Object Management for Persistence and Recoverability

by

Graeme N Dixon

PhD Thesis

The University of Newcastle upon Tyne

Computing Laboratory

July 1988

Abstract

As distribution becomes commonplace, there is a growing requirement for applications that behave reliably when node or network failures occur. To support reliability, operations on the components of a distributed application may be declared to occur within the scope of an atomic action. This thesis describes how atomic actions may be supported in an environment consisting of applications that operate on objects.

To support the failure atomicity and permanence of effect properties of an atomic action, the objects accessed within the scope of an atomic action must be recoverable and persistent. This thesis describes how these properties may be added to the class of an object. The approach adopted is to provide a class that implements recovery and persistence mechanisms, and derive new classes from this base class. By refining inherited operations so that recovery and persistence is specific to that class, recoverable and persistent objects may be easily produced.

This thesis also describes how an atomic action may be implemented as a class, so that instances of the class are atomic actions which manage the recoverable and persistent objects. Multiple instance declarations produce nested atomic actions, and the atomic action class also inherits persistence so that short-term commit information may be saved in an object store which is used to maintain the passive state of persistent objects.

Since the mechanisms and classes that support recovery, persistence, and atomic actions are constructed using the feature of an object-oriented language, they may be implemented in environments that provide suitable support for objects and object-oriented programming languages.

Acknowledgments

First and foremost, I owe a debt of gratitude to my supervisor, Santosh Shrivastava, for starting the Arjuna project and allowing me to become a member, for reading and commenting on the many drafts of this thesis, and in particular, for suggesting my research topic.

I am also grateful to Pete Lee for his constructive criticism of a later draft of this thesis, and the following people for reading and commenting on parts of it: Graham Parrington, Jim Lyons, Brian Randell, and Krithi Ramamritham.

I would also like to thank the members of the Arjuna project, and the Computing Laboratory, who have contributed to productive environment in which to work.

Last, but by no means least, I would like to thank my parents for the help and encouragement they have given me throughout my research, and particularly during the time I have spent working on this thesis.

Financial support for this work was provided by grant from the Science and Engineering Research Council, and a SERC/Alvey grant in Software Engineering.

Table of Contents

Abstract I

Acknowledgments II

Table of Contents III

List of Figures VI

Tables IX

1 Introduction 1

1.1 Atomic actions 2

1.2 Object-oriented programming 5

1.3 Programming in distributed systems 7

1.4 Programming with objects and actions in a distributed system 9

1.5 Thesis aims 10

1.6 Thesis structure 10

2 Reliable Programming in a Distributed System 13

2.1 The distributed system model 14

2.2 Fault-tolerance terminology 15

2.3 Reliability issues 17

2.4 Modelling and masking faults 19

2.5 Object-oriented issues 23

2.5.1 Data abstraction and encapsulation 23

2.5.2 Inheritance 25

2.6 Arjuna 28

2.7 A review of reliable distributed object-based systems 35

2.7.1 Argus 35

2.7.2 Clouds 36

2.7.3 Profemo 36

2.7.4 Camelot/Avalon 37

2.7.5 Other related projects 38

2.7.6 Comparison with Arjuna 40

2.8 Concluding Remarks 43

3 Constructing Atomic Actions 44

3.1 The atomic action model 45

3.2.1 Atomic action events 49

3.2 The functionality required by an atomic action 51

3.3 Constructing distributed atomic actions 54

3.4 The implementation of an atomic action 59

3.5 The operation of the class `AtomicAction` 65

3.5.1 A simple example 67

3.6 Commitment and crash recovery 71

3.7 The design of a distributed atomic action 75

3.8 Atomic actions as objects 80

3.9 Concluding remarks 82

4 Recoverability 84

4.1 Providing recoverability 85

4.2 Constructing recoverable objects 88

4.3 Managing recoverable objects 94

4.4 Implementing recoverable objects 95

4.5 Implementing state based recovery 97

4.6 Implementing operation based recovery 103

4.7 Constructing a new recoverable class 109

4.8 An assessment of constructing recoverable classes using inheritance 115

4.9 Concluding remarks 117

5 Persistence 119

5.1 Permanence of effect and persistence 120

5.2 Supporting persistence 127

5.3 Implementing persistence 131

5.3.1 A simple persistent class 141

5.4 The design of an object store 143

5.5 Implementing an object store 149

5.6 Concurrency control and the object store 152

5.7 An assessment of constructing persistent classes using inheritance 154

5.8 Concluding remarks 157

6 Performance and Optimisations 158

6.1 A banking system 159

6.2 Measuring the performance 166

6.3 Concluding remarks 174

7 Conclusions 178

7.1 Thesis summary 178

7.2 Future work 187

References 192

List of Figures

Figure 2.1: The class `Date` 24

Figure 2.2: Subtyping and multiple inheritance 26

Figure 2.3: The class `Buffer` 26

Figure 2.4: The class `CircularBuffer` 27

Figure 2.5: Local and distributed programs 29

Figure 2.6: The communication architecture 29

Figure 2.7: The Arjuna class `X` 31

Figure 2.8: The state and interface of an Arjuna class called `X` 32

Figure 3.1: Nested atomic actions 46

Figure 3.2: Action events 49

Figure 3.3: Atomic action states and events 50

Figure 3.4: Nested atomic actions and management information 52

Figure 3.5: After the commitment of the atomic action `B` 53

Figure 3.6: The composition of an atomic action 58

Figure 3.7 : The class `AtomicAction` 59

Figure 3.8 : Using the class `AtomicAction` 60

Figure 3.9: The class `AbstractRecord` 61

Figure 3.10: The atomic action subsystem class hierarchy 64

Figure 3.11: An example using an `AtomicAction` 67

Figure 3.12: The `ExampleClass update` operation 67

Figure 3.13: Objects created during `AtomicAction A` 68

Figure 3.14: Objects created before `B` commits 69

Figure 3.15: Objects in existence after `B` commits 70

Figure 3.16: During phase one 72

Figure 3.17: At the end of phase one 73

Figure 3.18: A distributed atomic action 78

Figure 3.19: Enforcing scope 81

Figure 3.20: Enforcing scope using macros 82

Figure 4.1: The *container* approach 90

Figure 4.2: The *unrecoverable inheritance* approach 90

Figure 4.3: The *recoverable inheritance* approach 92

Figure 4.4: The *multiple inheritance* approach 93

Figure 4.5: The class `Object` 99

Figure 4.6: The class `RecoverableInteger` 100

Figure 4.7: `RecoverableInteger` assignment 100

Figure 4.8: `RecoverableInteger save_state` operation 100

Figure 4.9: `RecoverableInteger restore_state` operation 101

Figure 4.10: `ObjectStateRecord abort` operation 102

Figure 4.11: The class `Operation` 103

Figure 4.12: The class `RecoverableStack` 104

Figure 4.13: The class `StackOperationLog` 104

Figure 4.14: `RecoverableStack pop` operation 105

Figure 4.15: The class `Printer` 107

Figure 4.16: The `RecoverablePrinter print` operation 108

Figure 4.17: The `PrinterRecord abort` operation 108

Figure 4.18: The class `RString` 110

Figure 4.19: The class `UString` 111

Figure 4.20: The class `URString` 112

Figure 4.21: The class `MRString` 112

Figure 4.22: The `MRString` assignment operation 113

Figure 4.23: The `MRString save_state` operation 114

Figure 5.1: Persistent object state transitions 123

Figure 5.2: A structured object 130

Figure 5.3: The class `Object` 134

Figure 5.4: Movement of persistent data 135
Figure 5.5: The `Object activate` operation 135
Figure 5.6: The `Object deactivate` operation 136
Figure 5.7: The `PersistentRecord top_level_prepare` operation 137
Figure 5.8: The persistent class `File` 141
Figure 5.9: The `File` constructor 142
Figure 5.10: A single level object store 145
Figure 5.11: The class `Entry` 145
Figure 5.12: The class `ObjectStore` 146
Figure 5.13: A two-level object store 148
Figure 5.14: Object store implementation 151

Figure 6.1: The class `Account` 159
Figure 6.2: The class `Customer` 160
Figure 6.3: The class `Bank` 161
Figure 6.4: The relationship between objects 161
Figure 6.5: The new declaration of class `Customer` 163
Figure 6.6: The new declaration of class `Bank` 164
Figure 6.7: The `Customer` transfer operation 165
Figure 6.8: The performance of the environment 167
Figure 6.9: The modified operation overhead 168
Figure 6.10: Recovery/persistent data for 3 `Accounts` 171
Figure 6.11: `Customer` optimisations 172
Figure 6.12: `Bank` and `Customer` optimisations 172

Tables

Table 1: Atomic action execution times 169

Table 2: `Customer` class optimisations 173

Table 3: `Bank` + `Customer` class optimisations 173

Chapter 1

Introduction

Over the last few years the impact of computing systems on our daily lives has steadily increased to the extent that we are now seeing their use in many, hitherto unthought of, application areas. Much of this proliferation can be traced to the introduction of low-cost microprocessors, which have rapidly developed to provide computational power beyond that of recent multi-user mainframe computers. Additional factors, such as the decrease in cost of semiconductor memory, have all led to the construction of powerful personal computers which are intended for the dedicated use of a single user.

As the technology used to build powerful personal computers has advanced, so too has the technology behind interconnection media such as *local area networks*. Connecting computers (*nodes*) together using a local area network to produce a *distributed system* is becoming commonplace. One reason is the flexibility of a distributed system since extra nodes may be added to the network as the demands on system increase. In addition, distributed systems are not susceptible to the single point of failure of a centralised system, where all executing computations are affected by the failure, as only those computations executing on the failing node will be affected.

To fully realise the advantages of distribution, a computation must be able to access the resources which are available on other nodes in the network. A computation which has this capability is termed a *distributed computation*, and consists of a series of operations on a collection of both local and remote resources. The execution of an operation on a resource is also a computation that in turn

may consist of a series of further operations on other local and/or remote resources.

While distribution may remove the single point of failure of a centralised system, a distributed computation becomes susceptible to remote node crashes, and failures in communication between the distributed computation and its remote resources. Independent failures of this type will result in the abnormal behaviour of the distributed computation. To benefit from distribution therefore, distributed computations should be *reliable* requiring consistent behaviour in the face of node crashes and communication failures.

This thesis describes how distributed applications may be constructed that operate consistently when node or network failures occur. To model this environment the *object-oriented* paradigm is employed and to support consistency *atomic actions* are used. The areas of research addressed by this thesis are how to implement atomic actions, and how to provide support for constructing a class so that instances of that class may be used within an atomic action. Since the purpose of using atomic actions is to support consistent behaviour during node or network failures in a distributed system, distribution aspects are also discussed but are not the main topic of this thesis. The rest of this chapter describes the components of this *object and action environment*, beginning with the properties of an atomic action.

1.1 Atomic actions

An atomic action is a computing abstraction that encapsulates a computation and controls its outcome, ensuring that the computation appears to execute in isolation to completion. To provide control over the encapsulated computation, an atomic action exhibits three properties; *failure atomicity*, *serialisability*, and *permanence of effect*.

Failure atomicity ensures that the computation either terminates with the intended results, or else all of the effects of the computation are undone and the system is restored to the state it was in at the beginning of the atomic action. If during the execution of a computation a fault causes the system to reach an erroneous state then some form of error recovery is required. On detection of an error, the error recovery mechanism provided by the containing atomic action may be employed to recover the system state by *aborting* the atomic action. By default, an atomic action provides *backward error recovery* where the system state affected by the computation is restored to the state held at the beginning of the atomic action. An alternative form of recovery is *forward error recovery*, where errors in the system are removed by manipulating the system to produce a new state.

The serialisability property of an atomic action ensures that if concurrent computations wish to use common resources then they may only access the required resources in a manner such that the effect is equivalent to that of the computations occurring serially (i.e. one after another). In many cases, such as in the case of the most common form of concurrency control which is *locking*, a computation is guaranteed exclusive access to a shared resource if the computation is to modify the resource, with rules such as *two-phase locking* ensuring that concurrent computations are kept *interference free* until they terminate. Serialisability also guarantees the *indivisibility* of the computation within the atomic action by ensuring that the intermediate states produced are not accessible, and as a result cannot be viewed, outside the computation encapsulated by the atomic action.

The remaining property of an atomic action, that of permanence of effect, guarantees that the results of a computation encapsulated by an atomic action will not be lost despite failures of the system after the successful completion of the atomic action, thereby providing a consistent state with which to restart the

system should this be necessary. To provide the permanence of effect property, newly established system state must be recorded in storage that is tolerant to node crashes and media failures.

The concepts and practice behind the use of atomic actions are well understood and have been utilised for a number of years in the database field where they are commonly termed *transactions*. It is only relatively recently however, that systems have been produced where atomic actions may be nested, and in which resources other than databases can be used.

