // Causes Blue write lock on c

            . . .
            // Acquire the locks on the object which are to be transfered
            // to B unchanged.

            a.Lock(xread, Red);      // Set Red exclusive read lock on a
            b.Lock(read, Red);       // Set Red read lock on b
      A.End();                       // Coloured action A end

      . . .
      B.Begin(Blue);                 // Coloured action B start

            a.Inspect(read,  Blue);
            b.Inspect(read,  Blue);
            d.Change(write, Blue);

            . . .

      B.End();                       // Coloured action B end

      . . .

C.End();                             // Coloured action C end
. . .
```

In the above system action *A* is glued to action *B* and the locks on objects *a* and *b* are transferred from action *A* to action *B*.

6.4.2.1 Implementing concurrent glued actions

Coloured actions can also be used to implement concurrent glued actions. The way in which colours are used to implement concurrent glued actions is identical to the way in which glued actions are implemented. The way in which colours are used is illustrated in figures 6–7 and 6–8.

Figure 6–7

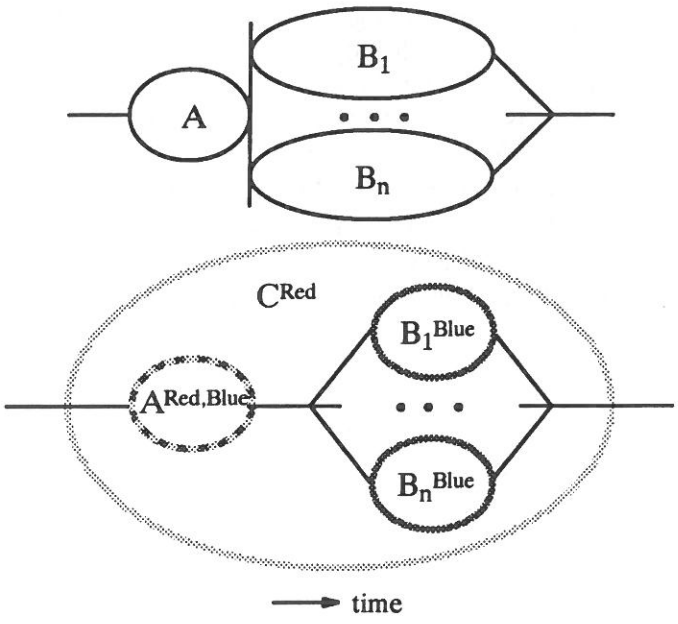
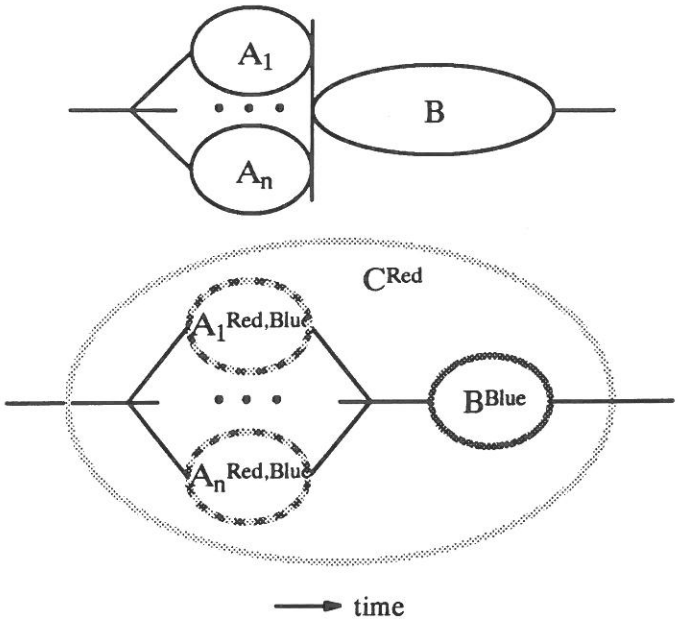


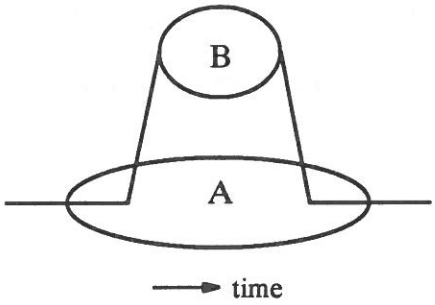
Figure 6–8



6.4.3 Implementing top-level independent actions

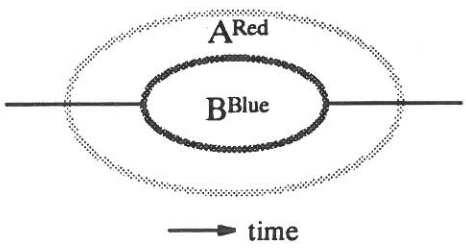
In the action system shown in figure 6–9, the action *A* invokes a top–level independent action *B*, which means that effects of action *B* will be made permanent even if action *A* is aborted.

Figure 6–9



This effect can be implemented using coloured actions, by using a nested coloured action of a different colour to its enclosing action. The locks which are released by the nested coloured action will not be inherited by the enclosing coloured action, so its effects will be made permanent.This is illustrated in figure 6–10 where action *A* could be coloured *red* and action *B* be coloured *blue*. Note that the system in figure 6–10 will only behave identically to the system in figure 6–9, provided that *B* does not need conflicting access to any objects locked by *A*. If this is the case, then *A* and *B* in the system depicted in figure 6–9 would be deadlocked, whereas this would not happen for the coloured system (but of course, *B* would then not strictly be a top–level independent action).

Figure 6–10



The code below would be used to implement such a system:

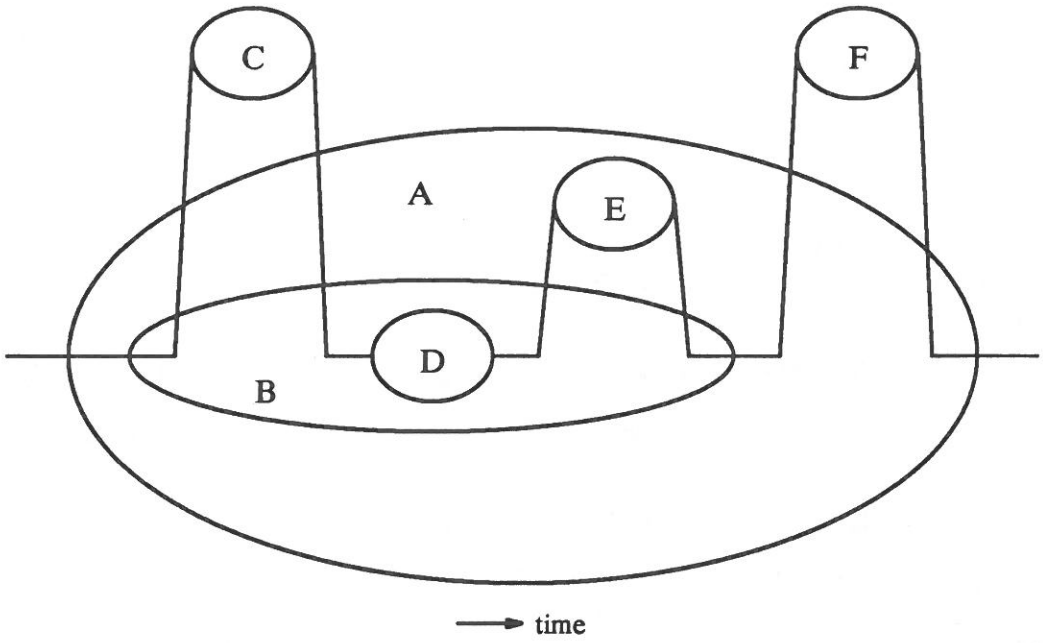
```
Data a, b;

Colour Red, Blue;

ColouredAction A, B;
. . .
A.Begin(Red);                                   // Coloured Action A start.
      . . .
      B.Begin(Blue);                            // Coloured Action B start.
            a.Change(Blue);
            b.Inspect(Blue);
            . . .
      B.End();                                  // Coloured Action B end.
      . . .
A.End();                                        // Coloured Action A end.
. . .
```

The modification of object *a* by action *B* will be made permanent even if action *A* aborts.

6.4.4 Implementing n-level independent actions

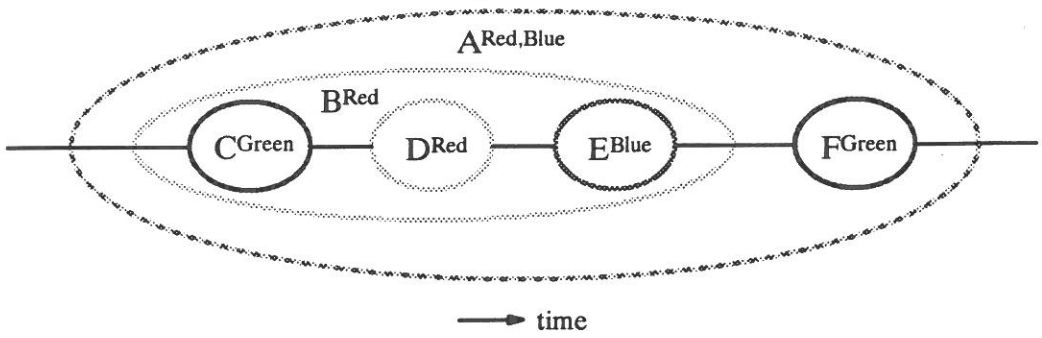
In the action system shown in figure 6–11, the actions *C*, *E* and *F* are n-level independent actions. The effects of these three actions will be made permanent even if their invoking action aborts. If either of the actions *C* or *F* commits, then its results will become permanent. But, for the effects of action *E* to become permanent, not only must it commit but also action *A* must commit, action *E* not being a top-level action.

Figure 6–11



N-level independent actions can be implemented using coloured actions, by using nested coloured actions of differing colours. By choosing the colours of the actions in the system the level of nesting can be controlled. This is illustrated in figure 6–12 which shows how the system in figure 6–11 could be implemented using coloured actions.

Figure 6–12



The code below could be used to implement the system given in figure 6–12:

```
Colour Red, Blue, Green;

ColouredAction A, B, C, D, E, F;
. . .
A.Begin(Red, Blue);                          // Coloured Action A start.
      . . .
      B.Begin(Red);                          // Coloured Action B start.
            . . .
            C.Begin(Green);                  // Coloured Action C start.
                  . . .
            C.End();                         // Coloured Action C end.
            . . .
            D.Begin(Red);                    // Coloured Action D start.
                  . . .
            D.End();                         // Coloured Action D end.
            . . .
            E.Begin(Blue);                   // Coloured Action E start.
                  . . .
            E.End();                         // Coloured Action E end.
            . . .
      B.End();                               // Coloured Action B end.
      . . .
      F.Begin(Green);                        // Coloured Action F start.
            . . .
      F.End();                               // Coloured Action F end.
      . . .
A.End();                                     // Coloured Action A end.
. . .
```

Note that actions *C* and *F* are top–level *green* actions, and that the *blue* action *E* is nested within the *blue* action *A*, but its effects will not be recovered by the aborting of the *red* action *B*.

6.5 Example application using coloured actions

In this section examples of applications will be developed to illustrate the power of coloured actions.

6.5.1 Pseudo random number generator

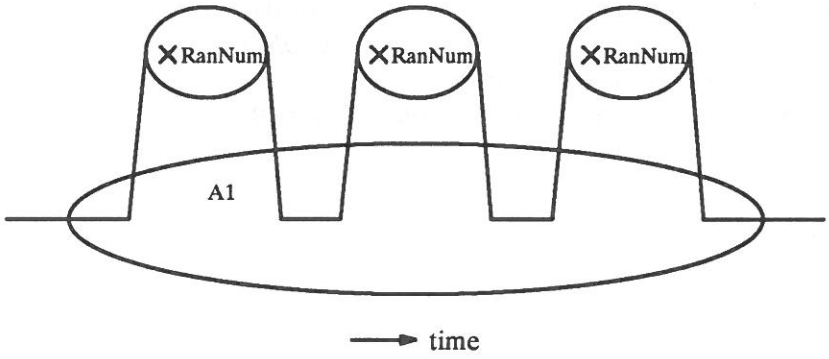
The pseudo random number generator described below is an example of an application using coloured actions and locks. This example will be used to demonstrate the way in which coloured actions can be used to provide flexible concurrency and recovery control on objects, and show how different colours of coloured actions interact. The example consists of a class *Random*. A user of an object of class *Random* may decide that the sequence of random numbers produced should be successive random numbers produced from the seed, or a user may decide that the random number need not be produced in this manner.

A user may decide that if the random number seed of the object is set by an action, then that action should obtain the sequence of random numbers which would be produced from that seed (i.e. not allow another action to set the seed), but allow other actions to get the value of the seed. A user may alternatively decide that other actions are not allowed to get the value of the seed. The class *Random* will possess a constructor, and three operations *RanNum()*, *GetSeed()* and *SetSeed()*. Each of these operations will be performed by a coloured action.

To illustrate the power of coloured actions, four systems of actions will be constructed using the operations of the class *Random*. The different systems will be produced by varying the environment in which operations are invoked, not by changing the operations of the class *Random*.

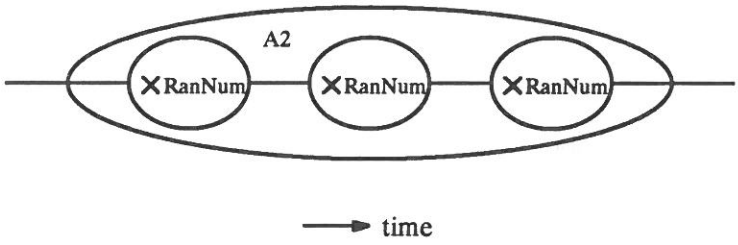
The first system of actions behaves is if each invocation of *RandNum()* gives rise to a top–level independent action, this is illustrated in figure 6–13. This means the pseudo random numbers obtained are not guaranteed to be successively produced from a seed.

Figure 6–13



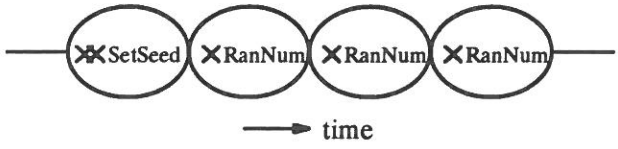
The second system behaves is if each invocation of *RandNum()* gives rise to a nested action, this is illustrated in figure 6–14. This means the pseudo random numbers obtained are guaranteed to be successively produced from a seed.

Figure 6–14



The third system behaves is if the invocations of *SetSeed()* and *RandNum()* gives rise to a series of glued actions, where the read lock obtained by *SetSeed()* on the *Seed* is passed between all the glued actions, and all other locks are immediately released by the acquiring action. This is illustrated in figure 6–15. This means the pseudo random numbers obtained are guaranteed to be successively produced from the specified seed, and the seed may be inspected by other actions.

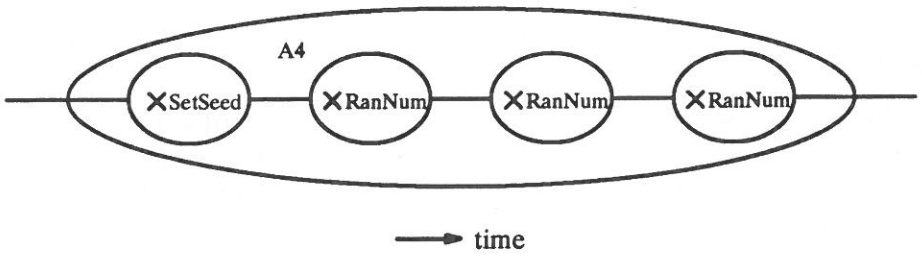
Figure 6–15



The final system behaves is if the invocations of *SetSeed()* and *RandNum()* gives rise to nested actions, this is illustrated in figure 6–16. This means the pseudo random numbers

obtained are guaranteed to be successively produced from the specified seed, and the seed may not be inspected by other actions, until the last pseudo random number was been obtained.

Figure 6–16



The operations of the class *Random* may acquire locks of two colours: *Global* and *Local*, on the object they are invoked upon. The colour *Local* is used for concurrency and recovery control for the operations of the class *Random*, and for no other part of the system, whereas the colour *Global* is used for concurrency and recovery control for all other parts of the system, and for concurrency control for the operations of the class *Random* which set the seed. The fact that both the *Global* and *Local* colours are used for concurrency control of the operations which set the seed, allows the user to specify the concurrency behaviour by varying the colours of enclosing coloured actions.