When an atomic action is nested within another atomic action (which is termed its *parent*), the results of the nested atomic action are dependent upon the outcome of the parent atomic action. As a result the permanence of effect property may be relaxed for nested atomic actions thereby providing failure atomicity without the expense of ensuring that the modified system state becomes permanent. In addition, nested atomic actions offer greater flexibility by enabling failures to be localised, since they can be aborted independently of their containing (*parent*) action.

Nested atomic actions are also advantageous in systems that allow resources to be distributed. In this environment the inaccessibility of a remote resource, due to either a network or remote node fault, may be contained by a nested atomic action. Should the resources also be replicated, then another node with the required resource may be chosen and the operation attempted anew at this alternative node once suitable error recovery has taken place.

An additional advantage of nested atomic actions is that an existing computation which employs atomic actions may be used as a sub-computation of an application even though the atomic action in the sub-computation may be nested within any atomic actions declared in the application. Clearly, if atomic

actions could not be nested then the re-use of the such a computation by an application would not be possible.

The discussion so far has described why atomic actions provide useful support when constructing reliable distributed computations. But what are the resources manipulated by a computation, and how can they be modelled? One of the best programming paradigms currently available for modelling a system is the *object-oriented* paradigm.

1.2 Object-oriented programming

The most fundamental element of object-oriented programming is the *object.* An object, which has an internal state and a set of operations that characterise its behaviour, may be defined using an object-oriented programming language which supports *data abstraction, encapsulation* and *inheritance.*

Object-oriented languages provide the *class* construct which supports data abstraction by hiding the representation of data and implementation of operations that constitute an abstraction. Instances of a class are objects, and all objects of the same class exhibit common behaviour.

If the only way to modify or access the state contained within an instance of a class is to invoke one of the public operations, then the data abstraction provided by the language supports *encapsulation.* If a language supports encapsulation then changes to the representation or implementation of a class can be made without affecting programs which use that class.

Another important property which may be provided by an object-oriented language is *inheritance.* This property enables new classes to be derived from existing classes, with the new class *inheriting* the state and operations of the existing class. The new class may in turn *refine* the existing class by adding extra functionality or by providing a restricted interface to the inherited state. The

main benefit of inheritance is the way that it may be used to avoid the re-implementation of common features. In a number of systems this takes the form of a class hierarchy in which common functionality is shared between classes that belong to the hierarchy.

Other requirements for true object-oriented classification have been proposed, many of which are shown by what is regarded as the archetypal object-oriented language - Smalltalk. Two of the more important properties are *message passing* and *dynamic-binding*, both of which are often associated together since messages contain names rather than values, the binding of name to value occurring dynamically when the value is required. In languages such as Smalltalk, the invocation of an operation on an object occurs by sending a message to the object, with the receiver of the message performing the requested operation and returning the result in a reply message. The alternative approach to operation invocation is based on the procedure call mechanism, where an operation is a conventional procedure call with arguments which may return a result. In addition, the process which executes the implementation of an operation may differ depending upon whether the objects are *active* or *passive*. In an active system an object is managed by a process that executes the operations on behalf of a client (the application). In contrast, in a passive system the operations are executed by the process that is executing the application.

In the following discussions it will be assumed that the invocation of an operation occurs via a procedure call and that objects are passive entities. In addition, the three properties (data abstraction, encapsulation, and inheritance) described above are assumed when object-oriented terminology is employed.

1.3 Programming in distributed systems

To enable a computation to access a remote resource (or object) requires a method for invoking an operation on the remote object. Since the only means of accessing an object is through one of its operations, it follows that access to remote objects requires a (network) protocol which provides the abstraction of invoking an operation (procedure) on the remote object. The generic term *remote procedure call* is used to describe a network protocol of this type, which involves the transmission of messages between local and remote nodes containing the request for service (the operation on the remote object) and any reply (containing the result of the operation).

When a computation (the *client*) invokes an operation on a remote object, the remote procedure call mechanism creates a process, termed the *server*, on the remote node to manage the required object and perform the requested operation. The remote procedure call mechanism also contends with the faults in communication that can occur, such as lost or multiple messages. Various semantics are associated with different remote procedure call mechanisms depending upon the effect of the call when it succeeds. For example, *exactly once* semantics state that if the remote procedure call succeeds, then only one execution of the requested operation was performed by the server. If the remote procedure call failed then partial execution of the operation may have occurred, and it is the responsibility of the client to ensure that the attempted operation is recovered.

A distributed computation can be constructed using a programming language that supports distributed programming, or by using a conventional (non-distributed) language with support for remote procedure call invocations. When a non-distributed programming language is employed, operations on local and remote objects are unlikely to be transparent since the remote procedure call

mechanism is usually built on top of a message passing subsystem which is a separate component of the system. As a result the invocation of an operation involving a remote procedure call is likely to have a different syntax from a local invocation.

When a distributed programming system consists of a conventional language and remote procedure call mechanism, a common approach to providing identical syntax for both local and remote invocations is to employ *stubs* to hide the remote procedure call invocations. In this way, a distributed computation will contain a stub object for each remote object, the stub object's interface being identical to the remote object. When an operation is invoked by a computation on the stub object, a corresponding remote procedure call to the server managing the remote object is made by the stub object's implementation.

Stubs can be produced automatically by a *stub generator* from an interface definition which specifies the operations that may be invoked on an object. Two stubs are generated: a client stub for the local computation, and a server stub for the server. The distributed computation is constructed using the client stub, and the server is constructed using the server stub and the implementation of the class of the object that the server is managing. The construction of mechanisms which support access to remote objects has been extensively researched and the solutions are well known. As a result, this thesis assumes that support for remote procedure calls, and stubs, are available and need not be considered in any greater detail.

1.4 Programming with objects and actions in a distributed system

The above descriptions have introduced the basic features of the type of system which is the concern and interest of this thesis. The system aims to be reliable and resources in the system may be distributed. Application programs (computations) which execute within the system consist of operations on objects which are the resources of the system. The objects used by a computation may be local or remote. If an object is remote, a computation invokes operations on the remote object's client stub object, that in turn makes a remote procedure call to the server managing the remote object.

To ensure a computation is reliable, surviving node crashes and communication failures, the operations that constitute the computation may be encapsulated in one or more atomic actions. The properties of an atomic actions maintain the consistency of the objects accessed during the execution of the atomic action should such failures occur. To guarantee that these properties can be met, the objects accessed during the atomic action must be *recoverable* to meet the failure atomicity property, *persistent* to meet the permanence of effect property, and provide *concurrency control* to meet the serialisability property.

Providing support for atomic actions in a distributed object-oriented programming system is an area of research that has been investigated by a number of research projects (a number of which are surveyed in chapter two), yet the approaches taken have been wide and varied. The aims of the work described in this thesis are detailed in the next section. This work forms part of a project called *Arjuna*, the aims of which are to develop a set of tools which enable reliable distributed programs, using objects and atomic actions, to be produced.

1.5 Thesis aims

This thesis examines how atomic actions may be supported, and the state of objects accessed within atomic actions managed, in a distributed programming system. Most projects which are addressing this area of research have concentrated on producing new languages or operating systems that provide the necessary support for objects and actions. The three main aims of the research presented in this thesis are therefore:

a) the design and implementation of mechanisms which enable the state of an object to be managed so that it is recoverable and persistent, and may be used from within an atomic action.

b) the design and implementation of atomic actions that control objects, which are recoverable, persistent, and provide concurrency control, when accessed within the scope of an atomic action.

c) the design and implementation of the mechanisms that meet the above two aims in a manner that may be generalised to other environments (i.e. not based on extending, or constructing a new, languages or operating systems).

1.6 Thesis structure

This thesis is organised as follows. The next chapter begins by expanding on the background to the work described in this thesis and defining the system model used to describe programming in a distributed environment using objects and actions. This description of the distributed environment is followed by a discussion of the properties of an object-oriented language, using the language C++ as an example. Given the model described at the start of the chapter, the later sections describe the approach taken by the Arjuna project, and compare and contrast a number of projects with similar goals to Arjuna.

Chapter three begins by defining an atomic action model and employing this model to discuss the functionality required to implement atomic actions in a distributed system. This functionality is used as the basis of a design that provides the abstraction of a non-distributed atomic action. After a description of the implementation of this design, the means by which atomic actions are committed and made crash recoverable are described. The following section describes how the non-distributed atomic action design may be extended to enable the atomic action abstraction to operate in a distributed environment. The penultimate section of chapter three discusses a number of issues relating to the use of objects to model atomic actions.

To support the failure atomicity property of an atomic action requires objects that are recoverable. The construction of a class that supports recoverability is therefore the subject of chapter four, which begins by describing two techniques that may be used to support the recovery of an object, and how the resulting recoverable objects are managed by an atomic action so that the failure atomicity property is met. To add recoverability to a class, the remainder of chapter four describes an approach based upon exploiting the inheritance property of an object-oriented language. The approach taken is to construct a class that provides recoverability and derive new recoverable classes from this base recoverable class so that recoverability is inherited. Implementations of the two recovery techniques are described, one is based upon managing the old state of an object, the other the operations invoked on an object. A number of examples of the flexibility of this approach to adding recoverability are included.

In chapter five it is shown how the mechanisms described in chapter four may be extended to enable a class that supports recoverability to also support persistence. The chapter describes how the scope rules of a language must be overcome and how an object must be moved automatically between permanent storage and the storage associated with an application for the object to persist.

The approaches taken by a number of persistent programming languages are considered, and the design and implementation of the mechanisms required described along with an object store design that provides a more suitable interface for the storage of objects.

Chapter six begins by developing a simple example to show how the mechanisms in earlier chapters of the thesis may be used to construct a class of objects that are recoverable and persistent. The remainder of the chapter describes a number of simple tests made on the example classes developed at the start of the chapter to illustrate the performance of the experimental prototype developed to test the soundness of the ideas presented in this thesis.

The final chapter speculates on future areas of work arising from the work presented in this thesis, provides a summary of the thesis and presents the conclusions of this work.

Chapter 2

Reliable Programming in a Distributed System

The previous chapter described how a distributed computation may behave abnormally when there are node crashes or communication failures during the invocation of operations on remote objects. Since abnormal behaviour is undesirable, there is a requirement for distributed computations that behave in a reliable manner when these types of failure occur. To construct a distributed computation that behaves reliably however, requires an understanding of the types of fault that give rise to node or communication failures and the techniques available for managing such failures. This chapter expands on these reliability issues, which were briefly described in the previous chapter, discussing the reliability requirements of a distributed computation and the technique used to model the resources accessed by a distributed computation.

The environment, models, and techniques described correspond to those adopted by Arjuna [Shrivastava *et al.* 87, Shrivastava *et al.* 88], a brief description of which is also included in this chapter. This description gives an overview of how an application may be constructed using the tools provided by Arjuna, covering issues such as how objects are named, created, made persistent, and controlled by atomic actions declared in an application program. A number of these issues are covered in greater depth in later chapters of this thesis.

The chapter begins by describing the distributed computation model and fault-tolerance terminology employed throughout this thesis. Section 2.3 discusses the reliability issues involved in producing a reliable distributed computation, and is followed by the various fault models and common techniques that may be applied

to tolerate any faults which may manifest themselves in a distributed computation.

Since resources are modelled using the object paradigm, the properties of an object-oriented language, and issues involving the use of such a language, are covered by the middle sections of this chapter. To illustrate a number of these issues, the language (C++ [Stroustrup 86]) chosen for the work described in this thesis will be used.

The final sections of this chapter describe Arjuna in greater detail, and end by briefly reviewing a number of research projects whose aims are similar to those addressed by Arjuna. A comparison of the functionality provided by each research project is made with that provided by Arjuna.

2.1 The distributed system model

The components of a distributed system, which were introduced in the previous chapter, are defined in this section so that later sections may discuss the reliability issues surrounding these components. The components are: the computing elements (nodes), the communication network which connects the nodes together, the computations which execute at the nodes, and the resources employed by the computations.

Each computation may be considered to consist of a series of sub-computations with each (sub-)computation consisting of a series of operations on *resources* which are modelled as objects. A system resource is provided by the underlying system and could be anything from the memory associated with a computation to the storage facilities provided by the operating system. User-defined objects are also considered to be resources since their implementation involves the use of system provided resources. The outermost computation is known as an *application program*, and will be referred to as an *application* in following

discussions to distinguish it from the sub-computations that constitute the application.

The objects used by a computation may be either local to, or remote from, the node where the computation is executing. It is assumed that when an operation is invoked on a remote object, a remote procedure call (*RPC*) [Nelson 81] is made which involves the computation (termed the *client*) sending value parameters to a *server* created by the RPC mechanism to manage and invoke the operation requested on the remote object. When the operation on the object at the server terminates, the RPC returns the result of the operation. The operation invocation at the server may also be considered to be a computation, as the implementation of the operation may involve the invocation of further operations on other objects in the system.

A distributed computation may behave abnormally if there are node or communication failures during the invocation of an operation on a remote object. To discuss these issues requires an understanding of the types of fault which result in a node or communication failure. The next section describes the terminology which will be used in section 2.3 to discuss the reliability issues of a distributed computation, and in section 2.4 to model the most common types of fault which are assumed to occur.

2.2 Fault-tolerance terminology

The terminology defined in this section is based on that of Anderson and Lee [Anderson and Lee 81, Anderson and Lee 82]. A *system* is defined to consist of a set of components that interact under the control of a *design*. The *components* of the system are also considered to be systems, as is the design.

The *internal state* of a system is an aggregation of the external states of all its components, and the *external state* of a system is an abstraction of its internal state. During the transition from one external state to another, the system may pass through a number of internal states for which the abstraction, and hence the external state is not defined. There is assumed to be an *authoritative specification* of the behaviour for a system which defines the external states of the system, the operations that can be applied to the system, the results of these operations and the transitions between external states caused by these operations.

A *failure* is said to occur when the behaviour of the system first deviates from that required by the specification. The *reliability* may be characterised by a function $R(t)$ that expresses the probability that no failure of the system will have occurred by the time t. An internal system state is termed an *erroneous state* when that state is such that there exists a point (within the specification of the use of the system), which after further processing by the system, will lead to failure. The internal part of the system that is incorrect is designated as an *error.*

The reason for the system reaching an erroneous state could be either the failure of a component or the design (or both). Since a component (or design) is a system, it may have failed because of its internal state being erroneous, the erroneous state being referred to as a *fault* in the system. A fault could be either a *component fault* (which can result in an eventual component failure) or a *design fault* (resulting in a design failure). Either of these failures cause the system to go from a valid state into an erroneous state, the transition being referred to as the *manifestation of the fault.*

There are two complementary approaches to constructing reliable systems. The first, *fault prevention*, aims to ensure that the system will not contain any faults by using *fault avoidance* techniques such as design methodologies, and *fault removal* techniques such as testing and verification. The second approach,

fault tolerance, attempts to prevent faults from causing system failures by detecting errors as they occur, applying damage assessment techniques to isolate the extent of the damaged system state, followed by error recovery to transform the erroneous system state to an error-free state allowing normal operation to occur. There are two error recovery techniques: *backward error recovery* or *forward error recovery*. During backward error recovery, the state of the component is replaced by a previous state which is assumed to be error-free, whereas when forward error recovery is employed, a state which is free from errors is established by manipulating the current state to produce a new (error-free) state.

The next section discusses the issues involved in constructing a reliable distributed computation, and the most common faults which occur. Section 2.4 models these faults, and describes the reliability measures available for masking and tolerating such faults to produce a reliable distributed computation.

2.3 Reliability issues

The important difference between a centralised and distributed computation is the possibility that components of the distributed computation may fail independently. Since a distributed computation lacks centralised control, components of the distributed computation may be executing normally while others have failed, resulting in the abnormal behaviour of the distributed computation. Since such abnormal behaviour is undesirable, fault tolerance techniques may be employed to ensure that faults in these components do not lead to failures so that abnormal behaviour does not occur.

To produce a system that operates reliably in the presence of a large number of different faults however, would require a very large amount of redundancy. The overhead that would result from large amounts of redundancy is usually considered to be economically unattractive, so that a tradeoff must be made by

limiting the number of faults for which tolerance is provided. Consequently, when constructing a fault-tolerant system, it is assumed that the type of fault for which no tolerance is provided are rare.

It is therefore assumed that the most common forms of fault which occur in a distributed system are communication and node faults, which result in *communication failures* and *node crashes* if no fault tolerance techniques were employed. In the presence of such failures, it is reasonable to require that a computation behaves *consistently*. A very simple consistency constraint is that of *failure atomicity*, whereby the computation either terminates normally producing the intended results, or is *aborted* undoing all effects of the computation. A distributed computation may be constructed to behave reliably if it meets this consistency constraint.

The next section models the faults which lead to communication failures or node crashes, and describes how they may be masked by the remote procedure call protocol. If the remote procedure call protocol is unable to mask a node or communication fault then the RPC will terminate *abnormally*. Atomic actions [Lomet 77, Gray 78] may be employed to handle the abnormal termination of an RPC so that the consistency constraint described above can be met.

By replicating resources, a distributed computation may be constructed to behave in a reliable manner, using atomic actions to handle the abnormal termination of an RPC, by allowing the operation which resulted in the abnormal termination of the RPC to be retried at an alternative node with a replica of the required resource.

2.4 Modelling and masking faults

Both the communication system and the nodes are assumed to be susceptible to the occurrence of faults. Faults in the communication system are modelled as being responsible for failures such as the loss, corruption, replication, or change in ordering of a message (relative to other messages) transmitted between the client (computation) and its servers.

Communication faults that corrupt messages during their transmission are assumed to be detected by well known techniques, such as checksums, with the corrupted message being discarded. The other communication failures which may occur (the replication, loss, or change in ordering of a message) are managed by the protocol used to provide communication: the remote procedure call. The protocol employed by the RPC mechanism will attempt to mask these failures. If *normal termination* of an RPC is not possible, due to the number of failures exceeding that specified by the protocol, then the RPC will terminate *abnormally*.

Node faults are modelled in a simple manner. A node either works perfectly or it crashes. If the node crashes then it is assumed that it immediately ceases to operate (in a non-malicious manner), and restarts executing within a finite amount of time. A node has two forms of storage: *volatile* and *non-volatile* storage. Data held in volatile storage is lost when the node crashes. Data held in non-volatile storage is permanent and assumed to be unaffected by the crash. Non-volatile storage can be implemented using techniques such as *stable storage* [Lampson and Sturgis 76] to provide this crash proof capability.

The crash of a remote node will affect communication between a distributed computation and its servers on the remote node. When the RPC protocol does not receive a reply to a message sent to a crashed node, the message will be resent until either a reply is received or the number of retries specified by the protocol is exceeded. Hence, a remote node failure will also result in the abnormal

termination of the RPC made by the client. Additional factors may result in the abnormal termination of an RPC, such as a partition in the communication network [Davidson *et al.* 84] producing a situation where a client and one (or more) of its servers are in different subnets. The client is assumed to be unable to determine from the abnormal termination of an RPC whether it is due to a communication failure, node crash, or a network partition.

The crash of the client node will result in the termination of the relationship between the client computation and its servers, so that if the servers have no other clients they will become *orphan computations* (or *orphans*). If orphans are not removed then they may consume resources which are required by other clients. To handle this situation, additional functionality may be added to the RPC mechanism so that a server can be recognised as an orphan and terminated. The RPC mechanism employed in the following discussion is assumed to have this orphan-killing capacity.

Since faults which occur during the execution of an operation on a remote object may result in the abnormal termination of the remote procedure call, this effective failure of the RPC must be handled in such as way that the computation which relies on the RPC is recovered so that the consistency constraint described in the previous section can be met. To manage this situation, an atomic action may be used. By enclosing a computation in an atomic action, the system state modified by the computation may be recovered to the state held at the start of the computation by *aborting* the atomic action.

The state restoration required during the abortion of an atomic action is provided by the failure atomicity property of the atomic action. Given that the system state consists of objects, then this involves restoring the abstract state of all objects modified by the computation. State restoration of this type is referred to as *backward error* recovery.

Backward error recovery may be provided by a number of techniques [Verhofstad 78]. For example, the old state of an object can be maintained so that it can replace the current state of the object when recovery is required. Alternatively, sufficient information about changes to the state of an object may be recorded, enabling the operations performed to be sequentially undone in reverse order, or compensating operations invoked, to produce the original object state.

To ensure that the new system state established by a successfully terminating computation is not lost through subsequent system failures, the permanence of effect property of an atomic action may be employed. Since the system state modified by a computation is simply the collection of objects utilised by the computation, those objects should become permanent when the atomic action which encapsulates the computation successfully terminates. Ensuring the permanence of an object involves saving its volatile state in non-volatile storage. An object which can be saved in non-volatile storage is said to be *persistent* [Atkinson *et al.* 83a].

An atomic action will also ensure that the consistency of the system is maintained during both client and server node failures, since the system state modified by the computation encapsulated by an atomic action will not be made permanent until the action successfully terminates. The process of successfully terminating an atomic action is known as *committing* an action, and involves the use of a *commit protocol* [Gray 78]. A node failure before the *commit point* (the point which successfully terminates the action) will undo the effects of the computation encapsulated by the atomic action.

As noted above, to guarantee the consistency of the system when an RPC terminates abnormally, the system state modified by the computation that invoked the RPC should be restored to its previous state. Since an atomic action

may be nested [Davies 73, Reed 78, Moss 81] within another (containing or parent) atomic action, the possibility of the abnormal termination of an RPC may be handled by enclosing the computation in a nested atomic action that may be aborted independently of the main computation. If the abnormal termination was a result of the failure of the server node then, after suitable damage assessment, an attempt may be made to tolerate the server node failure by retrying the RPC at an alternative node with an equivalent resource.

During the commitment of a nested atomic action, the objects modified by the computation encapsulated by the action need not become permanent since the containing action may abort and recover the state of those objects. Hence, the commitment of a nested action only involves making visible to the containing action the objects which have been modified, and as a result require state restoration should the containing action abort.

When the outermost (top-level) atomic action commits, a commit protocol is required to ensure that either all or none of the persistent objects that have been modified are made permanent, since a node failure during the execution of the action commit would be liable to leave inconsistencies as a subset of the new object states may be lost before they can be made permanent. In effect, the commit protocol ensure the atomicity of the commit operation.

In the above discussion, the system state accessed by a computation is assumed to be the collection of objects a computation may invoke operations upon. To fully understand the issues surrounding the use of objects to model system resources requires an understanding of the object-oriented paradigm. The next section expands on the description given in the last chapter, enabling the features of the paradigm to be employed with no further explanation when constructing classes whose instances are recoverable and persistent.

2.5 Object-oriented issues

An object-oriented language provides three properties which support the construction of objects: *data abstraction*, *encapsulation*, and *inheritance*. An object is an abstraction that has state and behaviour, with the behaviour being defined by a set of operations that are available on the object. The operations and the internal state of the object are defined by a class declaration, so that objects are instances of a class, and all instances of the same class share identical behaviour.

To support the abstraction properties, object-oriented languages provided facilities for data abstraction and encapsulation. These two properties enable the features of a resource to be considered abstractly by both the implementor and user of a class. The remaining property that is assumed to be provided by an object-oriented language is inheritance which enables the features of an existing class to be re-used by a newly declared class, thereby avoiding the re-implementation of common features.

This section describes these three properties in greater detail using examples written in the language C++. C++ [Stroustrup 86] is a superset of the language C [Kernighan and Ritchie 78], adding features which enable C++ to be considered as an object-oriented language. During the discussion of each of the properties, a number of the features of C++ will also be described.

2.5.1 Data abstraction and encapsulation

Perhaps the most important property of an object-oriented language is data abstraction. Data abstraction enables an object to be considered abstractly in terms of its behaviour rather than its state. For instance, an object that represents the date (the day, month and year) could have operations which enable the date to be set and the date maintained by the object retrieved. How the actual

date is maintained as the internal state of the object need not concern the user of the object, they need only be concerned with the behaviour, defined by the two operation provided by the class of the object. This is the power of data abstraction, since the implementation of the abstraction is removed from the behaviour that the abstraction provides.

Figure 2.1 illustrates one method of implementing a class that represents the

```
class Date
{
    int day;
    int month;
    int year;

public:
    Date(string);
    ~Date();

    void   set_date(string);
    string get_date();
};
```

Figure 2.1: The class `Date`

date (in the language C++). In this example, the date is represented by the class `Date` which provides two operations to set and read the date (`set_date` and `get_date` respectively). In the class `Date`, the internal representation of the date is implemented as three integers which, since both operations return or take the date in the form of a `string`, requires the implementation of the class to convert the internal representation of the date to or from a `string` when required.

An alternative implementation of the class `Date` could maintain the date as a `string` so that a conversion from the internal representation to the value returned by the `get_date` operation would be unnecessary. The encapsulation provided by an abstraction supports this ability to change the internals of a class without affecting users of the class when the interface (the operations provided by the class) does not change.

The class `Date` illustrates a few important features of the language C++. The operations which follow the `public` label may be invoked by a user with the exception of the two operations `Date` and `~Date`. The `Date` operation is known as a *constructor*, and provides a means by which the internal state of an object may be initialised. In this example, the constructor takes a `string` as an argument so that a `Date` object will always have an initial value. The `~Date` operation is the inverse operation and is known as the *destructor*. In the implementation of the destructor, any garbage collection required by an instance of the class may be performed before the `Date` object is destroyed. In practise, the language's compiler inserts calls to these operations when an instance of the class comes into and goes out of scope in an application.