```
Colours Global, Local;

class Random {
      long Seed;
public:
      Random(long);
      short RanNum();
      long GetSeed();
      void SetSeed(long);
};
Random::Random(long val)
{
      ColouredAction A;

      A.Begin(Global, Local);
            Lock(write, Local);
            Lock(read,  Global);

            Seed = val MOD 4321;
      A.End();
}
short Random::RanNum()
{
      ColouredAction A;
      short result;

      A.Begin(Local);
            Lock(write, Local);

            Seed = (1234*Seed) MOD 4321;
            result = (Seed MOD 10) + 1;
      A.End();

      return result
}

long Random::GetSeed()
{
      ColouredAction A;
```

```
        long result;

        A.Begin(Local);
              Lock(read, Local);

              result = Seed;
        A.End();

        return result
}

void Random::SetSeed(long val)
{
        ColouredAction A;

        A.Begin(Global, Local);
              Lock(write, Local);
              Lock(read,  Global);

              Seed = val MOD 4321
        A.End();
}
```

The *Lock* operation in the above example will set a lock on the instance of the class *Random* from which it is invoked.

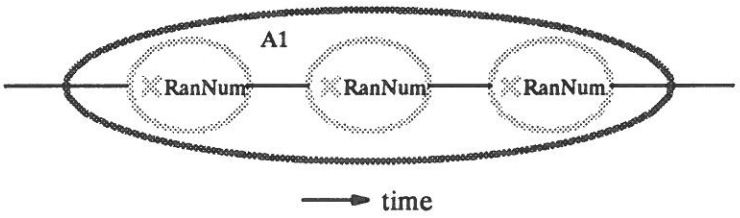
The special characteristics of using coloured actions will be shown by the following examples. Note that in this example, the operations *RanNum()* and *GetSeed()* do not acquire a lock on the object they are invoked upon with colour *Global*. Also note that *SetSeed()* obtains a read lock of colour *Global* whereas it obtains a write lock of colour *Local*. This will allow the user to specify the concurrent behaviour of the object of the class *Random*.

In the first system, the coloured action *A1* is coloured *Global*. This means that it will not retain the lock acquired by the operation *RanNum()* so the pseudo random numbers are not guaranteed to be successively produced from a seed. The coloured action system resulting from the invocation of coloured action *A1* is illustrated in figure 6–17.

```
Random R(10);
ColouredAction A1, A2, A3 ,A4;
short  r1;
short  r2;
short  r3;
A1.Begin(Global);
      r1 = R.RanNum();
      r2 = R.RanNum();
      r3 = R.RanNum();
A1.End();
```

In the diagrams, continuous ovals represent coloured actions, and crosses represent the acquiring of a lock (in read or write mode) on the object, performed by the named operation.

Figure 6–17

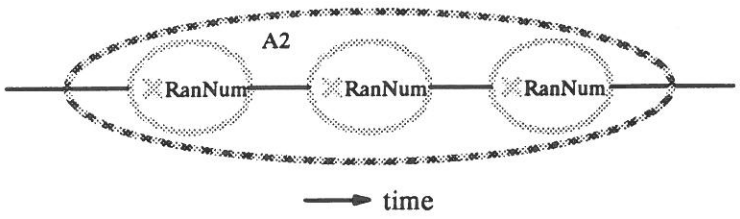


In the second system, the coloured action *A2* is coloured both *Global* and *Local*. This means that the lock acquired by *RanNum()* will be retained by coloured action *A2*, so the

pseudo random numbers are guaranteed to be successively produced from a seed. The coloured action system resulting from the invocation of coloured action *A2* is illustrated in figure 6–18.

```
A2.Begin(Global, Local);
      r1 = R.RanNum();
      r2 = R.RanNum();
      r3 = R.RanNum();
A2.End();
```

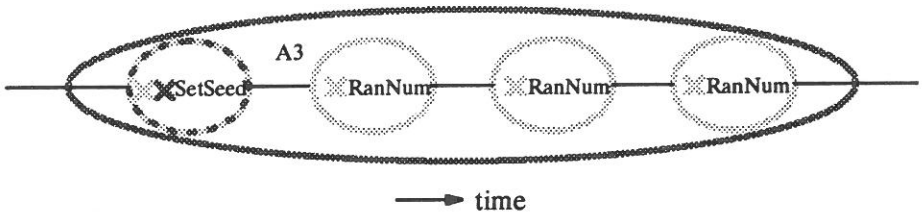
Figure 6–18



In the third system, coloured action *A3* is coloured *Global* but because the operation *SetSeed()* acquires a read lock with colour *Global* the pseudo random numbers produced are guaranteed to be successively produced from a seed. But the write lock acquired by *SetSeed()* is not retained by the coloured action *A3* because the lock was coloured *Local*. This means that other actions can invoke the *GetSeed()* operation on the object and get the value of the seed. The coloured action system resulting from the invocation of coloured action *A3* is illustrated in figure 6–19 (The coloured action which performs the *SetSeed()* operation contains two crosses because the *SetSeed()* operation obtains two locks on the *Seed*, one a read lock, the other a write lock.).

```
A3:   A.Begin(Global);
            R.SetSeed(100);

            r1 = R.RanNum();
            r2 = R.RanNum();
            r3 = R.RanNum();
      A.End();
```

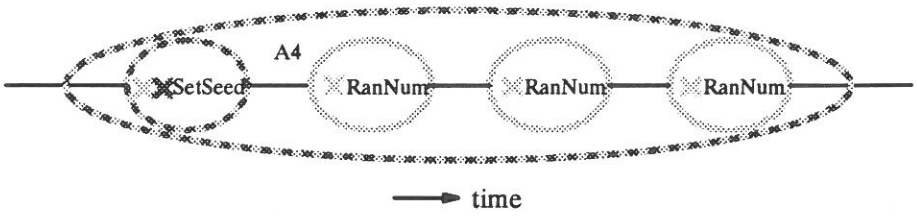
Figure 6–19



In the final system, coloured action *A4* is coloured *Global* and *Local*. This means that both the read lock coloured *Global* and write lock coloured *Local* acquired by *SetSeed()* by will be retained by the coloured action *A4*. This means that other actions cannot invoke the *GetSeed()* operation on the object and get the value of the seed. The coloured action system resulting from the invocation of coloured action *A4* is illustrated in figure 6–20.

```
A4:   A.Begin(Global, Local);
            R.SetSeed(100);

            r1 = R.RanNum();
            r2 = R.RanNum();
            r3 = R.RanNum();
      A.End();
```

Figure 6–20

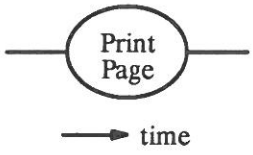


As shown by the systems above coloured actions provide a powerful mechanism of structuring systems to provide different forms of behaviour at the user level.

6.5.2 Pre-emptible print service

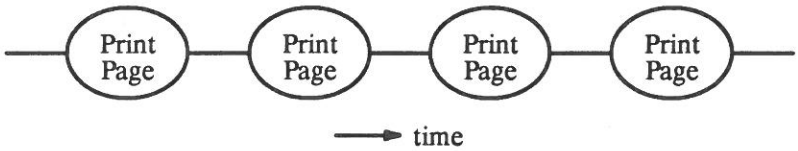
Assume that a system provides an object which has an operation which allows the printing of a single page on a print device where pages are not connected (e.g. laser printer). This operation is performed as an atomic action, which if successful will have modified the object (printed the page). This is illustrated in figure 6–21.

Figure 6–21



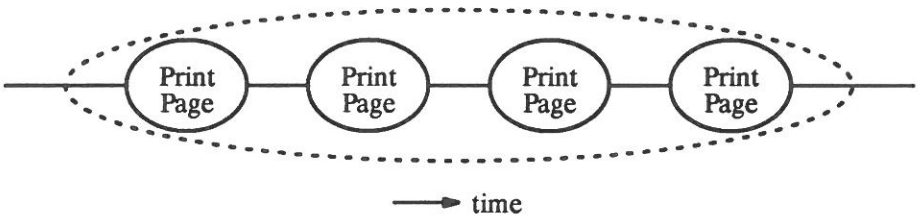
If a document is to be printed, a series of actions can be performed. This system is illustrated in figure 6–22.

Figure 6–22



But if two documents are to be printed concurrently using the above system, the pages of the two documents could be arbitrarily interleaved. To prevent this, a serialising action is required, is illustrated in figure 6–23.

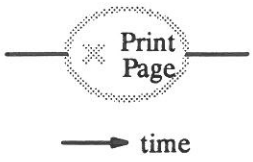
Figure 6–23



The system discussed above would act in a similar way to most print services, but it suffers from the problem that if a large document is to be printed it could cause a delay in the printing of other documents. What is needed is a print service which allows the printing of documents to be pre-emptible, thus allowing, for example, a small document to be printed "in the middle" of another large document, but still maintaining the non-interleaving of small documents and the non-interleaving of large documents. It is not possible to construct such a service using just nested atomic actions, so the implementation of the service using coloured actions will be investigated.

Suppose the system provides an object which has an operation which prints a page, and the operation is performed as a coloured action, and the coloured action is *"coloured red"*. This is illustrated in figure 6–24.

Figure 6–24



The implemention of the object and operation which provide the page printing service would be similar to:

```
Class PagePrintService
{
      . . .
Public:
      Void PrintPage(Page PageToBePrinted);
};

PagePrintService::PrintPage(Page PageToBePrinted)
{
      ColouredAction A;