2.5.2 Inheritance

When an object-oriented language supports inheritance, a new class may be declared that is *derived* from an existing *base* class, inheriting the state and operations of the base class, thereby avoiding the re-implementation of the functionality provided by the base class. If a derived class is only allowed a single base class then the language used to declare the derived class supports *sub-typing inheritance*, e.g. Smalltalk-80 [Goldberg and Robson 83]. When a class is allowed one or more base classes then the language supports *multiple inheritance*, e.g. Owl [Schaffert *et al.* 86]. Figure 2.2 illustrates the difference between sub-typing (Figure 2.2(a)) and multiple (Figure 2.2(b)) inheritance. The base class may also be termed the *super class* of the derived class, and the derived class is a *sub class* of the base class.

To illustrate how inheritance may be employed, consider a class that provides the abstraction of a fixed-size character buffer (called `Buffer`). Blocks of characters may be placed in the buffer using the operation `put`, or retrieved using the operation `get`. Each `put` or `get` operation moves the current position in the

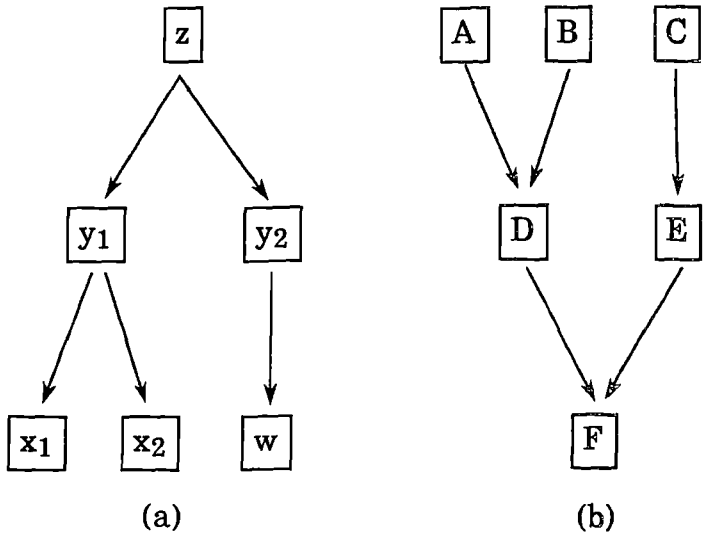

Figure 2.2: Subtyping and multiple inheritance

buffer, so that for example a sequence of `put` operations will place the data (passed as a pointer to a character array - the `char*` notation) in contiguous storage in the buffer. The `put` and `get` operations return the number of characters added or retrieved since the buffer size is bounded and both operations may move the current position to the end of the buffer. To move to a particular position in the buffer an operation called `position` is provided. The class declaration for the class `Buffer` is illustrated in Figure 2.3.

```
class Buffer
{
    char buffer[BufferSize];
    int  buffer_position;

public:
    Buffer();
    ~Buffer();

    int put(char*,int);
    int get(char*,int);
    int position(int);
};
```

Figure 2.3: The class `Buffer`

Under certain circumstances a buffer that is circular, where once the end of the buffer is encountered the current position moves back to the beginning of the buffer, may be more suitable. Since the class `Buffer` already implements much

of the functionality required by a circular buffer, a new class which represents a circular buffer may employ inheritance to avoid re-implementing this common functionality. The declaration of such a class (called `CircularBuffer`) is illustrated in Figure 2.4.

```
class CircularBuffer : Buffer
{
    int buffer_end;

public:
    CircularBuffer();
    ~CircularBuffer();

    int put(char*,int);
    int get(char*,int);
};
```

Figure 2.4: The class `CircularBuffer`

This declaration uses the C++ notation:

```
class DerivedClassName:BaseClassName
```

to define the inheritance of `Buffer` by `CircularBuffer`. In the implementation of the class `CircularBuffer` the inherited operations `put` and `get` are refined so that the class provides the abstraction of a circular buffer. To maintain the circularity, `CircularBuffer` adds extra state (the integer `buffer_end`) to the state that is inherited from the class `Buffer`.

In systems such as Smalltalk-80 [Goldberg and Robson 83] and Trellis/Owl [Schaffert *et al.* 86], inheritance is used to construct a class hierarchy, where all classes share a root class, and all types are represented as classes. Hence, objects represent all resources so that there is a uniform way of accessing and managing objects.

For a good set of guidelines on the use of types, inheritance and the construction of class hierarchies using an object-oriented language, the interested reader is referred to [Halbert and O'Brien 87].

2.6 Arjuna

Earlier sections of this chapter have described the distributed environment in which a computation must operate. This section expands on this description to detail the tools provided by Arjuna for constructing a reliable distributed computation by briefly describing how objects may be recoverable, persistent and provide concurrency control, and how such objects are controlled by an enclosing atomic action. These descriptions introduce the research presented in this thesis, which will be described in greater depth in the next three chapters.

The basic physical components of the distributed environment consist of a number of workstations, each running the UNIX operating system [Ritchie and Thompson 78], connected together by an Ethernet local area network [Metcalfe and Boggs 76]. Each application program constructed using the tools provided by Arjuna executes in this distributed environment.

The tools Arjuna provides are all constructed using the language C++, and consist of a stub generator and a number of C++ classes. The classes support the construction of user-defined classes, instances of which are recoverable [Dixon and Shrivastava 87], lockable [Parrington and Shrivastava 88], and persistent [Dixon *et al.* 87] (such objects will be termed *Arjuna objects* in the following discussion). The ability to declare nested atomic actions in an application is supported by another Arjuna-provided class. The remainder of this section will briefly describe each of these components, beginning with stub generator.

To provide transparent access to remote objects Arjuna employs *stubs* [Birrel and Nelson 84] which are created using the stub generator [Wheater 88a]. The difference between a purely local program and a distributed program that employs stubs is illustrated in Figure 2.5. An application operates on an interface provided by the class's (local) implementation in the case of a local object

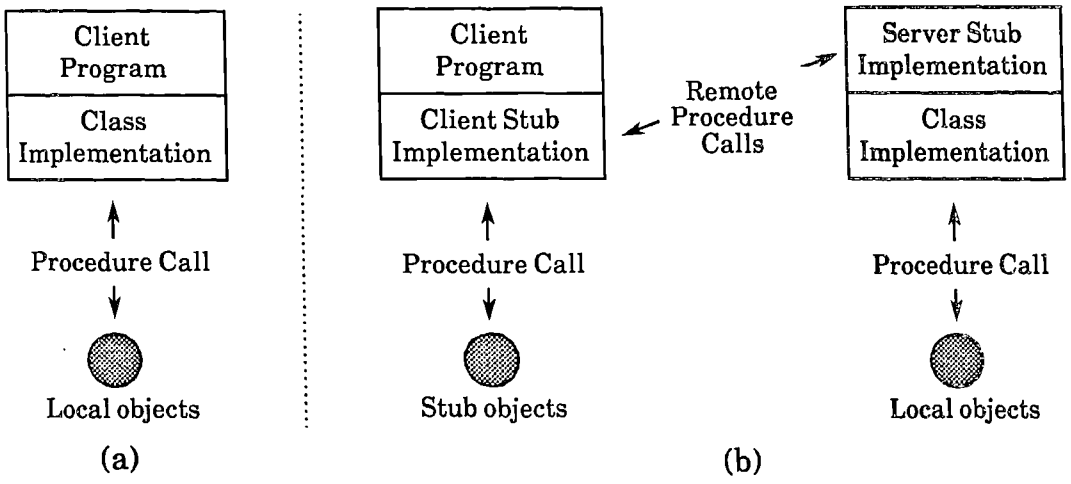

Figure 2.5: Local and distributed programs

(Figure 2.5(a)), whereas it operates on a client stub interface in the case of a remote object (Figure 2.5(b)).

The stubs add an extra level to the hierarchy of layers provided by the remote procedure call system. This communication hierarchy is illustrated in Figure 2.6.

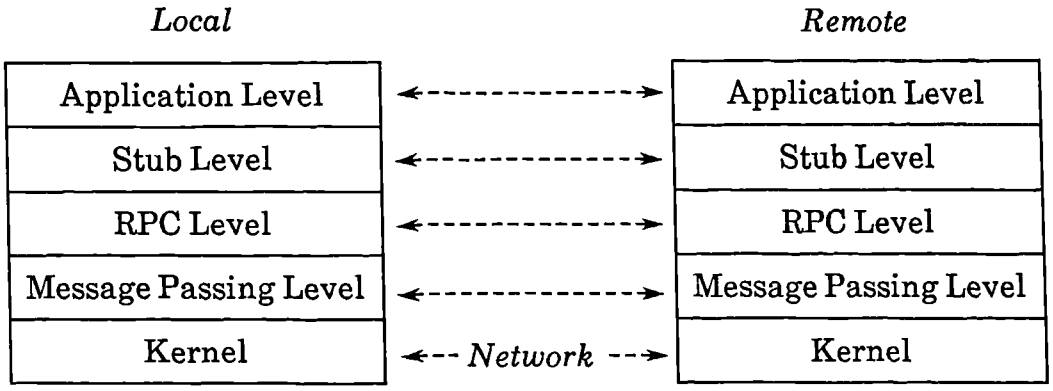

Figure 2.6: The communication architecture

The communication between a client stub object and its server occurs via a remote procedure call mechanism called *Rajdoot* [Panzieri and Shrivastava 88] which includes a mechanism for the detection and termination of orphan computations.

Arjuna provides nested atomic actions for constructing, and structuring, reliable computations. The integrity of an Arjuna object is guaranteed, despite communication failures or node crashes, providing that the operations invoked on the object occur within the scope of an atomic action. Atomic actions are implemented by the class `AtomicAction` which provides a number of operations that are associated with the abstraction of an atomic action. The implementation of the class `AtomicAction` manages a number of sub-components (termed *records*) which are responsible for ensuring that the three properties associated with an atomic action (failure atomicity, serialisability, and permanence of effect) are met. To create atomic actions in an application program, instances of the class `AtomicAction` may be declared. By invoking the operations provided by this class, the computation bounded by the operation which starts the action (`Begin`) and the operation which terminates the action (either `Commit` or `Abort`) will be encapsulated by the resulting atomic action and the outcome of the Arjuna objects accessed within this computation controlled. Multiple instances of the class `AtomicAction`, and the corresponding `Begin` invocations, result in nested atomic actions.

Arjuna objects are the only objects managed by an `AtomicAction` object, and are instances of user-defined classes that have been derived from an Arjuna-provided class called `LockCC`. In this way, functionality common to all objects accessed within an atomic action is provided by exploiting inheritance to inherit common features from the class `LockCC`, and its base class `Object`. Both `LockCC` and `Object` provide operations which must be invoked in the implementation of a user-defined class to coordinate the management of a user-defined object with an executing atomic action implemented by the class `AtomicAction`.

The operation provided by LockCC which must be invoked in the implementation of a user-defined class is called SetLock. As its name suggests, SetLock sets a lock (which is an instance of the class Lock) on the instance of the class and adds information about the lock to the enclosing action. In this way, the concurrency control provided by LockCC operates in conjunction with the atomic action to guarantee the serialisability property. Locks acquired in a nested action are propagated to their parent action when the nested action commits, and are released only if an action aborts or the action committing is the top-level (outermost) atomic action, thereby meeting the two-phase locking rules [Eswaren *et al.* 76] which are needed to guarantee serialisability.

LockCC is itself derived from the class Object which provides the support for maintaining the state of an object. Object also provides an operation (called modified) that will be inherited by a derived class, and which must be invoked to ensure that instances of the class are recoverable and persistent. When this operation is invoked, the old state of the object is saved, and information added to an enclosing action. This information enables the enclosing action to recover the object using the old state, should the action be aborted. In addition, the information is employed by a top-level action to decide which objects should persist by having their current state saved in the object store.

As an example consider the Arjuna class X (illustrated in Figure 2.7) which

```
class X : public LockCC
{
    state of X
public:
    operations provided by X
};
```

Figure 2.7: The Arjuna class X

has been derived from LockCC and invokes the two inherited operations when accessed. Figure 2.8(a) illustrates how the state of an instance of class X will

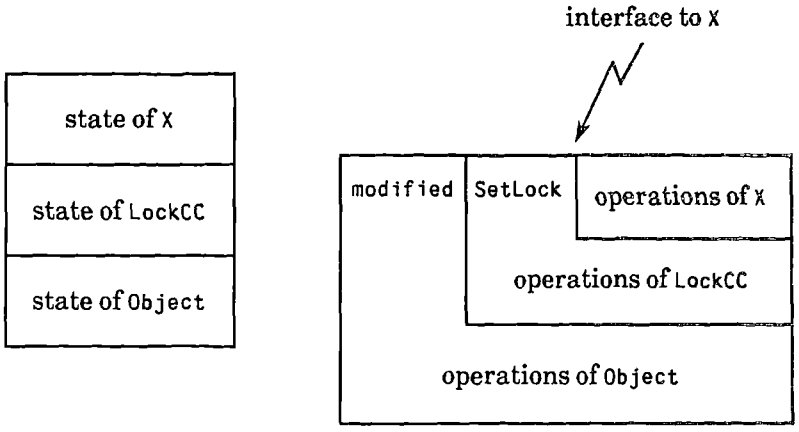

Figure 2.8: The state and interface of an Arjuna class called X

consist of the state inherited from both Object and LockCC, and Figure 2.8(b) shows how the interface to a class is also extended by the operations inherited from LockCC and Object. In this way, each Arjuna object contains extra functionality which assists the atomic action to control the object when it is accessed within the atomic action's scope.