      A.Begin(Red);
            setlock(Write, Red);
      . . .
      A.End();
}
```

From the object which allows the printing of a single page, a new object can be constructed which also allows the printing of whole documents, the printing of these documents being either pre–emptible or non–pre–emptible.

Figure 6–25 shows the system of coloured actions which is required to allow the printing of non–pre–emptible documents.

Figure 6–25

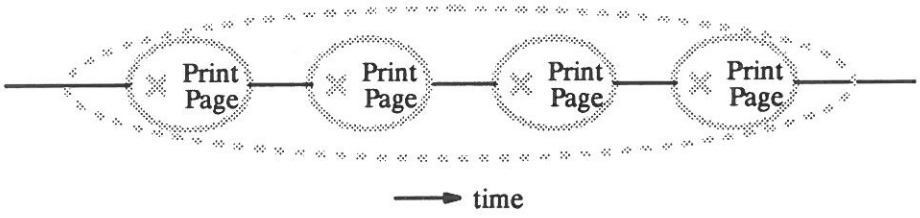
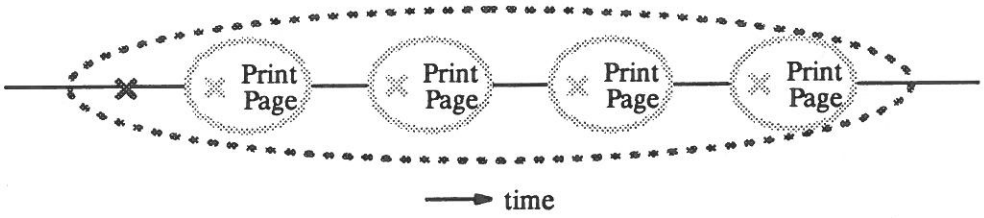


Figure 6–26 shows the system of coloured actions which is required to allow the printing of pre–emptible documents.

Figure 6–26



The dashed ovals in the diagrams above represent coloured serialising actions.

For comparison figure 6–27 shows the corresponding system which uses atomic actions and serialising actions to allow the printing of non–pre–emptible documents, and figure 6–28 shows the corresponding system which uses top–level independent action and serialising actions to allow the printing of pre–emptible documents.

Figure 6–27

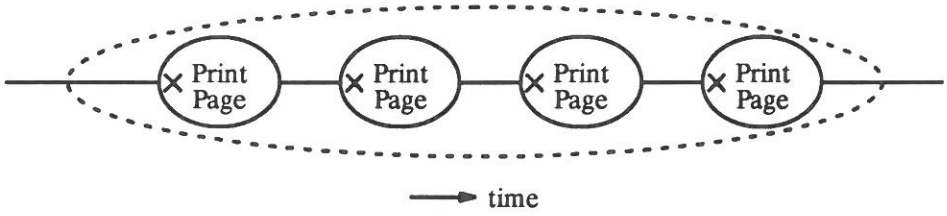
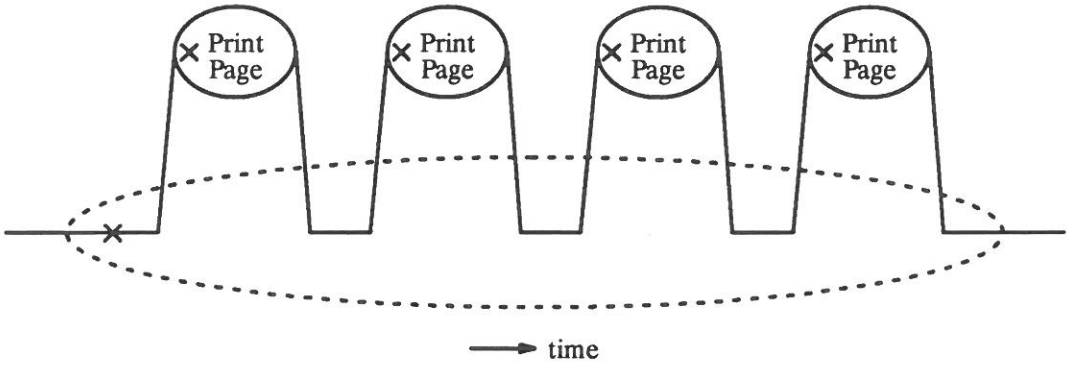


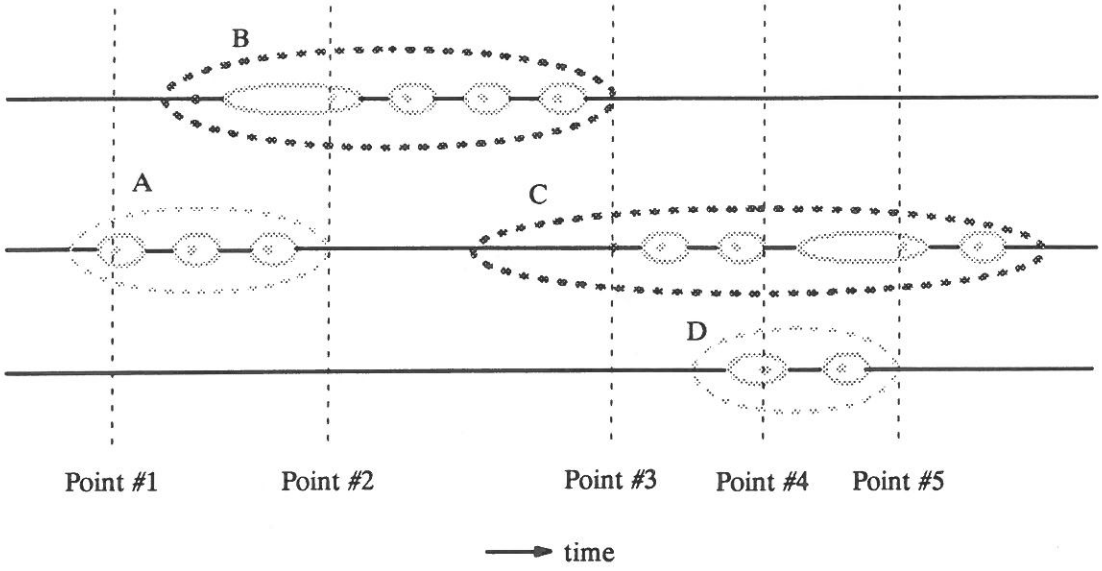
Figure 6–28



The implemention of the object and operations which provide the document printing service using coloured actions would be similar to:

```
Class DocumentPrintService : PagePrintService
{
      . . .
Public:
      void PrintShortDocument(Document DocumentToBePrinted);
      void PrintLongDocument(Document DocumentToBePrinted);
};

DocumentPrintService::PrintShortDocument(Document DocumentToBePrinted)
{
      ColouredSerialisingAction A;

      A.Begin(Red);
            Loop
                  PrintPage(PresentPage);
                  . . .
            Exit When PrintedDocument
                  . . .
            EndLoop
      A.End();
}

DocumentPrintService::PrintLongDocument(Document DocumentToBePrinted)
{
      ColouredSerialisingAction A;

      A.Begin(Blue);
            setlock(Write, Blue);
            . . .
            Loop
                  PrintPage(PresentPage);
                  . . .
            Exit When PrintedDocument
                  . . .
            EndLoop
      A.End();
}
```

Figure 6–29



Point #1 A write lock is obtained on the *page print object* by an action nested within the action *A*.

Point #2 The lock on the *page print object* inherited by action *A* is released, and then acquired by an action nested within action *B*, this action having been delayed waiting for the lock.

Point #3 The lock on the *document print object* inherited by action *B* is released, and then acquired by the action *C*, this action having been delayed waiting for the lock.

Point #4 The lock on the *page print object* acquired by an action nested within action *C* is released, and then acquired by an action nested within action *D*, this action having been delayed waiting for the lock.

Point #5 The lock on the *page print object* inherited by action *D* is released, and then acquired by an action nested within action *C*, this action having been delayed waiting for the lock. (This being the action in action *C* after the action which released the lock.)

6.6 Summary

In this section the concept of coloured actions was introduced, and their properties stated. It was shown how serialising actions, glued actions, top–level independent actions and n–level independent actions can be implemented using coloured actions. Finally two example applications were examined, with respect to their implementation using coloured actions.

7 Implementing coloured actions

The Arjuna system (described in section 3) provides an environment in which reliable distributed object oriented applications can be constructed using atomic actions which operate on (persistent) objects. This section will outline the modifications required to the Arjuna system to support coloured actions. These modification are confined mainly to the state management, concurrency control, and atomic action mechanisms. The stub generator, object store, and the RPC mechanism, remain largely unaffected by these modifications.

7.1 Class hierarchy

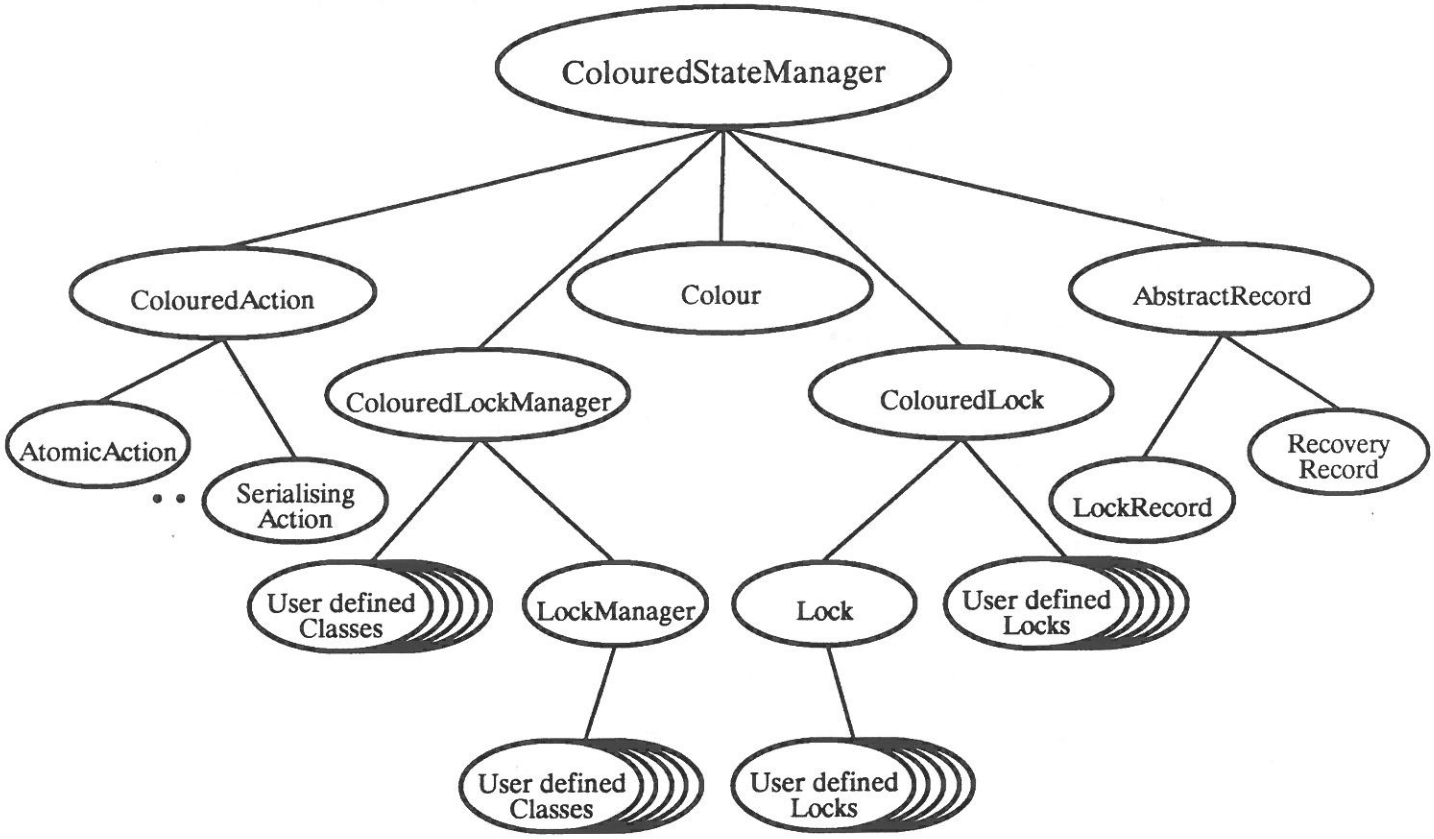
The modifications to the Arjuna system require extending the class hierarchy to provide classes which are suitable for colour based state management and concurrency control, along with a class which provides coloured actions.

The existing Arjuna classes: `AtomicAction`, `LockManager`, `StateManager` and `Lock` would not need extensive modifications to change their functionality to that required by the classes: `ColouredAction`, `ColouredLockManager`, `ColouredStateManager` and `ColouredLock`. From these new classes versions of `AtomicAction`, `LockManager`, `StateManager` and `Lock` could be derived, so maintaining the ability to implement applications using conventional atomic actions.

One new class which is required is the class `Colour`. The `Colour` class will be derived from `ColouredStateManager`, so enabling their states to be made persistent.

Figure 7–1 illustrate the new class hierarchy which would result from the modification outlined above. This can be compared with the class hierarchy for the existing Arjuna system which is illustrated in figure 3–5.

Figure 7-1



In the following sections the major classes in the class hierarchy will be described, along with the modifications required to the existing system, to obtain them.

7.2 Colour class

In keeping with the view that Arjuna is an object oriented system, *colours* will be provided by instances of the class `Colour`. The instances of the class `Colour` will be used by applications to represent colours when beginning coloured actions and when acquiring locks. The class `Colour` will be derived from `ColouredStateManager`; this allows instances of the class `Colour` to be saved in the object store, so allowing an application to save a colour object for future uses, for itself or other applications. A further advantage of providing colours as objects, is that they may be passed as parameters to operations. For example, this would provide the invoker of an operation with ability to specify the colours which are used when obtaining locks in the operation.

Colours are required to be mutually distinguishable. One way in which this can be achieved is for the instances of the class `Colour` to contain a *unique identifier* (`Uid`), and for the class `Colour` to provide an equality operator which compares the unique identifiers of the two colours. The class interface of such a class is given below.

```
class Colour : public ColouredStateManager
{
      Uid Colour_Type;
public:
      Colour();                  // Constructor for temporary colour.
      Colour(Uid* u);            // Constructor for new (persistent) colour.
      Colour(Uid u);             // Constructor for old (persistent) colour.
      ~Colour();

      int operator==(Colour&); // Equality operator.
```

```
        virtual bool save_state(ObjectState* s, object_type t);
        virtual bool restore_state(ObjectState* s, object_type t);
        virtual TypeName type();
};
```

7.3 ColouredLock class

The modifications needed to the existing `Lock` class to provide the functionality required by the `ColouredLock` class are that:

1) The colour of a lock can be specified.
2) The coloured locking rules be used, to determine if two locks conflict.
3) The colour of a lock is saved as a part of the lock's persistent object state.

The colour of a lock could be specified as a parameter to its constructor, along with the mode of the lock. This means that a coloured lock would be created and used in the following way:

```
Colour Red, Green;

res = setlock(new ColouredLock(Red, READ));

res = setlock(new ColouredLock(Green, WRITE));
```

Determining if two locks conflict, as per the coloured locking rules (described in section 6.3.1), can be achieved by the reimplementing the member operation of the `ColouredLock` class which assesses if locks will conflict.

The colour of a coloured lock can be made part of its persistent state by modifying its `save_state()` and `restore_state()` operations.

7.4 ColouredStateManager class

The only significant differences between the original class `StateManager` and the new class `ColouredStateManager` is that the `modified()` operation is parameterised to take the colour of the modification about to be performed. This allows the colour of a modification to be recorded in an instance of the class `ObjectStateRecord`, which is added to the currently running coloured action (if any). An `ObjectStateRecord` object contains information about modifications performed to an object. The purpose of `ObjectStateRecord` objects will be explained when the implementation of the `ColouredAction` class is described.

7.5 ColouredLockManager class

To provide the class `ColouredLockManager`, the existing class `LockManager` should be modified so that when a lock is set on an object, then a instances of the class `LockRecord` is added to the currently running coloured action. This `LockRecord` object contains information about the lock which has been set, including the colour of the lock.

Finally, if a write lock is placed on an object, the `setlock()` operations of the class `ColouredLockManager` should notify an object's `ColourStateManager` of the colour of the modification about to be performed on the object. This is done by calling the `modified()` operation of the object's `ColourStateManager` parameterised with the colour of the write lock.

7.6 ColouredAction class

The implementation of the class `ColouredAction` is required to provide the functionality of coloured actions, the object of the class having operations performed upon them to begin, commit and abort coloured actions.

The original class `AtomicAction` and the class `ColouredAction` differ only slightly, one of these differences being that the `Begin()` operation of the `ColouredActions` class is parameterised to take the set of colours which the coloured action possesses. To determine if a coloured action possesses a particular colour, the `ColouredActions` class would provided a boolean function `HasColour()`.

As mentioned in the previous two sections, the state management and concurrency control mechanisms record with the currently running atomic action information about object state modifications, and locks set on objects, using `ObjectStateRecord` and `LockRecord` objects, respectively (there are other forms of records, used by the Arjuna system, but because these records are not affected by change from atomic actions to coloured action, they are not mentioned here).

The `ObjectStateRecord` and `LockRecord` objects added to a coloured action are processed when the coloured action either commits or aborts.

In the case of the coloured action aborting, processing an `ObjectStateRecord` object will cause the recovery of the object, and the processing the `LockRecord` object will cause lock on the object to be released.

In the case of the coloured action committing, processing an `ObjectStateRecord` object will cause the object to be made persistent if the coloured actions was the top-level (with respect to the colour of the modification), and the processing the `LockRecord` object will cause the lock on the object to be released if the coloured action was top-level (with respect to the colour of the lock). The records which remain after processing will be merged by the committing coloured action into the ancestor of the appropriate colour.

The result of these modification is that the `ColouredAction` object is used as follows:

```
Colour          Red;                // Create a colour
ColouredAction  A;                  // Create an instances of coloured action

A.Begin(Red);                       // Begin the coloured action

      . . .

if (OK)
      Result = B.End();             // End (Commit) the coloured action
else
      Result = B.Abort();           // Abort coloured action
```

7.7 Implementing other forms of actions

From the class `ColouredAction` can be derived classes which provide other forms of actions, such as atomic actions and serialising actions. In the following two sections, it will be explained how classes which provide atomic actions and serialising actions can be derived from the class `ColouredAction`.

7.7.1 Atomic actions

A class which provides atomic actions can be derived from the class `ColouredAction` by simply providing a replacement for the virtual operation which indicates whether a coloured actions possesses a particular colour (`HasColour()`). The replacement operation will always indicate that the colour is possessed by the atomic action. The result of this is that all locks will be retained by the atomic actions, and all changes will be recovered if the atomic action aborts.

7.7.2 Serialising actions

A class which provides serialising actions can be derived form the class `ColouredAction` by simply providing a replacement for the virtual operation which

merges the lists of records when a coloured actions commits. The replacement operation would relock the objects it inherits using the scheme described in section 6.4.1, and then release the original inherited locks.

7.8 Summary

The class hierarchy of the existing Arjuna system can be easily modified to provide the additional functionality of coloured actions, from which it is also possible to provide atomic actions and serialising actions. A trial (non-distributed) version has been implemented to test the feasibility of the proposed approach.

The proposed method of providing atomic actions will allow coloured actions to be used from within an atomic action, without compromising the atomic action's properties of serialisability, failure atomicity, or permanence of effect. The main conclusions drawn from this section is that coloured actions are no more difficult to implement than atomic actions.

8 Examples of large action systems

In this section example applications which can be implemented by large action systems will be studied. The applications which will be studied are: a reliable distributed make, a distributed spreadsheet, and an application which arranges meetings. Each application will be studied with respect to how it would be implemented using atomic actions, then using the new structuring techniques such as serialising actions, common actions, glued actions, and top–level actions, and finally coloured actions will be studied as a means of implementing the new structuring techniques required.

8.1 Reliable distributed make

The following sections describe the requirements for a reliable distributed *make*–like program, then how this application could be constructed using atomic actions, and new structuring techniques, and finality how coloured actions can be used to implement the new structuring techniques required by the reliable distributed *make* application.

8.1.1 Idea behind make

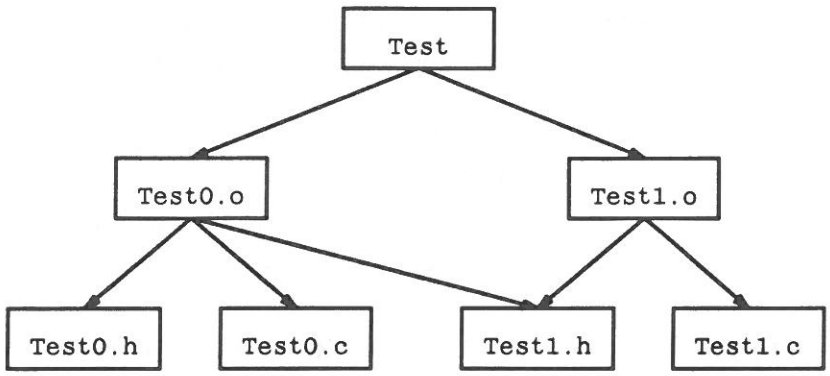
Make [Feldman 79] is a program used to maintain consistency between a set of files. A file is regarded as being consistent if all the files it depends upon are consistent and they were last changed at an earlier point in time than the file itself (a file has a timestamp showing the last time it was modified). The *make* program attempts to make a specified target file consistent in an efficient manner. The files that a target file depends upon are called *prerequisite* files. The procedure of making a target file consistent is recursive in nature, because it may involve making its prerequisite files consistent. The efficiency is obtained by not changing prerequisite files which are already consistent. The user must specify (in a file called "makefile") the dependencies between files, along with commands to be executed to re–establish the consistency of those files which are found to be inconsistent. *Make* is mostly used to maintain the consistency of programs which are constructed from many modules, ensuring that programs are recompiled if definitions in header files are changed.

An example of a *makefile* showing the dependencies between files, and the commands used to re–establish consistency 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 file `Test` depends upon files `Test0.o` and `Test1.o`, where `Test0.o` depends upon `Test0.h`, `Test1.h` and `Test0.c`, and `Test1.o` depends upon `Test1.h` and `Test1.c`. If the *make* program was run with the above *makefile*, and the files `Test0.o` and `Test1.o` are consistent, but had been more recently changed than `Test`, then the command "`cc -o Test Test0.o Test1.o`" would be executed to re–establish the consistency of the file `Test`. Below in figure 8–1 is a network showing the dependencies in the above example, where the boxes are files and the arrows represent dependencies, and the arrows point from a target file to its prerequisite files.

Figure 8–1



8.1.2 Requirements of a reliable distributed make program

A reliable distributed *make* program should be constructed so that it can take advantage of the high degree of possible concurrency available in distributed systems, so allowing many files to be made consistent simultaneously. The user should be able to provide alternative methods of obtaining consistency, so if a sequence of commands fail to re–establish consistency, a different approach may be taken to re–establish consistency.

If an attempt to make a target file consistent fails, it is desirable that the prerequisite files which were made consistent to obtain this goal should remain consistent, despite the failure to make the target consistent. It is also desirable that the failure to make a prerequisite file consistent does not stop other prerequisite files from being made consistent.

It will be assumed that there are no cyclic dependencies between files.

8.1.3 Implementation using atomic actions

To design a reliable distributed *make* program which is to be implemented using atomic actions, it is necessary to decide: how to split the operations of *make* into atomic actions, the order for executing these atomic actions, and to decide what should be done if an atomic action fails. It will also be necessary to decide the way in which failures are treated amongst concurrent atomic actions.

The process of making a target file consistent splits naturally into four phases: ensuring the consistency of the prerequisite files, obtaining the last changed times of prerequisite files, obtaining the target file's last changed time and the execution of the commands to re–establish consistency of the target file (if necessary).

8.1.3.1 Enclosing top–level atomic action

These four phases must be enclosed in a top–level atomic action, to ensure that prerequisite files are not altered (so becoming inconsistent), between the time they were made consistent and the time that they are used to make the target consistent.

The top–level atomic action does not acquire locks, but just retains them when they are released by nested atomic actions.

If any of the phases within the top–level atomic action fail, the top–level atomic action should not abort, because this would cause the recovery of any changes made to files which have been made consistent. So if any of the phases fail to be completed successfully then the top–level atomic action should not attempt any of the remaining phases, and simply commit the effects already accomplished.

8.1.3.2 Phase 1 (Ensuring the consistency of the prerequisite files)

Ensuring of the consistency of a prerequisite file involves the same atomic action structure as required for ensuring the consistency of target file, except that no enclosing atomic action is required.

Each prerequisite file can be ensured to be consistent concurrently, thus exploiting the available concurrency in distributed systems.

This phase could be expected to obtain read locks on files which are already consistent, and write locks on files which are required to be made consistent (creating the files if they do not exist).

8.1.3.3 Phase 2 (Obtaining the prerequisite files' last changed times)

The atomic actions which obtain the prerequisite files' last changed times can be performed immediately after the consistency of each prerequisite file has been established. The atomic actions will require a read lock on prerequisite files that they are inspecting.

If an atomic action fails while attempting to obtain the last changed time for a file, one way in which such a failure could be tolerated would be to take the last changed time to be the present system time; this will cause the commands to re-establish consistency to be executed even if the target file is consistent, so enabling progress.

8.1.3.4 Phase 3 (Obtaining the target file's last changed time)

The obtaining of the target file's last changed time can be performed by an atomic action. This atomic action will require a read lock on the target file.

If the atomic action fails, it is desirable that the last changed time of the target be taken as some old time, such that it will cause the commands to re-establish consistency to be executed (even if the target file is consistent), so enabling progress.

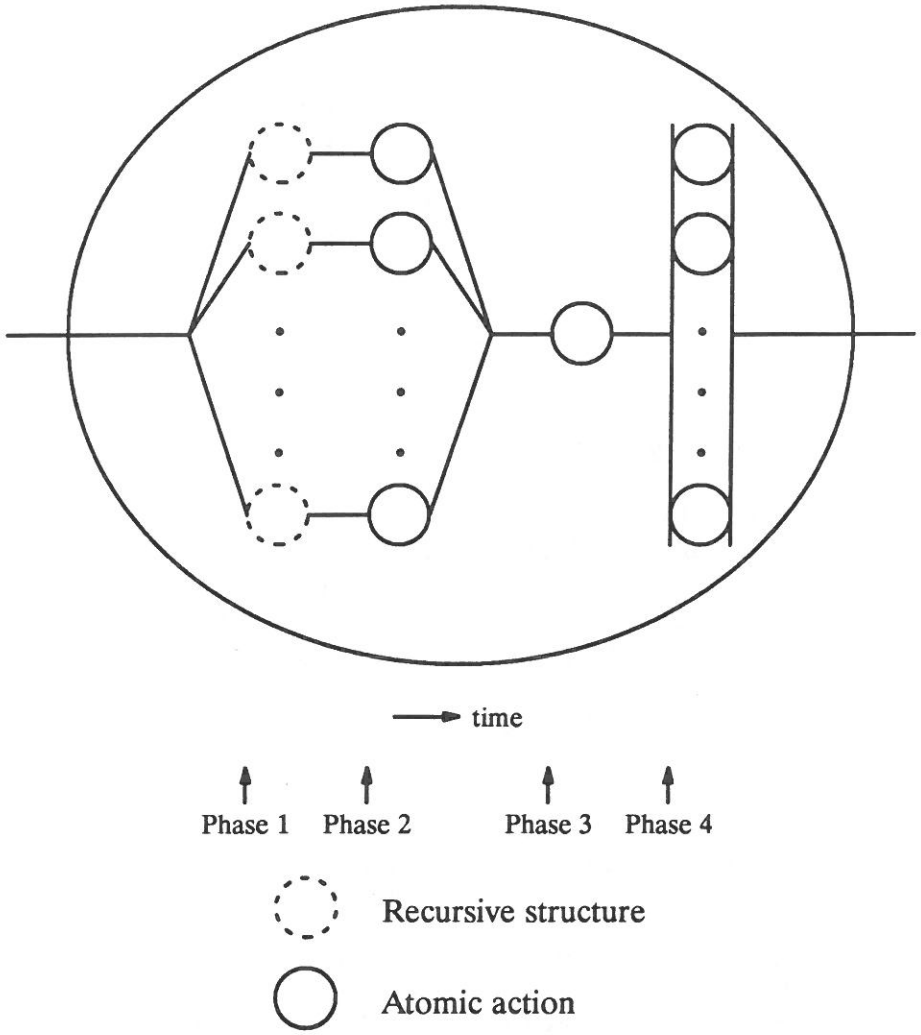
8.1.3.5 Phase 4 (Execution the commands to re-establish consistency)

The execution of the commands to re-establish consistency is structured as a set of alternative atomic actions each of which could obtain consistency of the target file. The alternatives will be tried in turn until one is succeeds. If an alternative fails, its effects will be recovered, so permitting the next alternative to be tried.

Each alternative atomic action will require a write lock on the target file along with read locks on each of the prerequisite files.

The diagram in figure 8–2 illustrates the system of atomic action which perform these four phases along with the enclosing atomic action.

Figure 8–2



8.1.3.6 Observations about implementation

It is interesting to note that though the above design is recursive in nature, is does not produce deeply nested atomic actions. The maximum depth the nesting of atomic actions ever reaches is two levels.

The creation of concurrent atomic actions occurs dynamically. The number of concurrent atomic actions created depends on the dependencies between files, and the number of files involved.

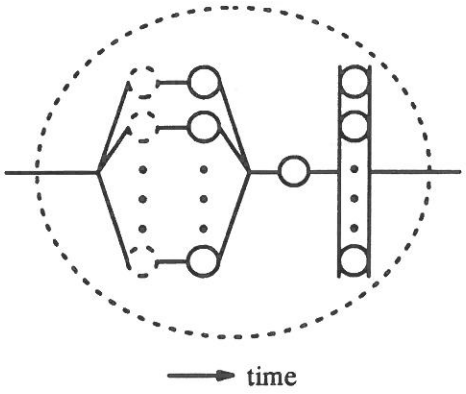
The use of an enclosing top–level atomic action is not totally satisfactory because all of the effects are only made permanent when it commits. So if a machine which is the home of one of the prerequisite file fails, this would mean that none of the effects produced could be made permanent.

8.1.4 Implementation using new structuring techniques

The major problem with the implementation described in the previous section is that the top–level atomic action does make any effect permanent until it commits. What is required is the use of a serialising action since this would allow the effects of atomic actions nested within the serialising action to be made permanent when the nested atomic actions commit, at the same time providing the serialising property needed for the system.

The resulting structure is illustrated in figure 8–3, the enclosing atomic action having been replaced by a serialising action.

Figure 8–3



8.1.5 Using coloured actions

The problem involved with implementing a *make* program using atomic actions can also be overcome by the use of coloured actions. The refinement using serialising actions described in the previous section can also be accomplished using the coloured actions. This is because (as detailed in a previous section) coloured actions can be structured in such a way that an enclosing coloured action has the characteristics of a serialising action.

8.2 Distributed spreadsheets