The type of concurrency control provided by the class LockCC, in conjunction with the class that implements the locks (the class Lock), is the single writer/multiple reader approach. It is possible however, to provide higher levels of concurrency by utilising the semantics of a class to refine the operations provided by Lock, thereby providing *type-specific locking*. Since the default locking provided by Lock and type-specific locking may be mixed, the concurrency control provided by Arjuna can be very flexible.

Similarly, the object state management provided by the class Object may be overridden to provide alternative forms of recovery control which may be tailored to the object being managed. Such flexibility is the result of the use of inheritance for adding the extra functionality to a class, as with inheritance it is natural to be able to refine existing implementations.

To ensure the permanence of effect property of an atomic action, Arjuna objects are persistent. Each persistent object is stored in its *passive* form in an object store until it is required by a computation. When a computation wishes to use a persistent object, the object is said to become *active* as it moves from the (non-volatile) object store to the (volatile) memory associated with the computation. The control over the movement of the state of an object is split between the object itself and the state management record of the atomic action. The activation of an object is performed by the object, whereas the deactivation is controlled by the atomic action. In this way, the newly established state of a persistent object will only become permanent providing that modifications to the object occurred within the scope of an atomic action (and that the atomic action and its parents commit).

The object store, that manages and locates the state of objects which are instances of classes that are ultimately derived from the class `Object`, is implemented by the class `ObjectStore`, which is itself ultimately derived from `Object`. Since the class `AtomicAction` is also derived from `Object`, the state of instances of `AtomicAction` may also be saved in the object store.

To locate the persistent state of an object in the object store so that the object may be activated requires a means of uniquely naming the object, and mapping this unique name to the non-volatile state of the object in the object store. Since all objects held in the object store will be instances of a class which has the class `Object` as its root class, a common means of naming is provided by this root class. An instance of the class `Uid` is declared as a part of the state of the class `Object`. `Uid` provides the abstraction of a unique identifier, and is implemented using a method which creates a value based upon the host identifier of the node where the instance is created and the time of the instance's creation. Class `Object` provides

operations to access the value of the `Uid` instance contained in the state provided by `Object`.

An operation is provided by the class `Object` for activating a persistent object. This operation (called `activate`) employs the value of the `Uid` to locate the persistent state in the object store, so that to activate an existing persistent object the `Uid` value for the object must be known. To avoid having to operate at the level of unique identifiers, higher level naming schemes are available (such as strings) with a name server mapping the higher-level name to the `Uid` value.

Since the class `AtomicAction` is also derived from `Object`, atomic actions may be named and accessed in the same way as any other object in the system. A single instance of the class `AtomicAction` is an atomic action, and multiple instances may be created to produce nested atomic actions (nesting occurring when the action is begun). Each new nested atomic action maintains a reference to its parent (containing) action when it is created, thereby enabling the action hierarchy to be traced by executing an operation provided by `AtomicAction` that returns this reference.

This concise description of the features of Arjuna will be expanded in the following chapters as the design and implementation of the mechanisms that support atomic actions, and manage the state of an object, are described. The class `Object` which supports the management of the state of an Arjuna object, the object store, and the class `AtomicAction` which provides the abstraction of an atomic action are all part of the research described in the following chapters. The class `LockCC` which provides concurrency control is the work of Parrington [Parrington 88] and is only briefly discussed in later chapters. To provide a comparison of the features of Arjuna with existing approaches to a reliable distributed programming system, the following section reviews a number of projects and ends by briefly comparing the important features of each approach.

2.7 A review of reliable distributed object-based systems

There are, and have been, a number of research groups occupied with constructing a distributed programming system that address the same issues as those addressed by Arjuna. This sections briefly reviews these systems and ends by comparing each approach with that taken by Arjuna.

2.7.1 Argus

Argus [Liskov 84, Liskov *et al.* 87, Liskov 88] is a reliable distributed programming language, based on the language CLU [Liskov *et al.* 81], which provides support for nested atomic actions. In Argus, a distributed program consists of a collection of operations on *guardians* [Liskov and Scheifler 83], which are stable, crash resistant object managers that provide various services. Each guardian provides a set of *handlers* which constitute the public interface to the objects it manages. Each handler invocation creates a new process and nested atomic action to manage the call.

To support atomicity Argus provides *atomic data types* [Weihl and Liskov 85], instances of which are both serialisable and recoverable. In addition, the instances of atomic data types, termed *atomic objects*, are recoverable and persistent as their state is (only) saved on stable storage when a top-level action that has modified the object commits. The language provides a number of built-in atomic data types and constructs [Weihl 84] which enable user-defined atomic data types to be constructed. The built-in atomic types employ the multiple reader/single writer policy for locking, whereas user-defined atomic types can implement type-specific concurrency control to provide increased concurrency.

2.7.2 Clouds

Clouds [Allchin and McKendry 83] supports robust distributed applications which consist of objects and actions using an approach based on an operating system kernel [Spafford 86] and associated language (*Aeolus* [LeBlanc and Wilkes 85]). The objects which Clouds provides are location independent and may be controlled using nested atomic actions, the implementation [Kenley 86] of which is based on work by Allchin [Allchin 83].

The language *Aelous* provides support for objects, but not inheritance, and uses the support provided by the Clouds kernel to facilitate the construction of reliable distributed applications. Objects are specified in Aelous using an *object definition part* which describes the internal state, operations, and *class* of an object. In Aelous the term *class* is used to define the form of object management that objects of a particular type require. A variety of different *classes* are provided, enabling instances to be managed by the Aeolus run-time system, the Clouds kernel, or action event handlers provided by the implementor of the type.

To guarantee serialisability, locks may be defined by the implementor of a type or generated by the compiler for the language. For the compiler to generate locks, the keyword *autosynch* is required in the object definition part along with the keywords *modifies* and *examines* in the relevant operation declarations.

2.7.3 Profemo

The environment provided by Profemo [Nett *et al.* 85, Nett *et al.* 86] consists of an object-oriented language and operating system executing on specially designed hardware. The hardware provides support for the management of an object's state, and includes an implementation of stable storage for the long-term storage of objects.

The language supports type inheritance for constructing new types, and the system reflects the type hierarchy using instances of a *type object* which maintain the compiled operation code for the objects the type object represents. All objects are named using a unique identifier and located using the *Global Object Manager*, thereby providing system-wide transparent access. In addition, the Profemo object model makes a distinction between objects which may be shared, and therefore require concurrency control, and local objects which may not be shared and are purely local to an object or program. Concurrency control takes the form of locking with the conventional multiple reader/single writer rules.

An interesting aspect of the action management in Profemo is the separation of the successful completion of an action from its commitment, so that an action may have completed but its effects are not made permanent allowing the possibility of an abort. Consequently the system does not explicitly prevent cascading aborts (the *domino effect* [Randell 75]) since objects accessed by a completed action are released when it has completed rather than committed, thereby allowing greater concurrency. To ensure that failure atomicity is possible, the dependencies that may arise from the use of uncommitted objects are maintained in a *recovery graph* so that, if necessary, a global recovery line (the point at which there is no dependency between concurrent computations) can be found using a *chase protocol* [Merlin and Randell 78] to return the system to a consistent state.

2.7.4 Camelot/Avalon

The Camelot system [Spector *et al.* 87] is the latest undertaking from the team which developed TABS [Spector *et al.* 85]. Camelot is a general purpose system which supports nested atomic actions operating in a distributed environment. An object-oriented programming environment is being provided by the Avalon project [Herlihy and Wing 86] which is employing linguistic constructs, in the

form of extensions to languages such as C++, on top of the facilities provided by Camelot to enable atomic actions to be employed in an application.

Camelot relies on the support provided by the Mach [Jones and Rashid 86] operating system for inter-node communication and an interface specification language and compiler called *Matchmaker* [Jones and Rashid 86]. Objects are maintained by *data servers* which can accept multiple requests for service. Each object is named by the *port* of the object's data server, and a logical object identifier that identifies the object in that data server. Atomic action commit information and object modification records are written to a log which is implemented using stable storage techniques. Such information is only written when a top-level action commits.

The approach adopted by Avalon [Detlefs *et al.* 87] to the construction of objects that operate within the atomic action environment supported by Camelot is clearly influenced by Arjuna. The support for synchronisation and recovery is provided by a number of classes (called `resilient`, `atomic`, and `dynamic`), so that new classes may be derived which inherit the basic functionality in a similar manner to the Arjuna classes `Object` and `LockCC`. The Avalon approach only differs from Arjuna in that Avalon classes may use the underlying support provided by Camelot.

2.7.5 Other related projects

There have been various other systems which address a number, if not all, of the issues Arjuna is addressing. This section briefly describes these systems.

The Eden project [Almes *et al.* 85, Black 85] consists of an object based programming system which, although not directly supporting atomic actions, provides mechanisms which assist in the construction of reliable distributed applications. Objects defined in the Eden programming language (called *Ejects*)

have the ability to checkpoint their state and are kept active between accesses by applications. The successor to Eden is Emerald [Black *et al.* 86, Black *et al.* 87], an object-based language for constructing distributed applications. The aim of the Emerald project is to simplify the programming of distributed applications through language support rather than address reliability issues.

The ISIS project [Birman 86] was primarily concerned with providing reliability and availability through replication. By replicating objects on different nodes in the network, the unavailability of an object due to the crash of the node where the object is located can be tolerated, since the inaccessible object can be replaced by a replica which has been kept in synchronisation with the unavailable object. The system also supports nested atomic actions, and makes extensive use of broadcast primitives to maintain consistency between the replicated objects. The successor to ISIS is $ISIS_2$ [Birman and Joseph 87], a toolkit for distributed programming.

There is growing interest in persistent programming, and perhaps the first example was the language PS-Algol [Atkinson *et al.* 83b] which added persistence to an existing language S-Algol [Morrison 79]. A number of operations have been introduced into the language to enable all data objects to persist. In particular, functions are provided to open and close a database (the designer's term for an object store) in which a persistent object is to be explicitly saved. Objects are not recorded in the database until a commit procedure is invoked. The transfer of an object's state is managed by the Persistent Object System (POMS) [Cockshot *et al.* 84]. When the run-time system copies an object from the program's heap to the database, any pointers in the object are converted into the persistent identifiers (pids) of the objects referenced. When an object is restored from the database any pids are left in the object, with attempts to de-reference a pid causing a trap into POMS which automatically restores the referenced object from the database.

The successor to PS-Algol is a new programming language called Napier [Atkinson and Morrison 87] which supports persistence as a part of the language rather than through functional extensions (as with PS-Algol). Persistence is also being supported in distributed systems such as Guide [Balter *et al.* 88], an object-oriented operating system that provides nested atomic actions. Another project that has constructed a new operating system which supports objects is the COSMOS project [Blair *et al.* 86, Nicol *et al.* 87]. The COSMOS kernel includes a database which provides storage and version management for objects, and support for single level atomic actions. An alternative approach to persistence is to provide hardware support, illustrated by the REKURSIV [Harland *et al.* 86] architecture which supports an object store called OBJEKT [Harland and Beloff 87]. OBJEKT is a single level storage system that effectively provides a large object memory, automatically moving objects between volatile and non-volatile storage in response to demands on the system.

The number of projects which are beginning to address the areas of research that the Arjuna project has been addressing, continues to grow rapidly as the importance of constructing reliable object-oriented distributed applications becomes apparent. The next section compares the approach taken by the Arjuna project with the projects reviewed in greater length at the start of this section.

2.7.6 Comparison with Arjuna

Supporting atomic actions which operate on objects in a distributed environment can be approached in a number of ways, as the above reviews show. Extending or defining a new language is one approach taken by a project such as Argus. Alternatively, a new operating system can be constructed, with a language relying on the underlying support it provides, in the manner of the Clouds project. Arjuna, however, has taken a different approach. Since an object-oriented language is employed, Arjuna provides the facilities required to support

atomic actions by employing the features of the language to effectively extend the language without having to modify it. Since no direct modifications are made to the language, and the support for objects controlled by an atomic action is provided through inheritance, this approach is very flexible because it allows various techniques for access and state control to be employed and operate in parallel with each other. For example, the state of an object may consist of a number of objects each providing their own form of recovery or concurrency control. One object may rely on the default locking scheme provided by the `Lock` class, whereas another may use a special lock class that provides locking specific to the semantics of the class of the object.