Typical spreadsheets [McBride 89] [Bookbinder 89] are programs which allow a user to construct a system of arithmetic calculations, and display the results for a given input. The input may be changed interactively, and the spreadsheet will then perform the calculations and display the corresponding results.

In the following sections, methods of providing a reliable and distributed form of a spreadsheet using different forms of actions will be investigated. The spreadsheet is intended to be distributed, in that the state which makes up the spreadsheet will be distributed amongst many machines, and that the spreadsheet should be accessible from many machines. It is also intended that parts of the spreadsheet will be shared, allowing different users to alter parts of the spreadsheet simultaneously.

The objective is to investigate the ways in which an object oriented spreadsheet system [Piersol 86] could be constructed using actions in which the consistency of the spreadsheet can be maintained in the presence of users performing simultaneous changes to the spreadsheet and failures of machines on which parts of the state which make up the spreadsheet are held. It is intended to construct the spreadsheet system so that it can take full advantage of the concurrency available in distributed systems.

8.2.1 Conventional spreadsheets

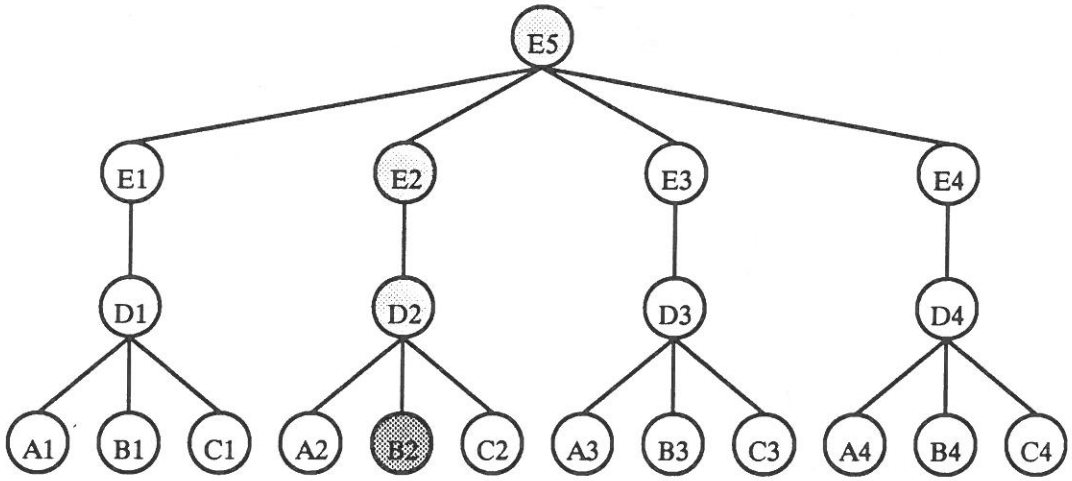
An example of a conventional spreadsheet is given in figure 8–4. The spreadsheet comprises of row and columns, the columns being labelled *A* to *E*, and the rows labelled *1* to *5*. The rows and columns are made up of elements, the name of the element being derived from the names of the row and column it is on, for example the element which is on row *4* and column *D* is given the name *D4*. In the example below the system of calculations associated with the spreadsheet has been defined to give the following behaviour. The elements of a row i, $1 <= i <= 4$, in columns *A*, *B* and *C* are added and the result for that row is placed in column *D*. The entries in column *D* are divided by three, and the result is put into column *E*. The result of averaging the values in column *E* from row *1* to *4* is then placed in *E5*.

Figure 8–4

	A	B	C	D	E
1	42	40	25	107	35.67
2	31	45	65	141	47.00
3	33	43	34	110	36.67
4	30	50	58	138	46.00
5					41.33

It is clear that a spreadsheet contains many dependencies between the elements it contains. The dependencies can be used to find the elements which need to be updated if an element is changed. The dependencies for the above example are given in the figure 8–5. (The dependency graph of the elements of a spreadsheet does not have to be a tree as in the example below.)

Figure 8–5



In figure 8–5 if the element labelled *B2* were changed, the elements labelled *D2*, *E2* and *E5* would need to be updated.

8.2.2 Spreadsheet application

In this section the part of the spreadsheet application which is to be studied will be described. This will involve specifying the method of representing the spreadsheet, describing sets of possible consistency constraints which are suitable for that representation method, and specifying the operation which is to be implemented using a system of actions.

8.2.2.1 Spreadsheet representation

A spreadsheet can be regarded as being constructed from elements, each element being a formula. The formula may depend on the values of other elements' formulae, for example the element *A7* may have a formula *A4+A5+2*.

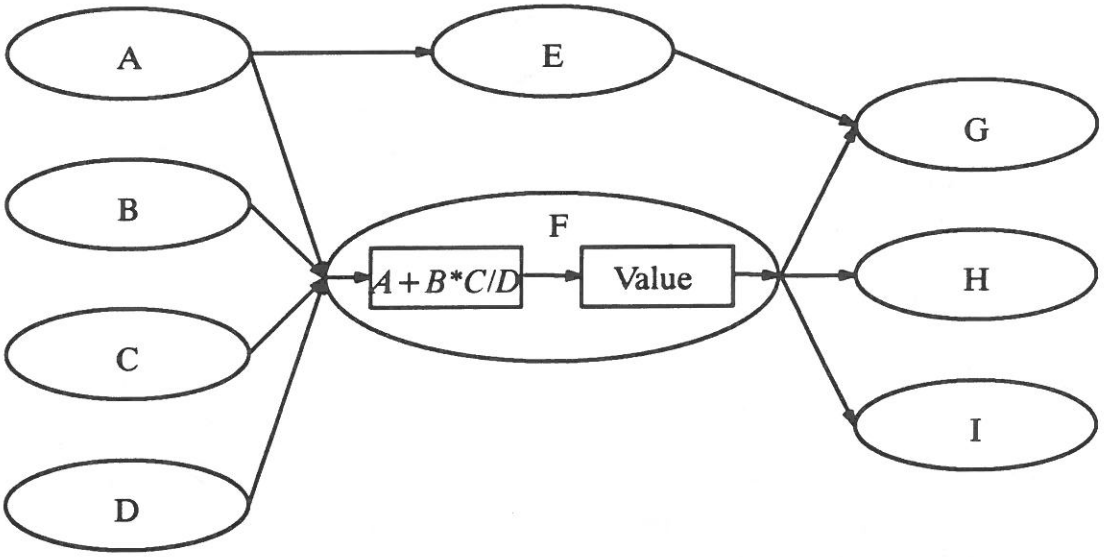
If we assume that recalculating the value of the formula is a time consuming operation, then the representation of a spreadsheet as a collection of elements containing formulae would be very inefficient, because calculating the value of a single element for display may cause many formulae to be recalculated even if their values have not changed. This method of representing a spreadsheet is also inappropriate because if a formula which makes up part of the spreadsheet is not available due to the machine on which it resides

having crashed, then all the parts of the spreadsheet which depend upon the value of that formula cannot be recalculated, until such time as the machine starts working.

What is needed is for each of the elements in the spreadsheet to contain a *value*, a set *of dependent elements* and the *formula*. The value contained in the element is the value of the formula, and is a means of *caching* the result of the formula, preventing the formula from having to be recalculated each time the value of the element is required. The set of dependent elements is required so that the elements which use the value of this element can be forced to be recalculated if the value of the element changes. The set contains only those elements which directly use the element's value. It will be assumed that the set of elements which an element depends upon can be found by examining it's formula. For example, in figure 8–4 the element *D1* has the formula *A1* + *B1* + *C1*, so element *D1* depends upon elements *A1*, *B1* and *C1*. This approach to representing a spreadsheet will be assumed in the following sections.

Figure 8–6 illustrates how elements are related in a spreadsheet, and the relationship between an element's *formula* and *value*, the formula of element *F* being of the form *A* + *B***C*/*D*, and value of *F* formula is required by the formulae of the *G*, *H* and *I* elements.

Figure 8–6



8.2.2.2 Consistency constraints on a spreadsheet

When deciding how to implement a distributed spreadsheet system, it is important to specify what the consistency constraints of the spreadsheet are. The consistency which will be considered here is that between the element's value, its set of dependent elements and its formula. The consistency constraints of the spreadsheet will indicate which operations can be performed concurrently, and what recovery mechanisms are required.

Outlined below are three of the possible approaches which could be taken for defining the consistency constraints of a distributed spreadsheet system:

A) To a user it appears that for any element, *F*, the value of *F* is that of the element's formula, and the set of elements which depend on *F* is also consistent with their respective formulae (all and only the elements which use the value of the element *F* are in *F*'s set of dependent elements. For example, in figure 8–6, this set of elements for *F* will include elements *G*, *H* and *I*).

B) To a user it appears that an element's value is either that of the element's formula or marked as being invalid, and the set of dependent elements for an element is consistent with respect to all element's formulae. It is also necessary that an invalid value will be made valid eventually.

C) It appears to the user that an element's value is either that of the element's formula or some old value of the element's formula, and the set of dependent elements for an element is consistent with respect to all element's formulae. It is also necessary that the value of an element will be made to correspond with the formula eventually.

The consistency constraints outlined above range from the strong consistency constraint of *A* to the weak consistency constraint of *C*. (As far as the user is concerned, consistency constraint *C* suffers from the problem that the user cannot tell if an element's value is consistent with its formula.)

8.2.2.3 Operation of study

The operation on the spreadsheet which will be examined with regard to the consistency constraints proposed above is that of changing of an element's formula. The action system which implements this operation will be required to perform the following tasks:

1) The removal of the element (whose formula is being changed) from the set of dependent elements of each element used in the old formula.

2) The changing of the formula.

3) The addition of the element whose formula has been changed to the set of dependent elements of each element used in the new formula.

4) The changing of the value of the element.

5) The changing of the value of the elements which depend on the element whose formula has been changed.

For example, if the formula of element *D1* in figure 8–4 were changed from *A1* + *B1* + *C1* to *A1* + *B2* + *C3*, the five tasks would be:

1) The removal of *D1* from *A1's*, *B1's* and *C1's* sets of dependent elements.

2) Change the formula from *A1* + *B1* + *C1* to *A1* + *B2* + *C3*.

3) The addition of *D1* to *A1's*, *B2's* and *C3's* sets of dependent elements.

4) The recalculation of the value of *D1*.

5) The recalculation of the values of *E1* and *E5*.

8.2.3 Implementation using atomic actions

In this section the implemention of the operation described above using atomic actions will be examined for each of the consistency constraints specified.

8.2.3.1 Implementation for consistency constraint A

The consistency constraint *A* is such, that it is necessary to perform all of the tasks of the operation within a single top–level atomic action. The internal structure of this atomic action will be described in the rest of this section.

Within the top–level atomic action, the first task can be performed by a set of concurrent nested actions, one for each of the set on which the element appears. If any of the nested atomic actions fail, then the task will not be completed satisfactorily, so the top–level atomic action should be aborted.

The next task within the top–level atomic action is the changing of the formula; this can be performed by a single nested atomic action (internally this action could contain nested atomic actions). If this nested atomic actions fails, the formula will not have been changed, so the top–level atomic action should be aborted, causing the recovery of the effect of this task, and the previous task.

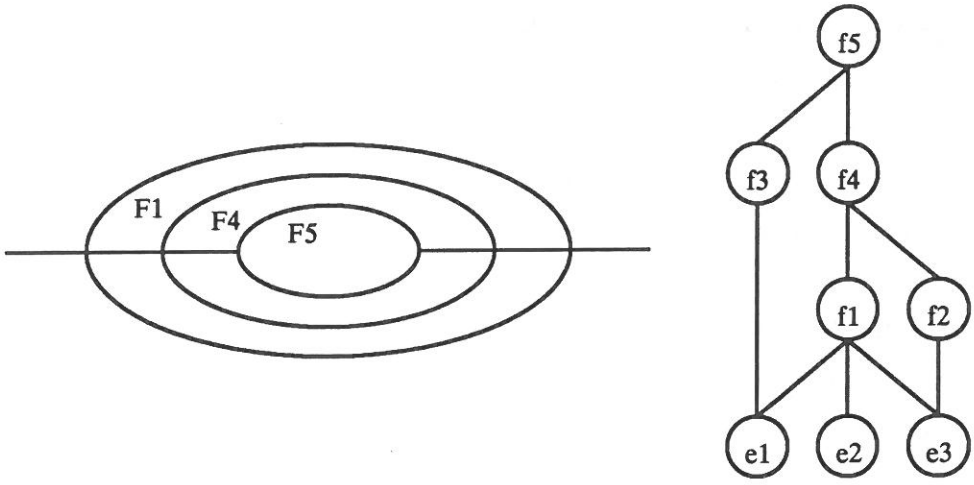
After the formula has been changed, the next task is the addition of the element whose formula has been changed to the set of dependent elements for each of the elements used in the new formula. This can be performed by a set of concurrent nested actions, one for each of the set on which the element appears. If any of the nested atomic actions fail then the task will not have been completed, so the top–level atomic action should be aborted.

The task of calculating the value of the element can be performed by a single nested atomic action. If this nested atomic actions fails, then the value of the element will not correspond to the element's formula, so the top–level atomic action should be aborted, causing the recovery of the effect of this task, and the previous tasks.

The final task needed to be performed is the recalculation of the values of the elements which depend on the value of the element whose formula has been changed. For a large spreadsheet this could be a very time consuming task, so it is important that this task is performed with the aim of exploiting as much concurrency as possible.

One approach is to create a concurrent nested atomic action for each of the elements which directly depend upon the element. Within each concurrent nested atomic action the new value of that element is recalculated, and further concurrent nested atomic actions are created within the atomic action for each of the elements which depend directly upon it. Figure 8–7 shows such an arrangement of atomic actions when the formula for *e2* has been changed, thereby requiring the recalculation of the values of the elements which depend on *e2* (*f1*, *f4* and *f5*). To recalculate the value of the elements which depend directly on *e2*, atomic action *F1* is invoked. *F1* recalculates the value of the elements *f1* and creates a further nested atomic actions, *F4*, to recalculate the value of *f4*. This process continues recursively until all elements which depend upon *e2* are recalculated.

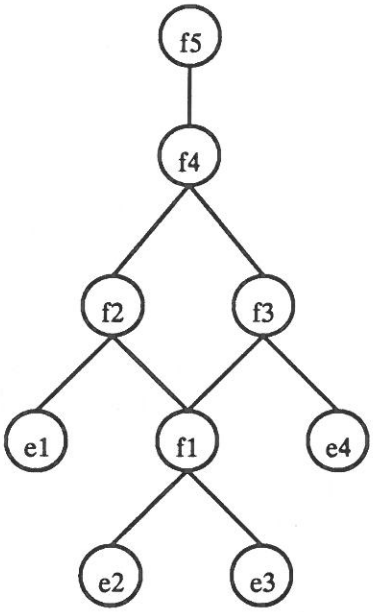
Figure 8–7



There is a problem with this approach in that lock conflicts can occur between nested atomic actions which are recalculating the values of elements. Such lock conflicts between concurrent atomic actions of a task cannot be distinguished from deadlocks with nested atomic actions in other top–level atomic actions. To cope with the possibility of a deadlock, a mechanism which progressively aborts and restarts each level of in the atomic action hierarchy could be used. If such a mechanism is used and the lock conflict is just due to another concurrent nested atomic action in the same task, such a mechanism could cause large amounts of work to be aborted.

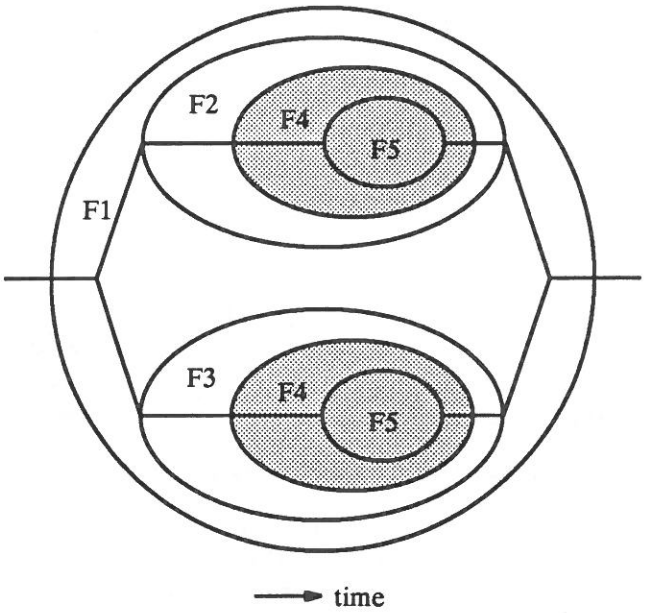
For example consider the dependency graph given in figure 8–8.

Figure 8–8



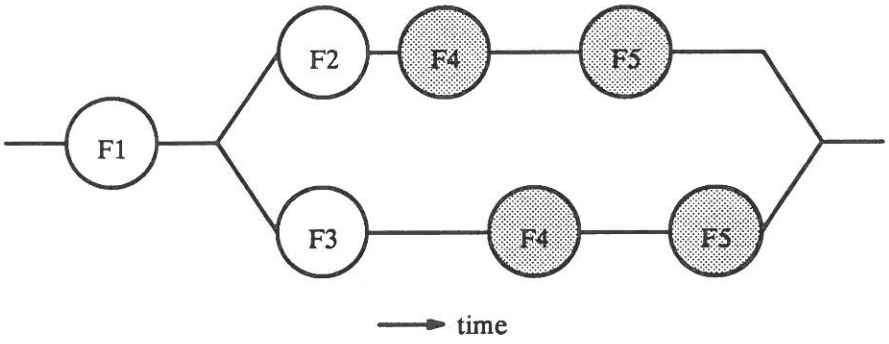
The structure in figure 8–9 shows the atomic actions which make up the recalculation task, which is produced when the formula of element *e2* is changed. Note that there are two nested atomic actions which are attempting to recalculate values of the element *f4* (atomic action *F4*) and *f5* (atomic action *F5*); this could cause lock conflicts between these two atomic actions.

Figure 8–9



Given the problem with the approach outlined above another method of accomplishing the task is desirable. An alternative is to reduce the degree of nesting, as shown in figure 8–10. This structure corresponds to the recalculation task associated with the changing of the formula of element *e2* in the dependency graph given in figure 8–8.

Figure 8–10

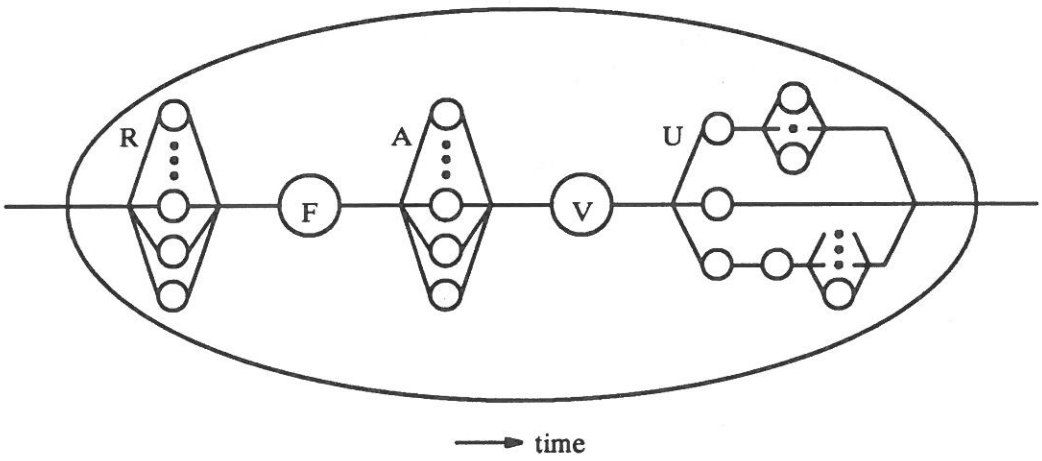


Use of this approach means that locks on elements are held by nested atomic actions for shorter periods of time, so reducing the chances of lock conflicts.

Given the method outlined above for performing the task, if any of these nested atomic actions fail, not all the values of the elements of the spreadsheet will be correct, so the top–level atomic action should be aborted.

The internal structure of the entire top–level atomic action is illustrated in figure 8–11, where the structures *R*, *F*, *A*, *V* and *U* represent tasks 1 through to 5 respectively.

Figure 8–11



The approach of performing the whole operation as a single top–level atomic action suffers from the disadvantage that, the long period the operation may take would mean that locks obtained on elements could give rise to large amounts of lock conflict, especially if many users are attempting to alter elements in a large spreadsheet simultaneously.

8.2.3.2 Implementation for consistency constraint B

A system which implements the operation using consistency constraint *B*, could be implemented using the same approach as in the system which implemented consistency constraint *A*, but this would have the disadvantage mentioned above. Furthermore such an implementation would not take advantage of the allowance for elements to be marked as invalid.

An approach more suited to implementing the operation, and which obeys the consistency constraint *B*, is atomically to perform tasks 1 to 3 and in addition the marking as invalid of the value of the element being changed, and all the elements which depend upon the element which is being changed.

To perform these tasks atomically it would be natural to enclose them within an atomic action. The internal structure of this atomic action would be identical to the system discussed in the previous section, except that instead of the last two tasks recalculating the

values of elements, they simply mark them as invalid. If any of the component tasks within the atomic action fails to be completed properly then the enclosing atomic action is aborted.

Once this first phase has been successfully completed, all the elements which have been marked as invalid must be recalculated. This need not be performed atomically, so the recalculation of the values of individual elements can be performed by top–level atomic actions. The structure of these atomic actions is the same as the structure of the nested atomic action which marked the elements values as invalid, but performed as top–level atomic actions.

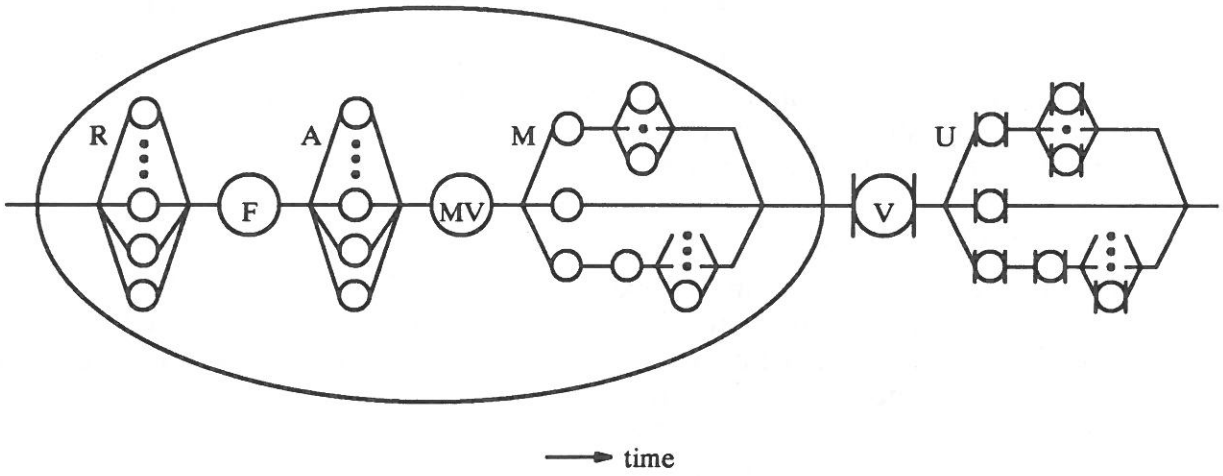
Each of the top–level atomic action which is recalculating the values of elements must eventually succeed. If the atomic action initially fails then it should be retried until it is successful; this must be done to ensure that all the element whose values have been marked as invalid are eventually recalculated.

This approach to implementing the operation has many advantages over the one discussed in the previous section. It is reasonable to assume that the marking of an element as invalid will take considerably less time than recalculating the element's value, so this will mean that the first phase will complete considerably quicker, thereby reducing the interval in which lock conflicts can occur. The second phase being performed by top–level atomic actions will in effect allow the recalculation tasks of different users to cooperate in the recalculation, instead of competing for locks on elements.

One disadvantage of this approach is the additional overhead of the marking of elements as invalid, which is not required in the approach detailed in the previous section.

The full system of atomic actions needed to perform an operation on the distributed spreadsheet is illustrated figure 8–12, where the structures *R*, *F*, *A*, *V* and *U* represent tasks 1 through to 5 respectively, and where the nested atomic action *MV* is the marking as invalid of the elements value, the structure *M* is the marking as invalid of the elements which depend on the element which is being changed.

Figure 8–12



8.2.3.3 Implementation for consistency constraint C

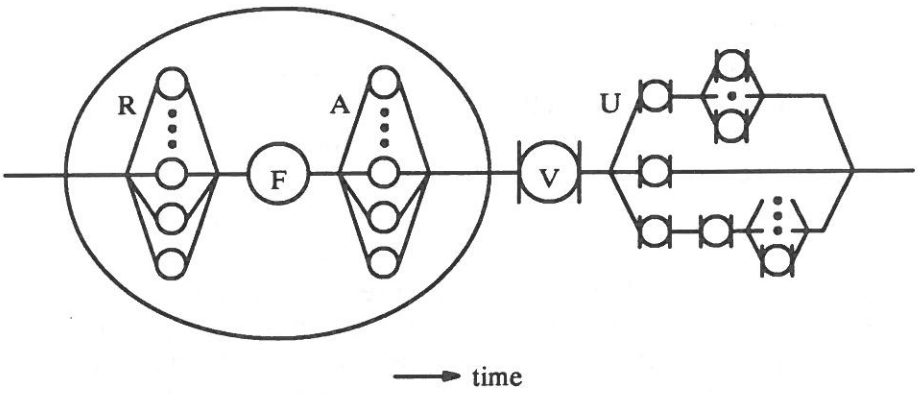
The weakness of consistency constraint *C* means that the value of an element need not reflect the present value of that element's formula. The only restriction is that eventually the value of an element should reflect the present value of that element's formula.

This means that the system of atomic actions which implements the operation and also obeys consistency constraint *C* can be structured like the system described in the previous sections, except that the part of the first phase in which the values of the elements are marked as invalid, is not required.

This means that the system contains two phases, the first phase is the changing of the sets of dependent elements and the element's formula (this must appear atomic), and the recalculating of the values of out of date elements.

The structure of the atomic action system which implements the operation is illustrated in figure 8–13, where the structures *R*, *F*, *A*, *V* and *U* represent tasks 1 through to 5 respectively.

Figure 8–13



The great advantage of this method is that only the elements which appear in either the old or new formula, need to be available while the first phase is being done. Whereas in the previous two sections, not only where these elements required to be available, but in addition every element which depended upon the changed element. This means that this method will be less affected by the unavailablity of elements due to the failure of machines. But this method does have the disadvantage that a user will not be able to tell if the value of an element reflects that of its formula. This could be a serious problem in large spreadsheets where the time taken to recalculate the values of all the elements affected by a change could become unacceptably long.

8.2.4 Implementation using the new structuring techniques

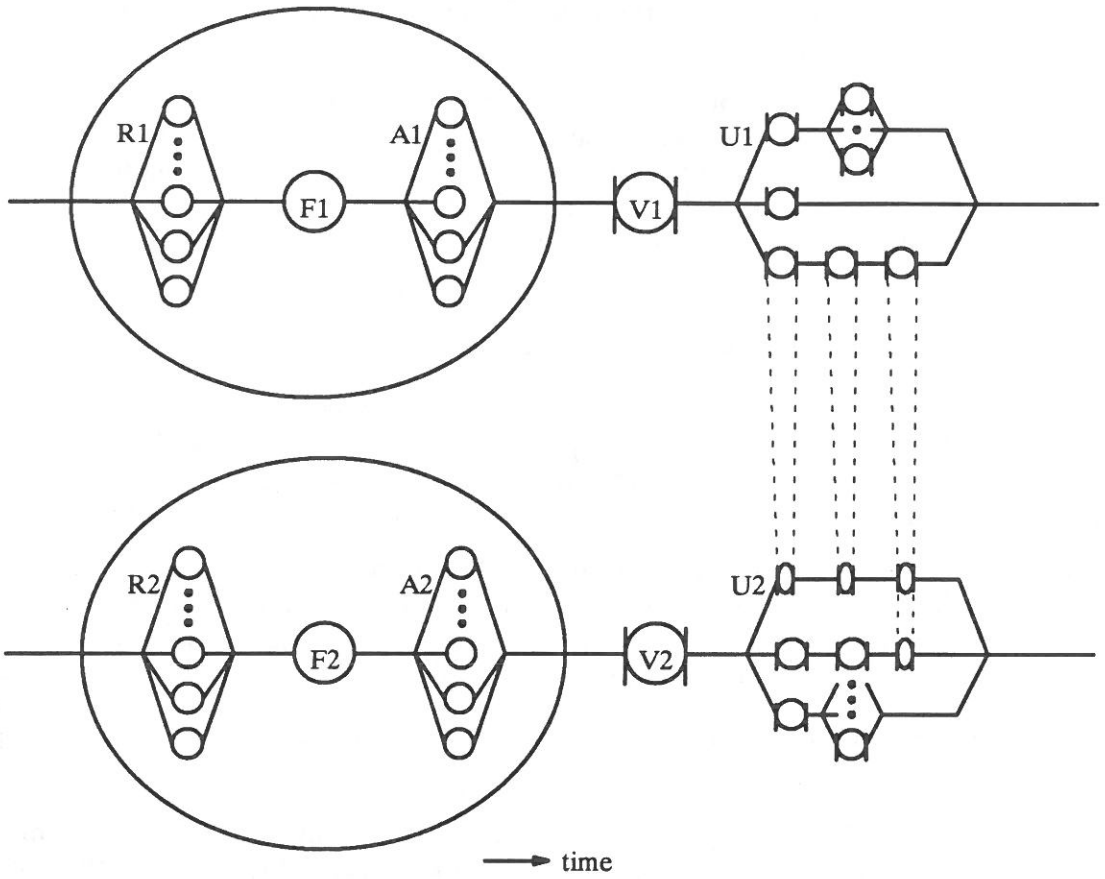
The structuring technique which is most useful for the implementation of the operation studied is the common action. Common actions can be used to enable top–level atomic actions to share the invocations of nested atomic actions, if the operation performed by the nested atomic action is idempotent. Common actions also allow different applications to share in the invocation of a top–level atomic action, if the operation performed by atomic actions is idempotent.

Many of the sub–tasks within the operation of changing the element's formula are idempotent (if implemented appropriately). For example, the following sub–tasks are idempotent: marking an element's value as invalid, and recalculating of the value of an element's formula (due to the locks required by this operation, it can only be performed as a common action, at the top–level of nesting).

The use of common actions for these sub–tasks will reduce the amount of lock conflict between users simultaneously attempting to change different elements, and in addition reduce the amount of duplicated work. It is also likely that such common actions would be shared within the same task.

An example of the atomic action system which uses common actions, is illustrated in figure 8–14. The tasks *U1* and *U2* are recalculating the values of elements, for two different user operations, and some of the elements which require recalculation are common to both operations, so the top–level common actions which perform the sub–task are shared.

Figure 8–14



8.2.5 The use of coloured actions

The structuring technique shown to be useful in the previous sections to aid in the implemention of the distributed spreadsheet application was common actions. At present, no totally general method is known by which common actions can be implemented using coloured actions. But coloured actions allow the implementor of the spreadsheet to acquire many locks on objects, but to only retain the locks which are strictly necessary. For example, all of the implementations described in the previous sections require an atomic action which adds an element to a set of dependent elements, and the implementor must be careful that the atomic action does not prevent other atomic actions from adding elements to the set. For example if the set was implemented by an ordered linked list the atomic action which added an element would acquire read locks on the linked list up to the insertion point, and if retained these lock would prevent an element being inserted into this portion of the linked list. The use of coloured actions would enable those locks on the linked list to be totally released when the action committed. This could be achieved by acquiring the locks using a different colour.

8.2.5.1 The spreadsheet graphical representation

A problem not discussed before is the relationship between the graphical representation of the spreadsheet and the elements it contains. If the graphical representation is just another object in the system which can be manipulated using atomic actions, then it is not unreasonable that the user of the spreadsheet would require that the graphical representation be read locked, so preventing it being changed while the spreadsheet was being used. The problem is that the top–level atomic action within which the lock on the graphical representation of the spreadsheet is obtained, must encompass the whole use of the spreadsheet, so preventing other users form manipulating the elements of the spreadsheet. If coloured actions are used than the coloured action which

encompasses the use of the spreadsheet may be a different colour to the coloured actions used to manipulate the elements of the spreadsheet, so allowing the other users to change elements of the spreadsheet.

One approach to solving this problem, which resulted from work in the database area, is the use of *atomic data sets* and *setwise serialisability* [Sha 85] [Sha 88]. The placing of the objects which contain data on the graphical representation of the spreadsheet, and the elements of the spreadsheet, in different atomic data sets, means that their consistency is independent of each other, so allowing the operations on these sets to be serialised using setwise serialisability.

8.3 Arranging a meeting

In this section an application which arranges meetings between a group of users will be discussed. This application will be specified, then examined with respect to its implementation using atomic actions, the new structuring techniques, and how the required new structuring techniques can be implemented using coloured actions.

The application of arranging a meeting is different from the applications studied in the previous sections in that the execution of the application is intended to take days, or even weeks. The application is not active during all of this period, but will consist of short periods of activity separated by long periods of idleness.

8.3.1 Specification of the application which arranges meetings