Another difference between the projects reviewed is each project's approach to object management. The difference lies in the lifetime of the process which manages an object. In Argus, for example, an object may only be accessed by invoking the handler calls provide by the object's guardian. Each guardian exists until it is explicitly deleted. This approach may be contrasted with that employed by Arjuna (and Clouds), where an object is activated by a server process which was created for the object, with the server existing only as long as there is a client for the object. When the last client terminates, the server also terminates. Hence, the process managing an object is only active when it is required, thereby consuming less system resources. A good example of the resources required by an Argus type approach to object management, is the requirement for at least 8 megabytes of memory to run Camelot release (0.98) [Spector 88]. The disadvantage with the Arjuna approach is that the first operation invoked on an object will take longer than subsequent invocations, as the server must be created and the object activated. Another disadvantage is that method employed to manage the state of an object in non-volatile storage is of greater importance, since inefficient storage will also affect the time taken to activate an object. For object managers, such as the guardians provided by Argus or the data servers

provided by Camelot, the state of an object held in non-volatile storage is only required when the node crashes and the server process is recreated, so the way that an object's state is stored in non-volatile storage is less critical.

The level of support for atomic actions and the control over their use, varies from project to project. For instance, in an application constructed using Argus all remote invocations occur within an implicity provided nested atomic action. Such atomic actions are in addition to any explicit declarations of atomic actions the implementor of the application may provide. In Arjuna, there is no implicit declaration of a nested atomic action as part of a remote procedure call. An atomic action must be explicitly declared in an application, but may also be declared in the implementation of a class to ensure that operations on the class occur within the scope of an atomic action. The implementor of a class or application has complete control over any atomic actions they may declare.

Of the systems reviewed, Clouds is perhaps the most uniform in its use of the object paradigm for modelling and naming resources. Arjuna, however, has taken a completely uniform object-oriented approach to all resources associated with the provision of support for atomic actions. State management information is maintained as objects in Arjuna, locks are objects, and even atomic action are objects. Such an approach may be contrasted with a system such as Profemo where almost all resources are objects, yet the state of an object, required to ensure the object is recoverable, is not maintained as an object and as a result cannot be saved in the object store provided for all other objects.

2.8 Concluding Remarks

The aim of this chapter was to describe the environment in which a distributed application may be constructed. In the first half of this chapter, this took the form of describing the types of fault that are assumed to occur in a distributed computation by defining the fault models for the two main components of the distributed environment: the communication system and the node. Given these models, the techniques that may be employed to maintain the consistency of the system in the face of node crashes and communication failures were presented. In addition, the method used to model resources, the object-oriented paradigm, was described, and issues involving its use discussed, using examples written in the implementation language chosen by Arjuna.

Later sections of this chapter described how the Arjuna project supports atomic actions operating on objects in a distributed environment. After this description, a number of related projects which are addressing the same issues were briefly reviewed. The approaches taken by these projects were contrasted with the approach taken by Arjuna.

This chapter included a brief description of the way in which support for nested atomic actions is provided. The next chapter expands on this description, concentrating on the issues involved in designing and implementing the support for atomic actions in a non-distributed environment. The design of a distributed atomic action is also briefly presented. The next chapter assumes that the objects accessed within the scope of an atomic actions are recoverable, persistent, and provide concurrency control. The methods used to ensure an object can be recoverable and persistent will be described in the chapters four and five.

Chapter 3

Constructing Atomic Actions

An atomic action is a control abstraction that may be used to guarantee the integrity of the system state modified by the computation it encapsulates, and may be characterised by the three properties it exhibits: failure atomicity, serialisability, and permanence of effect. This chapter discusses the issues involved in implementing atomic actions which provide the above three properties in a distributed system. In addition to this implementation level view of atomic actions, this chapter also discusses how atomic actions may be used to structure and control the computations that make up an application.

A description of the distributed system model employed throughout this thesis was given in the last chapter. In this model, the resources provided by the distributed system are modelled as objects, so that the system state consists of a collection of objects. An atomic action must therefore manage the objects a computation may access to provide the properties associated with an atomic action. Since all resources are objects, the following discussions will assume that atomic actions are also objects, and will employ the object-oriented terminology defined in the last chapter when discussing the operation of an atomic action.

This chapter begins by describing the atomic action model, assumed in this thesis, and the sequence of events that occur during the execution of an atomic action. The following section then discusses the functionality needed to provide the three properties associated with an atomic action. Given the model and functionality required to provide the abstraction of atomic actions, the middle sections of the chapter discuss the design and implementation of atomic actions which manage objects that are local to the application in which the atomic action

is declared. After the description of a non-distributed atomic action implementation, and issues such as commitment and crash recovery, the following section describes how the implementation may be extended to enable atomic actions to manage remote objects. The final sections describe how an atomic action may be used in an application and how the atomic action boundaries may be enforced.

3.1 The atomic action model

In the model described in this section, an atomic action is assumed to be a passive entity that controls the outcome of the computation it encapsulates. When discussing atomic actions however, phrases such as *running* or *executing* may be employed for convenience. In addition, when the term *computation* is used in the context of an atomic action, the term refers to the computation encapsulated by the atomic action unless explicitly stated otherwise.

Each computation accesses resources which are modelled as objects. An object may be newly created by the computation or may be an existing (persistent) object. A persistent object is normally maintained in a passive state (in an object store) until it is required by a computation, at which point it is activated by being copied into the volatile storage associated with the computation. As a result it is assumed that a computation only manipulates the volatile state of an object (this restriction will be lifted in the next chapter when the discussion moves on to constructing recoverable objects).

There are assumed to be three primitive operations which may be used to declare and control an atomic action, these are: *begin*, *commit*, and *abort*. The *begin* operation starts an atomic action that may be terminated successfully by the *commit* operation. Using the terminology defined in [Anderson *et al.* 78] to describe recovery, the *begin* operation *establishes* a *recovery point* and the *commit* operation *discards* a previously established recovery point. The resulting region

between the *begin* and *commit* operations (which is the computation) is termed a *recovery region*. Hence, a recovery region corresponds to an atomic action.

An atomic action provides automatic backward error recovery through the *abort* operation which restores the objects modified by the computation between the *begin* and *abort* operations to the state held at the beginning of the atomic action. To provide this capability, management information must be recorded with the atomic action about the objects accessed during the execution of the computation. Once the *commit* operation is invoked, this management information will be discarded in such a way that it will not be possible to recover the state of the objects to the state they held at the beginning of that atomic action.

In this model, atomic actions may be nested within other atomic actions, and may be represented using an *action diagram* such as that shown in Figure 3.1. In

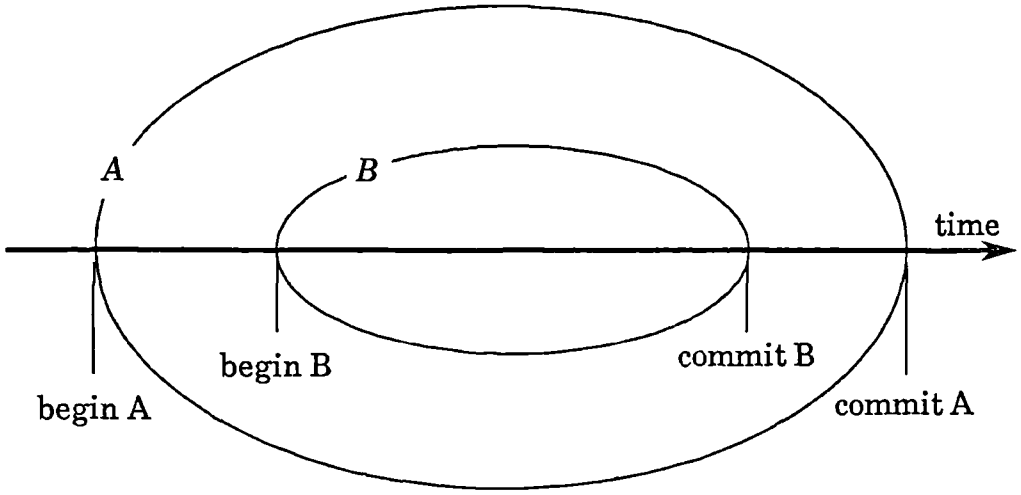

Figure 3.1: Nested atomic actions

this figure an atomic action called *B* is nested within an *outermost* or *top-level* atomic action called *A*. It is assumed that only one atomic action can be active at any one time in an application, so that nested atomic actions may not overlap.

Once a nested atomic action (such as *B*) terminates, control passes to the parent or *containing* action (*A* in the above example).

The boundaries of an atomic action define the *scope* of the atomic action. The scope is considered to be the region bounded by the *begin* and corresponding *commit* operations. If during the execution of the computation it becomes necessary to abort the atomic action then the *abort* operation returns control to the point immediately after the *commit* operation (to ensure that the scope of aborted and committed atomic actions are the same).

Given the control structure described above, the execution model of an atomic action may be described. By executing in *isolation to completion*, an atomic action maintains the consistency of the objects accessed by the computation it encapsulates.

The term *"in isolation"* implies that atomic actions may execute in parallel with other atomic actions which require the same object, but the implementation guarantees that the concurrent atomic actions appear to execute, and access the shared objects, in a serial execution order (i.e. one after another). This capability is provided through the serialisability property exhibited by an atomic action, and requires concurrency control techniques to be employed when shared objects are accessed by concurrent computations.

The term *"to completion"* refers to the failure atomicity and permanence of effect properties of an atomic action. Failure atomicity guarantees that either all or none of the modifications to the objects are made, and permanence of effect ensures that once a modified object is made permanent, the state of the object is guaranteed to remain in a consistent state despite subsequent system failures. If an atomic action is unable to execute to completion then the previous consistent state of all objects that have been modified must be restored. Recovery is

supported by the failure atomicity property and requires the objects to be recoverable.

If an atomic action is interrupted, for example as a result of the crash of the node where the atomic action is sited, then the intermediate object states established by the computation, and the management information required by the atomic action, are all assumed to be lost as they are maintained in volatile storage. The result therefore, is the termination of the atomic action, and the effective recovery of the system to the state held at the beginning of the atomic action. During the commitment of an atomic action, a *commit protocol* is required to achieve the atomicity of the commit operation despite such node crashes or communication failures.

The commit protocol assumed by the following discussions is the well known *two-phase commit protocol* [Gray 78]. During the first phase of this protocol, an attempt is made to place the system in a state which will be unaffected by subsequent node crashes. If the successful completion of first phase is not possible, due for instance to a remote node crash, then the atomic action must be aborted and the system state recovered to that held at the beginning of the action. If the first phase succeeds then the second stage may proceed to make permanent the modified system state.

To avoid inconsistencies during the update of permanent state the commit protocol must maintain management information in non-volatile storage so that a node crash during either phase can be tolerated and a consistent state produced. As soon as a node recovers from a crash, a crash recovery mechanism is assumed to execute and by utilising the management information held in non-volatile storage, the consistency of the modified objects may be established. When a nested atomic action commits it is not necessary to maintain the management information required by the protocol in non-volatile storage since a node failure

will result in the abortion of the atomic action as the volatile state of the system is lost.

As an aid to the discussion of the execution of an atomic action, the sequence of events that occur as each of the three operations is invoked can be considered in terms of a series of action events and changes in the state of an action. The next sub-section defines these events.

3.2.1 Atomic action events

In effect an atomic action has seven events associated with the three basic operations (illustrated in Figure 3.2). The *begin* and *abort* operations each result

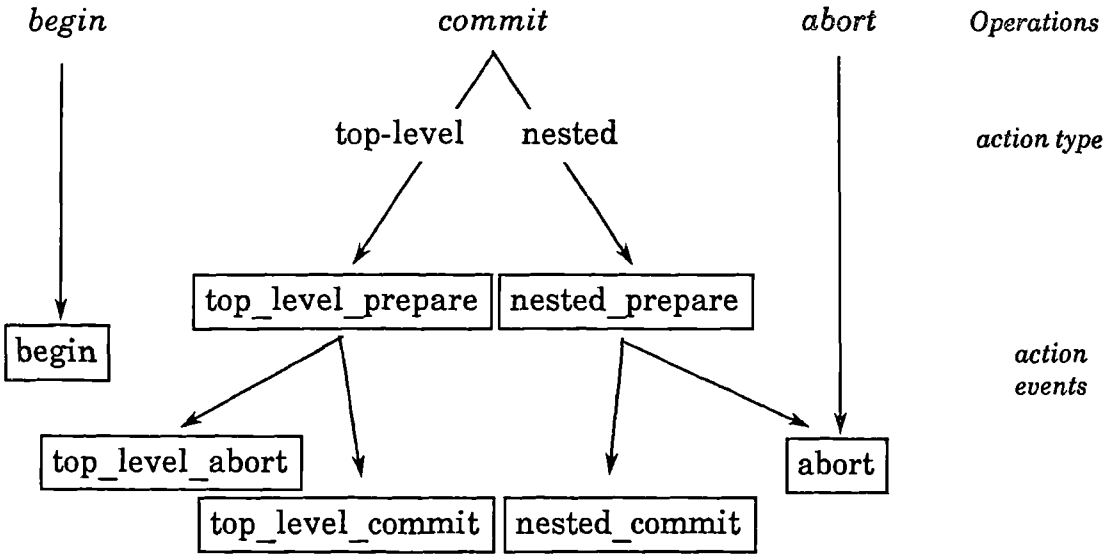

Figure 3.2: Action events

in a single event (*begin* and *abort*), whereas the *commit* operation produces two events due to the use of a two-phase commit protocol.