The purpose of the application is to arrange the date of a meeting between a group of people, and for the date to be acceptable to all the people involved in the meeting. It will be assumed that each user has access to a calendar object which records the dates and times of meeting. The application will initially receive from each member of the group a set of preferred dates for the meeting. Once all this information has been gathered it will be analysed to find which dates are popular for the meeting, and which dates are impossible or unpopular. A list of popular dates for the meeting will then be distributed to the group of people involved in the meeting, who then provide an additional set of preferred dates. This process is repeated until a date is found which is acceptable to all the members of the group. The date of the meeting will then be registered, and notified to the people taking part in the meeting (it will be assumed that once the date and time of a meeting is registered, it can only be changed by executing a separate "cancel meeting" application function not discussed here).

The application will have to be designed to cope with the possibility of many meetings being arranged simultaneously, and this will require some form of control to prevent two instances of the application from attempting to register a person as having a meeting at the same time. The control between the applications will be obtained by the setting of locks on, and writing information to, entries of a user's calendar object. The calendar objects are not solely used by this application, and may contain information placed there by other types of applications. It cannot be assumed that other types of applications will not place information in empty entries of calendar objects, unless the entries are locked.

To prevent problems of contention over dates, each application will be required to have a set of popular dates which is disjoint with respect to the sets belonging to other instances of the applications, if the instances of the application have a common person in their respective meetings.

It will be assumed that actions cannot be guaranteed to be executed at specific times, and that an empty entry in a user's calendar does not imply that date is available, though it will be assumed that if an entry contains information, then that date is unavailable.

8.3.2 Implementation using atomic actions

The implementation of this application using atomic actions clearly cannot be performed within a single enclosing top–level atomic actions, due to the nature of the interactions required between the users and the application.

Because the application cannot be enclosed within a top–level atomic action locks on entries cannot be retained between phases of the selection process, so it cannot be guaranteed that the entries will not be written into by other applications. This means that the set of popular dates chosen by one application could get used by other applications. This application as specified in the previous section does not appear to be suitable for implementation using atomic actions.

8.3.3 Implementation using the new structuring techniques

The structuring technique which would provide the right functionally required by the application is the use of glued actions. Glued actions allow selected locks to be transferred between an atomic action and the atomic action which immediately follows it, without the possibility of other atomic action being able to acquire the locks.

The application will be constructed from glued top–level atomic actions, and each top–level atomic action may be separated from the next by a long period of inactivity. A given top–level atomic action will perform one iteration in the process of finding a date for the meeting which is acceptable to all the people taking part.

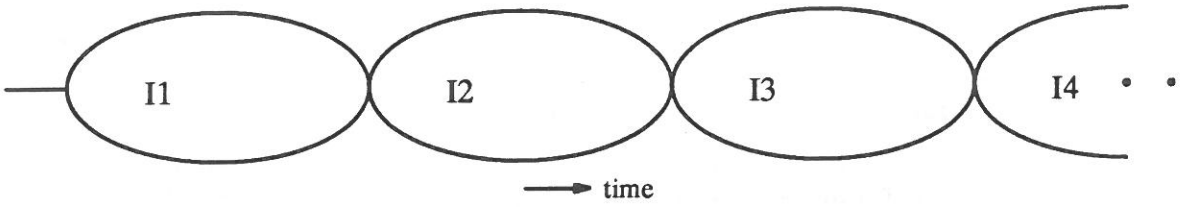
Each iteration will involve: acquiring from each user their initial/additional set of preferred dates, determining which dates are impossible due to any of the participants having prior commitments, choice of a new set of popular dates, and informing the people involved in the meeting of the present popular dates. If there are any dates in the set of popular dates which are acceptable to all the people participating in the meeting, the registration of one of these dates is the respective calendar objects will be performed.

Because each iteration will involve a top–level atomic action, and to ensure that entries in calendars which correspond to popular dates are not used, they must be locked (so preventing other types of applications from using the entry), and these locks must be held for the whole period in which the entries are required to be used (Which may be longer than the top–level atomic action). But each individual iteration which makes up the application will be required to acquire information from the people involved in the meeting and return information back about the present set of popular dates.

Glued actions are useful for this application because the locks on entries in calendar objects which correspond to the present set of popular dates can be transferred between the top–level atomic actions which perform the iterations, while still allowing each iteration's effects to be made permanent once it is finished, thereby allowing the required interaction with the people involved in the meeting. Also, the locks on entries in calendar objects which are regarded as being unpopular can be released at the end of the iteration, so allowing other instances of the application to use the corresponding dates to arrange other meetings.

The structure of the resulting system of actions is illustrated figure 8–15. Although the top–level atomic actions which perform the iterations (*I1*, *I2*, *I3*, *I4*) are illustrated as following each other instantaneously, in reality this need not be so.

Figure 8–15

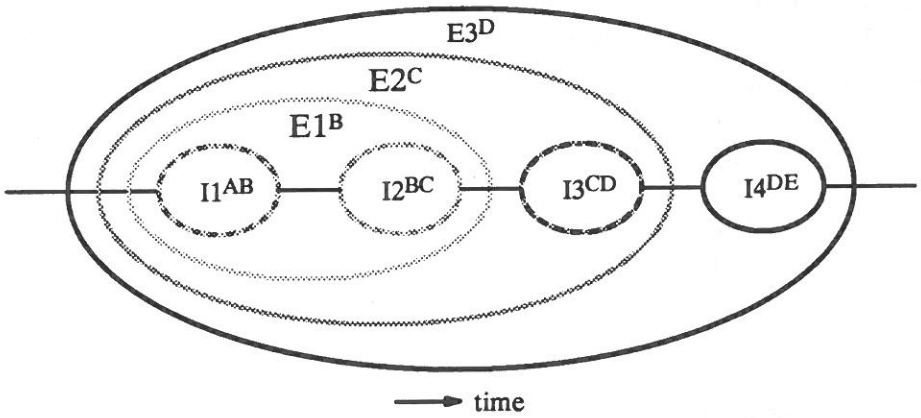


8.3.4 The new structuring techniques using coloured actions

As we have seen before, structuring techniques of a series of glued top–level atomic actions can be implemented easily using coloured actions. In this section how this can be done will be explained, and the problems examined.

To achieve the effect of a series of glued top–level atomic actions colours have to be assigned to the actions of the iteration. Such a scheme is illustrated in figure 8–16, where the system contains four iteration stages *I1*, *I2*, *I3*, and *I4*, which requires three enclosing top–level coloured actions *E1*, *E2* and *E3*. The iterations *I2* is coloured ^{B} and ^{C}, and is enclosed in the coloured actions *E1*, *E2* and *E3*, the coloured action *E1* will finish before the next iteration (*I3*), but the coloured action *E2* encloses *I2* and *I3*.

Figure 8–16



A problem with this approach is that the number iterations which can be performed is limited by the initial number of enclosing coloured actions created.

8.4 Summary

This section describes the design of three example applications. Each application is examined with regard to its implementation using atomic actions, then with regard to how the new structuring techniques could be employed. The three applications which are described are: a distributed make program, a distributed spreadsheet, and an application which arranges meetings.

The distributed make program illustrates the problems which arise when partial success is acceptable from an application. The spreadsheet applications shows how the structure of an application can change, depending on the consistency constraints that the application is required to maintain. The application which arranges meetings illustrates some of the problems which can arise when using long running atomic actions.

9 Conclusions

This section will summarise the material covered in this thesis, and give an indication of the possible areas of future research which are related to the work covered by this thesis.

9.1 Thesis summary

The initial part of this thesis presents the background to the construction of reliable distributed application using actions and objects, describing: fault tolerance, distributed systems, and object oriented programming.

In the next section of this thesis, techniques for the construction of reliable distributed application are discussed. This section covers the use of atomic actions for constructing reliable distributed applications, and some of the refinements to the methodology of using atomic actions for the construction of application.

The section covering the Arjuna system contains a brief description of its structure, then proceeds by describing the various mechanisms provided, along with how they can be used for constructing reliable distributed applications.

In the next section the development of a simple spreadsheet application using the Arjuna system is described, the spreadsheet being constructed as a class, providing a number of operations. Four possible implementations of the spreadsheet class providing differing functionalities are outlined: providing persistent objects, providing persistent objects whose operations are performed using atomic actions, providing persistent objects whose operations are both remotely invokable and performed using atomic actions, and finally the class is modified so that the state of the spreadsheet object is distributed amongst many nodes. For each implementation of the class, the reliability of the resulting implementation is examined.

In the next section, new techniques for structuring applications are developed. Initially, the problems which arise when using atomic actions with nested structures to construct applications are described, then new techniques which ease the construction of such applications are explained. The techniques which are described are: serialising actions, top–level independent actions, N–level independent actions, common actions, and glued actions.

Following the section about the new structuring techniques, the concept of coloured actions is introduced.The differences between atomic atomics and coloured actions are described. Colour actions are shown to be a uniform mechanism by which: serialising actions, top–level independent actions, N–level independent actions, and glued actions, can be implemented. Finally in the section, examples of applications which show the power of coloured actions are described.

In the next section, an outline of the modifications required to the Arjuna system to support coloured actions is given, and shows that coloured actions are no more difficult to implement then atomic actions.

The final section describes the design of three reliable distributed applications. Each application is examined with regard to being implemented using atomic actions and the new structuring techniques. The three applications which are examined are: a distributed make program, a reliable distributed spreadsheet, and an application which arranges meetings.

9.2 Main contributions

The contributions made by this thesis can be summarised as follows:

(i) Identification of shortcomings in the currently proposed atomic action based structuring techniques (nesting and concurrent actions) for composing distributed applications.

(ii) Proposing new structuring techniques: serialising actions, top–level independent actions, N–level independent actions, common actions, and glued actions.

(iii) Coloured actions as a generic mechanism for structuring the above types of actions.

(iv) A practical object oriented implementation for supporting coloured actions from which the new actions can be derived.

9.3 Future work

This section will outline the potential areas of future work, which are related to the ideas of structuring techniques and coloured actions.

The new structuring techniques outlined in this thesis are not presented as a fully comprehensive set of additional tools for the construction of applications. It is likely that other structuring techniques would be found necessary when larger classes of applications are examined.

A methodology for constructing reliable applications, using coloured actions as the low level mechanisms, is a potential areas of research. This thesis uses the flexibility of coloured actions as a basis for the implementation of the new structuring techniques proposed. The issue which would have to be addressed by such methodology are: automatically deriving the structure and colours of the coloured actions from the application requirements, and determining the mode and colour of locks which are placed on objects.

The formalisation of the properties of coloured actions could be a profitable undertaking, possibly showing limitation on the uses, or even showing ways of using coloured actions which have as yet not been explored. From such a formalisation it could be possible to suggest new structuring techniques for coloured actions.

Another area of future work could be to examine if new forms of locks can be suggested, the modes of these locks being dependent on the colour of the lock. For example, an exclusive "coloured" read lock, would be a shared read lock with respect to other locks of its colour, but exclusive with respect to other colours. Another form of locks which could be integrated into the coloured action model are intention locks [Gray 76], so improving the ability of coloured actions to manipulate hierarchical structures.

Finally, and obviously, a full implementation of the new structuring techniques using coloured actions is necessary in order to evaluate them in practice by building the types of applications discussed in this thesis.

References

[Allchin 83]

J. E. Allchin and M. S. McKendry, "Synchronizing and Recovery of Actions", Proceedings of the 2nd Annual ACM Symposium on Principles of Distributed Computing, pp. 31–44, August 1983.

[Anderson 81]

T. Anderson and P. A. Lee, "Fault Tolerance: Principles and Practice", Prentice–Hill, 1981.

[Birman 87]

K. Birman and T. Joseph, "Exploiting virtual synchrony in distributed systems", in 11th Symposium on Operating System Principles, pp. 123–138, ACM SIGOPS, November 1987.

[Birman 88]

K. Birman, T. Joseph and F. Schmuck, "ISIS – A Distributed Programming User's Guide and Reference Manual", The ISIS Project, Department of Computer Science, Cornell University, Ithaca, NY, 14853, March 1988.

[Birwhistle 73]

G. M. Birwhistle, O–J. Dahl, B. Myhrhaug and K. Nygaard, "Simula Begin", Academic Press, 1973.

[Bookbinder 89]

D. J. Bookbinder, "The Lotus Guide to 1–2–3 Release 3", Addison–Wesley, 1989.

[Campbell 86]

R. H. Campbell and B. Randell, "Error Recovery in Asynchronous Systems", in IEEE Transactions on Software Engineering, Vol. SE–12, No. 8, pp. 211–241, August 1986.

[Cooper 83]

E. C. Cooper, "Writing Distributed Programs with Courier", in UNIX Programmer's Manual – User Contributed Software, University of California, Berkeley, 1983.

[Dahl 70]

O–J, Dahl, B. Myhrhaug and K. Nygaard, "SIMULA Common Base Language", Norwegian Computing Centre S–22, Oslo, Norway, 1970.

[Davies 78]

C. T. Davies, "Data Processing Spheres of Control", in IBM Systems Journal, Vol. 17, No. 2, pp. 179–198, 1978.

[Dixon 87a]

G. N. Dixon and S. K. Shrivastava, "Exploiting Type-Inheritance Facilities to Implement Recovery in Object Based Systems", in Proceedings of 6th Symposium on Reliability in Distributed Software and Database Systems, Williamsburg, pp. 107-114, March 1987.

[Dixon 87b]

G. N. Dixon, S. K. Shrivastava and G. D. Parrington, "Managing Persistent Objects in Arjuna: A System for Reliable Distributed Computing", in Proceedings of the Workshop on Persistent Object Systems, Persistent Programming Research Report 44, Department of Computational Science, University of St. Andrews, August 1987.

[Dixon 88]

G. N. Dixon, "Object Management for Persistence and Recoverability", Ph.D Thesis, Technical Report TR/276, Computing Laboratory, University of Newcastle upon Tyne, December 1988.

[Dixon 89]

G. N. Dixon, G. D. Parrington, S. K. Shrivastava and S. M. Wheater, "The Treatment of Persistent Objects in Arjuna", in Proceedings of ECOOP 89. European Conference on Object Oriented Programming, University of Nottingham, pp. 169-204, July 1989. (also in The Computer Journal, Vol. 32, No. 4, pp. 323-332, April 1989)

[Eswaran 76]

K. P. Eswaran, J. N. Gray, R. A. Lorie and I. L. Traiger, "The Notion of Consistency and Predicate Locks in a Database System", in Communications of the ACM, Vol. 19, No. 11, pp. 624-633, November, 1976.

[Feldman 79]

S. I. Feldman, "Make: a program for maintaining computer programs", in Software: Practice and Experience, Vol. 9, No. 4, pp. 255-265, April 1979.

[Gibbons 87]

P. B. Gibbons, "A Stub Generator for Multi-language RPC in Heterogeneous Environments", in IEEE Transactions on Software Engineering, Vol. SE-13, No. 1, January, 1987.

[Goldberg 83]

A. Goldberg and D. Robson, "Smalltalk-80: The Language and its Implementation", Addison-Wesley, 1983.

[Gray 76]

J. N. Gray, R. A. Lorie, G. R. Putzolu and I. L. Traiger, "Granularity of Locks and Degrees of Consistency in a Shared Data Base", in Modelling in Data Base Management Systems, ed. G. M. Nijssen, North-Holland, 1976.

[Gray 78]

J. N. Gray, "Notes on Data Base Operating Systems", in Operating Systems An Advanced Course, Lecture Notes in Computing Science, Vol. 60, Springer-Varlag 1978.

[Horning 74]

J. J. Horning, H. C. Lauer, P. M. Melliar-Smith and B. Randell, "A Program Structure for Error Detection and Recovery", in Lecture Notes in Computer Science, Vol. 16, Springer-Verlag, pp 171-187, 1974.

[Hughes 86]

F. L. Hughes, "Multicast Communications in Distributed Systems", Ph.D Thesis, Computing Laboratory, University of Newcastle upon Tyne, October 1986.

[LeBlanc 85]

R. J. LeBlanc and C. T. Wilkes, "Systems Programming with Objects and Actions", in Proceedings of the 5th International Conference on Distributed Computing Systems, pp. 132-139, May 1985.

[Lippman 89]

S. B. Lippman, "C++ Primer", Addison-Wesley, 1989.

[Liskov 79]

B. Liskov, R. Atkinson, T. Bloom, J. E. B. Moss, C. Schaffert, R. Scheifler and A. Snyder, "Clu Reference Manual", Technical Report MIT/LCS/TR-225, MIT Laboratory for Computer Science, Cambridge, Mass., October 1979.

[Liskov 84]

B. Liskov, "Overview of the Argus Language and System", in Programming Methodology Group Memo 40, Massachusetts Institute of Technology Laboratory for Computer Science, February, 1984.

[Liskov 88]

B. Liskov, "Distributed Programming in Argus", in Communications of the ACM, Vol. 31, No. 3, pp 300-312, March 1988.

[Lomet 77]

D. B. Lomet, "Process structure, synchronisation and recovery using atomic actions", in Proceedings of ACM Conference on Language Design for Reliable Software, SIGPLAN Notices, Vol. 12, No. 3, pp. 128-137, March 1977.

[Kernighan 78]

B. W. Kernighan and D. M. Ritchie, "The C Programming Language", Prentice-Hall, Englewood Cliffs, New Jersey, 1979.

[Kung 81]

H. T. Kung and J. T. Robinson, "On Optimistic Methods for Concurrency Control", in ACM Transaction on Database Systems, Vol. 6, No. 2, pp. 213-226, June, 1981.

[Marshall 80]

L. F. Marshall, "An Error Recovery Scheme for Concurrent Processes", Ph.D Thesis, Computing Laboratory, University of Newcastle upon Tyne, 1980.

[McBride 89]

P. K. McBride, "Successful Spreadsheets using SuperCalc" Heinemann Professional, 1989.

[Merlin 78]

P. M. Merlin and B. Randell, "State Restoration in Distributed Systems", in Digest of Papers FTCS-8: Eighth Annual International Symposium on Fault-Tolerant Computing, Toulouse, pp. 129-134, June 1978.

[Meyer 88]

B. Meyer, "Object-oriented software construction", Prentice-Hall, 1988.

[Moss 81]

J. E. B. Moss, "Nested Transactions: An Approach to Reliable Distributed Computing", Technical Report MIT/LCS/TR-260, Massachusetts Institute of Technology Laboratory for Computer Science, April, 1981.

[Nelson 81]

B. J. Nelson, "Remote Procedure Call", Ph.D. Thesis, Technical Report CMU-CS-81-119, Department of Computer Science, Carnegie-Mellon University, 1981.

[Nett 85]

E. Nett et al, "Profemo: Design and Implementation of a Fault Tolerant Distributed System Architecture", GMD-Studien, No. 100, Technical Report, GMD, St. Augustin, June, 1985.

[Panzieri 85]

F. Panzieri, "Design and Development of Communication protocols for local area networks", Technical Report TR/197, Computing Laboratory, University of Newcastle upon Tyne, 1985.

[Panzieri 88]

F. Panzieri and S. K. Shrivastava, "Rajdoot : a remote procedure call mechanism supporting orphan detection and killing", in IEEE Transactions on Software Engineering, Vol. SE-14, No. 1, pp. 30-37, January 1988.

[Parrington 88a]

G. D. Parrington, "Management of Concurrency in a Reliable Object-Oriented Computing System", Ph.D Thesis, Technical Report TR/277, Computing Laboratory, University of Newcastle upon Tyne, December 1988.

[Parrington 88b]

G. D. Parrington and S. K. Shrivastava, "Implementing Concurrency Control in Reliable Distributed Object-Oriented Systems", in Proceedings of ECOOP 88. European Conference on Object Oriented Programming, Norway, August 1988.

[Parrington 89]

G. D. Parrington, "Distributed Programming in C++ via Stub Generation", in preparation, Computing Laboratory, University of Newcastle upon Tyne, 1989.

[Piersol 86]

K. W. Piersol, "Object Oriented Spreadsheets: The Analytic Spreadsheet Package", in OOPSLA '86 Conference Proceedings, pp. 385–390, September 1986.

[Pu 88]

C. Pu, G. Kaiser, and N. Hutchinson, "Split-Transactions for Open-Ended Activities", in Proceedings of the 14th International Conference on Very Large Data Bases, pp. 26–37, Los Angeles, California, September, 1988.

[Randell 75]

B. Randell, "System Structure for Software Fault Tolerance", in IEEE Transactions on Software Engineering", Vol. SE-1, No. 2, pp. 220–232, June 1975.

[Rosenkrantz 78]

D. J. Rosenkrantz, R. E. Stearns and P. M. Lewis II, "System Level Concurrency Control for Distributed Database Systems", in ACM Transactions on Database Systems, Vol. 3, No. 2, pp. 178–198, June 1978.

[Schwarz 84]

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

[Sha 85]

L. Sha, "Modular concurrency control and failure recovery – Consistency, correctness and optimality", Ph.D. dissertation, Dep. Elec. Comput. Eng., Carnegie-Mellon University, 1985

[Sha 88]

L. Sha, J. P. Lehoczky and E.D. Jensen, "Modular concurrency control and failure recovery", in IEEE Transactions on Computers, Vol. 37, No. 2, February 1988.

[Schaffert 86]

C. Schaffert, T. Cooper, B. Bullis, M. Kilian and C. Wilpolt, "An Introduction to Trellis/Owl", in OOPSLA '86 Conference Proceedings, pp. 9–16, September 1986.

[Schlichting 83]

R. D. Schlichting and F. B. Schneider, "Fail-Stop Processors: An Approach to Designing Fault-Tolerant Computing Systems", ACM Transactions on Computing Systems, Vol. 1, No. 3, 1983.

[Shrivastava 82]

S. K. Shrivastava, "A Dependency, Commitment and Recovery Model for Atomic Actions", in Second Symposium on Reliability in Distributed Software and Database Systems, University of Pittsburgh, Pittsburgh, PA, July 1982.

[Shrivastava 87]

S. K. Shrivastava, L. Mancini and B. Randell, "On the Duality of Fault Tolerant System Structures", in Lecture Notes in Computer Science, Vol. 309, Experiences with Distributed Systems, pp. 19–37, Springer-Verlag 1987.

[Shrivastava 88]

S. K. Shrivastava, G. N. Dixon, F. Hedayati, G. D. Parrington and S. M. Wheater, "A Technical Overview of Arjuna: A System for Reliable Distributed Computing", Proceedings of UK IT 88 Conference, July 1988.

[Spector 87]

A. Z. Spector, D. Thompson, R. F. Pausch, J. L. Eppinger, D. Duchamp, R. Draves, D. S. Daniels, and J. J. Bloch, "Camelot: A Distributed Transaction Facility for Mach and the Internet - An Interim Report", Technical Report CMU-CS-86-129, Department of Computer Science, Carnegie-Mellon University, June 1987.

[Stroustrup 86]

B. Stroustrup, "The C++ Programming Language", Addison Wesley, 1986.

[Taylor 86]

D. J. Taylor, "How Big Can an Atomic Action Be", in Proceedings of the 5th IEEE Symposium on Reliability in Distributed Software and Database Systems, pp. 121-124, January 1986.

[Tanenbaum 88]

A. S. Tanenbaum, "Computer Networks", Prentice-Hill, 1988

[Walker 83]

B. J. Walker, G. J. Popek, R. English, C. Kline and G. Thiel, "The LOCUS Distributed Operating System", in Proceedings of the 9th ACM Symposium on Operating System Principles, Bretton Woods, New Hampshire, pp. 49-70, October 1983.

[Walker 85]

B. J. Walker, "The Locus Distributed System Architecture", MIT Press, 1985.

[XEROX 81]

XEROX, "Courier: The Remote Procedure Call Protocol", XSIS 038112, Stamford, Connecticut, December 1981.