The events that occur during the commit operation depend upon whether the action is nested or top-level. In the first phase of the commit protocol, either a *nested_prepare* or *top_level_prepare* event occurs. If the prepare phase fails, the

corresponding abort event occurs. If the prepare phase succeeds, the corresponding commit event occurs.

Each event is associated with a change in the state of the atomic action, and not all changes, and hence event sequences, are valid. Figure 3.3 illustrates the

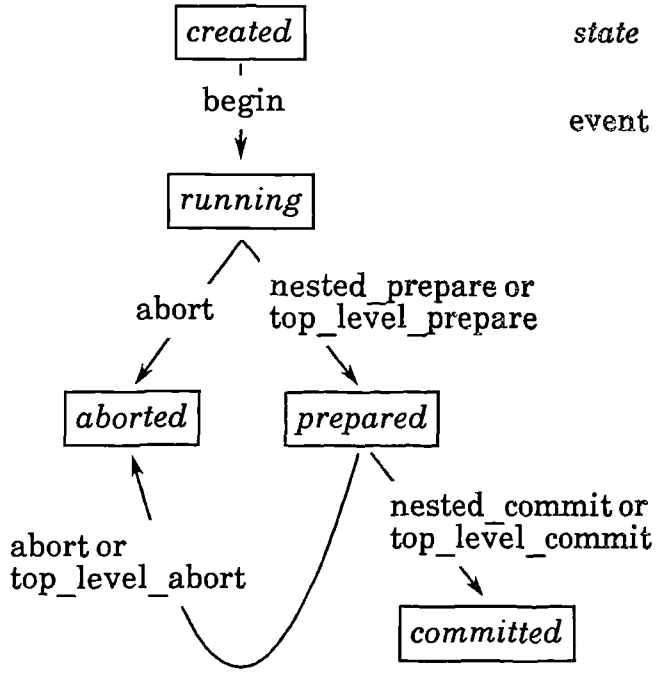

Figure 3.3: Atomic action states and events

allowable state changes and events. Any operation invocation which results in an action state and event sequence that is different to that illustrated in Figure 3.3 is invalid. For example, invoking a *abort* operation on an action whose status is not *running* (i.e. which has not had a *begin* invocation or which has already been committed) would attempt to move an action from its state of *created* to *aborted.* Since this is not in the above figure it is an invalid transition, and should not be allowed by an atomic action implementation. The properties provided by an atomic action are only available to objects if the action's status is *running*, and are therefore defined by the region between *begin* and either *commit* or *abort* invocations.

In addition to the final states described above, there are a number of intermediate states which reflect the type of operation invoked on an atomic action. When the *abort* operation is invoked, the state of the atomic action changes to *aborting* and holds this state until the *abort* operation terminates (at which point the state is changed to *aborted*). Similarly, the *prepare* operation results in the intermediate state *preparing*, and the *commit* operation the intermediate state *committing*.

3.2 The functionality required by an atomic action

Before discussing the issues involved in implementing atomic actions, the functionality required to provide the properties associated with an atomic action during the execution of an application that contains atomic actions must be considered. This is the purpose of this section. Since an atomic action is responsible for controlling the system state, and this state is represented by objects, it therefore follows that the abstraction of an atomic action must manage the objects accessed within its scope to provide these three properties.

When an object is modified within the scope of an atomic action, information about the changes made to the object must be maintained by the atomic action to enable the object to be recovered and thereby satisfy the failure atomicity property. Similarly, modifications or access to a shared object must involve some form of concurrency control on the shared object, with the addition of the corresponding management information to the controlling atomic action. The management information maintained about the concurrency control should enable it to be removed from an object when a top-level atomic action commits (to guarantee two-phase locking [Eswaran *et al.* 76]) or when the atomic action, within which the concurrency control was applied, is aborted.

To illustrate how management information is generated and maintained by an atomic action about the objects accessed within the computation it encapsulates, a modified form of the action diagram notation used in Figure 3.1 is employed. The modifications involve removing the time axis and indications of scope which result from the use of the atomic action operation names.

Consider an application containing two atomic actions (one nested within the other in the manner of Figure 3.1) illustrated in Figure 3.4. In Figure 3.4, the

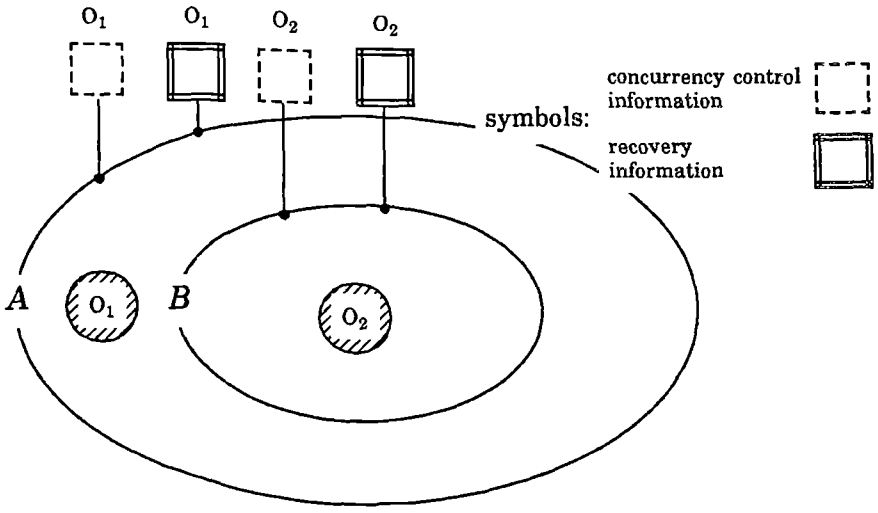

Figure 3.4: Nested atomic actions and management information

nested atomic action *B* is the active atomic action and has modified the object O_2. Before the modification took place, concurrency control was applied to O_2 to ensure that the action *B* had exclusive access to O_2. During the application of the concurrency control, concurrency control management information was added to the atomic action *B*. Once concurrency control was applied, the object was modified and recovery information added. Before the atomic action *B* was created, the parent action of *B* (*A*) modified an object O_1 which resulted in recovery and concurrency control information being maintained by *A* about O_1. Figure 3.4 illustrates what information is created and by which action it is maintained.

When a nested atomic action commits, the recovery and concurrency control information maintained by the nested atomic action must be passed to (or *merged* into) its parent since the system state established by the nested atomic action will need to be recovered, and concurrency control released, should the parent atomic action abort. This will not be possible unless the parent atomic action has the management information for the objects accessed by the computation encapsulated by the nested atomic action. Figure 3.5 illustrates how the

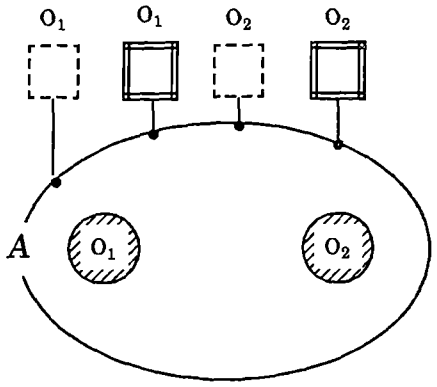

Figure 3.5: After the commitment of the atomic action *B*

management information maintained by the atomic action *B* has been passed to its parent during the commit operation.

Since the outcome of a nested atomic action is dependent upon its parent, the permanence of effect property need only be associated with a top-level atomic action. The information about the modified objects held by a top-level action must therefore be sufficient to enable the new state of those objects to be made permanent. In the example illustrated in Figure 3.5, it is the management information maintained about the objects O_1 and O_2 that indicate these objects should be made permanent if the atomic action *A* commits. During the commitment of the atomic action *A*, a commit protocol is required to ensure that a node crash during the commit operation either results in both O_1 and O_2, or neither object, being made permanent.

This section has described the basic functionality required by an atomic action. To summarise, an atomic action must maintain information about the objects accessed during the execution of the computation it encapsulates. The information must be sufficient to enable the objects to be recovered to meet the failure atomicity property, and any concurrency control applied to meet the serialisability property released at the appropriate time. In addition, node crashes during the commitment of a top-level atomic action must be tolerated so that a consistent system state is established.

Given this functionality, the following section describes an approach to implementing distributed atomic actions. The approach is to implement a non-distributed atomic action and extend this implementation to a distributed environment. The distributed atomic action described in the next section is therefore only a design, whereas the non-distributed atomic action design has been implemented and extensively tested.

3.3 Constructing distributed atomic actions

This section discusses the issues involved in providing atomic actions which may be used in applications that access both local and remote objects, starting with a discussion of how the support for atomic actions may be provided. One approach is to produce a programming language that includes syntactic extensions which correspond to atomic action declarations by either defining a new, or modifying an existing, programming language. The resulting atomic action syntax may then be used by the language's compiler to generate the necessary support for atomic actions. An alternative approach is to modify the underlying operating system to provide a set of system calls which support atomic actions. A third alternative is a mixture of the previous two, where a language provides syntactic constructs but relies on support provided by the underlying operating system. The final alternative which is worth considering does not

require any modifications to either the language or operating system, but provides the necessary support by utilising the existing features of a programming language. The most common approaches has been language extensions which generally rely on additional support from the underlying operating system.

Clearly each implementation approach has its advantages and disadvantages, for instance an approach based on extending a language requires modifications to the compiler which may be non-trivial, yet the resulting atomic action syntax is likely to be integrated into the language and as a result could be the easiest of the approaches to use. The most important differences between all approaches however, is the type of support for atomic actions. For instance, in a language based approach (such as *Aeolus* [LeBlanc and Wilkes 85]) it is common to be able to declare an operation (or procedure) in an application to be atomic, implying that the operation executes as an atomic action and is committed if it terminates normally or is aborted if not. The atomic action boundaries are therefore implicit, whereas in an approach such as that provided by Argus [Liskov 84], special keywords provided by the language may be employed to explicitly define the boundaries of an atomic action. Of equal importance when considering the support for atomic actions is whether the concurrency control and recovery management information (which were described in the last section) are created statically during the compilation of a program written in the language, or are created and added to an atomic action dynamically during the execution of the computation.

The problem with language or operating system approaches is that they are closely associated with a particular language or operating system, and as a result are difficult to generalise to other environments (languages and/or operating systems). If an approach is based on employing the features of a language, and those features are provided by other languages, then such an approach may be

employed in other suitable environments. This is the reason why the work described in this thesis employs a language based approach, and uses a programming paradigm (object-oriented programming) that provides abundant functionality with which to construct the necessary support.

Since a language based approach has been chosen, the resulting atomic actions must be explicitly declared in an application, and the management information must be dynamically added to the atomic action. Given this approach, consideration must be given to how the objects accessed by the computation encapsulated by an atomic action are to be managed, and how to ensure that the objects are suitable for management since support is not available from the compiler for the programming language or the underlying operating system.

If an object is modified then the state of, or changes to, the object must be recorded so that if recovery is required to ensure the failure atomicity property the previous object state may be re-established using the recorded information. When a class is implemented using a language that supports encapsulation, knowledge about changes to the state of an instance will be limited to the implementation of the class. If an atomic action is required to control modifications to the system state implemented by such a class then, since the encapsulation should not be broken, the responsibility for providing recovery must lie with the class itself. Hence, instances of the class should be *recoverable*.

To provide the serialisability property of an action, any objects accessed during the computation encapsulated by an action must be controlled. Since the only means of accessing an object is through the invocation of an operation provided by the class of the object, knowledge about the type of access made by an operation will only be available to the implementation of the class of the object.

Hence, access control must be provided by the class and employed during the operations provided by the class.

When a top-level atomic action successfully terminates, the system state that has been modified by the action (and any nested actions which committed) should become permanent. Since the system state is represented by objects, the new object states must become permanent. Permanent objects are persistent objects which exist beyond the lifetime of the computation that created or used them. Instances of a class therefore must also be *persistent.*

In addition to having the functionality described above, an object must be recorded with an executing atomic action in such a way that the functionality the object provides is invoked, when required, to provide the properties associated with an atomic action. The coordination of these properties may be considered from the viewpoint of the type of management each property requires. To ensure a class is recoverable and persistent, the *state* of instances of the class must be managed. To control access to an instance of a class clearly requires *access* management. Consequently, the management of the three properties may be considered in two distinct stages. One is concerned with managing the state of an object, the other with managing access to an object.

Given this approach, it would be desirable to provide a general mechanism for both access and state management that could deal with the various different types of object and alternative concurrency control techniques. The assumption may be made however that to manage each, a corresponding sub-component (termed a *record*) may be provided. The state management record therefore is used for ensuring the *failure atomicity* and *permanence of effect* properties, and the access management record is used for ensuring the *serialisability* property.

The distributed programming environment in which atomic actions are assumed to operate was described in the last chapter. In this environment, a remote object is accessed through a stub object at the local node, which makes remote procedure calls to the server that manages the remote object at the remote node (see Figure 2.5). To enable an atomic action to manage the remote objects an additional management record may be provided: the distribution management record. Figure 3.6 summarises how an atomic action is assumed to consist of

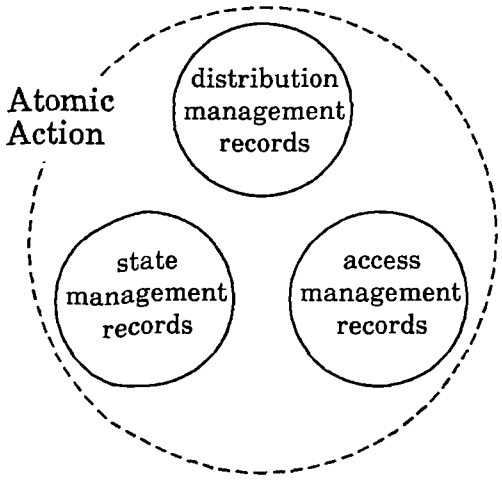

Figure 3.6: The composition of an atomic action

three different types of record, each managing a particular aspect of the control provided by the resulting abstraction in the distributed environment assumed by Arjuna.

When an atomic action is aborted, the state record must ensure that the object which it is managing is recovered, the access record must release the concurrency control maintained on the object, and the distribution record must ensure that the necessary recovery and concurrency control operations take place at the remote node managed by the record. Each record should therefore provide an operation that can be invoked when the action is aborted or committed. To manage the resulting records and provide the abstraction of, and interface to, an atomic

action an additional component is needed. This component is termed the *atomic action subsystem.*

The above description has introduced the atomic action design. The following section describes the design in greater detail, and how the design has been realised.

3.4 The implementation of an atomic action

This section describes how the functionality required to provide the abstraction of an atomic action may be implemented. This implementation is limited to managing objects which are local to the application in which an atomic action is declared, and begins by considering how the atomic action subsystem may be implemented.

The assumption throughout this thesis is that the implementation language is object-oriented, so that a practice of modelling resources as objects may be followed. Since an atomic action can be considered to be a resource, it therefore follows that atomic actions should also be modelled as objects. To illustrate this approach using the language C++, Figure 3.7 illustrates a suitable class which is

```
enum Action_Status  {CREATED,RUNNING,ABORTING,ABORTED,
                     PREPARING,PREPARED,COMMITTING,COMMITTED};

class AtomicAction ....
{
    ....  // private variables and operations
public:
    ....
    void          Begin ();
    Action_Status Commit();
    void          Abort ();
    ....
    Action_Status Status();
    ... add(AbstractRecord*);
};
```

Figure 3.7 : The class `AtomicAction`

called `AtomicAction`. This class is assumed to implement the state transitions defined in section 3.2.1 as each action event occurs. To determine the current

state of an atomic action declared in an application an additional operation called `Status` is provided. Since the `Commit` operation may fail, the status of the atomic action is also returned when this operation is invoked. The `Begin` and `Abort` operations are assumed to always execute correctly, so that they do not return a result (indicated by the `void` declaration in the above class). The remaining operation (`add`) is provided so that the various management records may be added to an `AtomicAction` instance.

Instances of the class `AtomicAction` may be declared in an application, and the three operations provided by the class used to control and utilise the resulting atomic actions. As an example, the skeleton program given in Figure 3.8 shows

```
AtomicAction A, B;
A.Begin();
     // operations on object O1
     B.Begin();
          // operations on object O2
     B.Commit();
A.Commit();
```

Figure 3.8 : Using the class `AtomicAction`

the code necessary to implement the simple example that was illustrated in Figure 3.4.

A class such as `AtomicAction` provides the interface to atomic actions, and manages the various records that in turn manage each of the properties associated with an atomic action. Since the records are also resources they should also be objects, requiring a suitable class for each. As each action event occurs during the lifetime of the atomic action, that event should be reflected in all of the records (with the obvious exception of the *begin* event). Since the set of events is common to all records, a base class can be defined with an operation corresponding to each action event. The records may then be implemented as classes derived from this base class. In each of these derived classes the set of

operations should be refined so that each record class invokes operations specific to the function of that record. In this way the class `AtomicAction` may treat the records as if they are instances of the base class, simply invoking the operations corresponding to the action events as they occur.

An implementation of a suitable base class called `AbstractRecord` is illustrated in Figure 3.9. The name of this class is intended to indicate that the

```
class AbstractRecord ....
{
    AbstractRecord *next;
    AbstractRecord *last;

public:
    AbstractRecord();
    ~AbstractRecord();
    ....

    virtual int  nested_prepare();
    virtual void nested_commit();
    virtual void abort();
    virtual int  top_level_prepare();
    virtual void top_level_commit();
    virtual void top_level_abort();
    ....
};
```

Figure 3.9: The class `AbstractRecord`

class is an *abstract* class (a class created to provide common functionality and not in order to have instances). This class provides facilities for the management of a linked list of instances of itself, with a set of operations that may be invoked at the various action events. Each of the event operations is declared to be a *virtual* operation (a C++ mechanism that ensures dynamic binding of the operation occurs).

When a nested atomic action is committed, the management information (or records) must be passed to the parent atomic action (as described in section 3.2). Depending upon the type of management provided by the record, the record may be added to the parent atomic action, or may replace a similar record (discarding the record in the parent atomic action), or may be discarded if the record in the parent atomic action is more suitable. The responsibility for deciding which action to take belongs to the implementation of the record, so that the

`AbstractRecord` class provides a suitable set of operations (not illustrated in Figure 3.9) which may be invoked by `AtomicAction` to decide what action to take.

To provide atomic actions which operate in a non-distributed environment it is only necessary to provide record classes which manage recovery, concurrency control, and persistence. For distribution, a distribution management record is required. A number of suitable record classes have been defined, and the operation of these classes will now be briefly described (a more complete description of each will appear in the relevant sections of the following chapters).

If the implementor of a class wants to construct a recoverable class which employs state restoration, based on the old state of an object, then they can derive their class from a system provided class called `Object`. The class `Object` provides an operation, which should be invoked before the state of the derived class is modified, that creates and adds an instance of a record class to an instance of `AtomicAction`. This record class, called `ObjectStateRecord`, manages the recovery data required by the class `Object` to enable the user-defined object to be recoverable. For example, when the *current instance of* `AtomicAction` *is* aborted, the implementation of the `abort` operation provided by the `ObjectStateRecord` class invokes the operation provided by `Object` to recover the object, passing the recovery data maintained by the `ObjectStateRecord` instance as an argument.

To manage the persistence of an object, which is an instance of a class derived from `Object`, another record class is required. This record class called `PersistentRecord` is derived from `ObjectStateRecord`, so that a persistent object is also recoverable. The fact that an instance of `PersistentRecord` is recorded in a top-level instance of `AtomicAction` is sufficient to indicate that the object managed by the `PersistentRecord` instance has been modified, and

should therefore persist when the top-level atomic action is committed. The implementation of the top-level action event operations for the PersistentRecord class ensure that the object being managed is saved in an object store to effect persistence.

If the implementor also wants to provide concurrency control for a class, then they can derive their class from another system provided class, called LockCC. The class LockCC provides an operation (SetLock) which may be invoked to set a read or write lock on an object during the invocation of an operation provided by the object. When SetLock is invoked, the implementation of the operation creates and adds an instance of the class LockRecord to the current active instance of AtomicAction. This record class manages the information about the lock set on the object, and ensures that two-phase locking [Eswaran *et al.* 76] occurs by only releasing the lock (using the LockCC operation ReleaseLock) when the top-level AtomicAction commits or the current AtomicAction aborts. The propagation of LockRecord objects from a nested to parent atomic action also ensures that the lock inheritance rules described in [Allchin 83] are met.

The remaining record of importance is the distributed management record. When a computation accesses remote objects, the class ServerActionRecord will be instantiated and added to an AtomicAction to manage the atomic actions which are created by the server for the remote object at the remote node. ServerActionRecord ensures that for each of the action events at the local node, the remote server also has a corresponding operation invoked on the server action which is managing the state of, and access to, the objects at that server. The more detailed description of the function and purpose of this class will be given in a later section which describes the design of a distributed atomic action.

All of the classes described above belong to a class hierarchy that has the class `Object` as the root class. Both `AbstractRecord` and `AtomicAction` are derived from this class so that they can inherit a number of fundamental properties useful to objects. One of the main purposes of the class `Object` is the provision of a common means by which all objects can be named. This is provided by the internal declaration (in `Object`) of an instance of the class `Uid` (which provides the abstraction of a unique identifier), and an operation to read (`get_Uid`) its value. The most important feature of the class `Object`, however, is the operations that it provides for state based recovery and persistence which were briefly described above.

Figure 3.10 illustrates how the record classes described above fit into the class

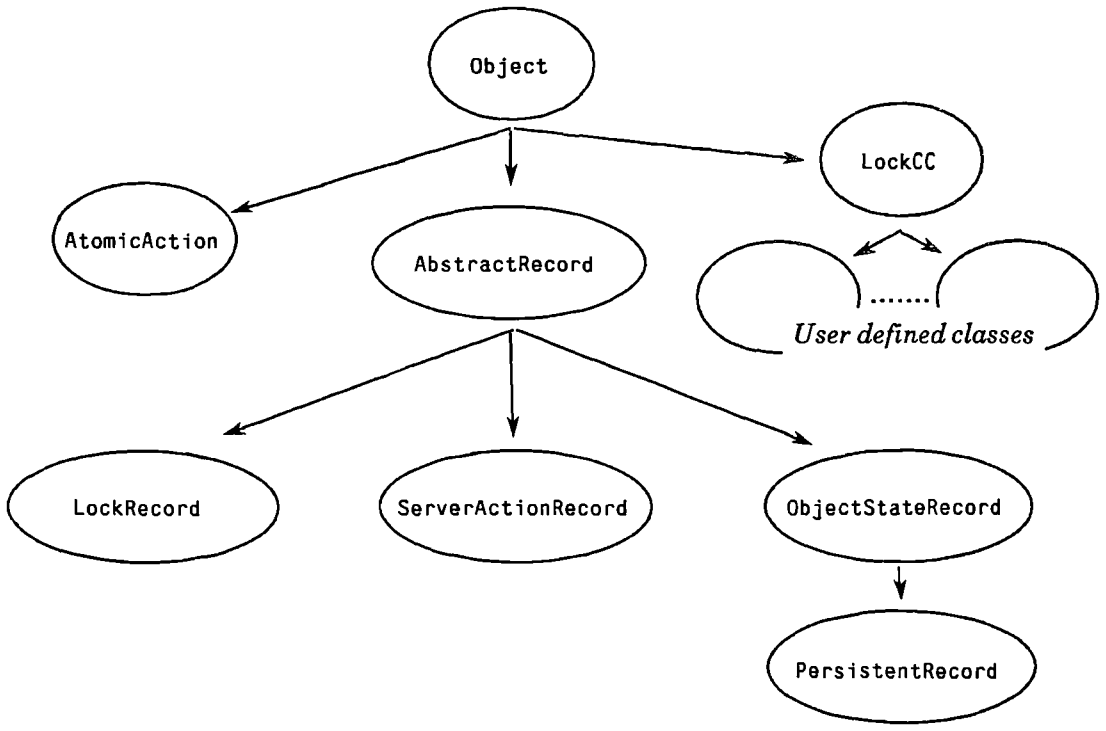

Figure 3.10: The atomic action subsystem class hierarchy

hierarchy. As this figure illustrates, the class `LockCC` is derived from the class `Object`, so that a user-defined class which wants to employ the state restoration

provided by `Object` and locking provided by `LockCC`, need only be derived from the class `LockCC`.

If the implementor of a class wishes to provide alternative forms of recovery and/or concurrency control, they may then create suitable management records by defining the corresponding record classes (using `AbstractRecord`) and ensuring that these record classes are instantiated and added to an `AtomicAction` during the invocation of operations on their class. The manner in which a record class may be declared to provide class-specific management forms part of the discussion on recoverable objects in the next chapter.

This section has described how the class `AtomicAction` manages records which are instances of classes derived from the class `AbstractRecord`. The record classes each manage a property which characterises the behaviour of an atomic action. The next section discusses and illustrates the operation of the `AtomicAction` and record classes using a simple example that employs the record classes described above.

3.5 The operation of the class `AtomicAction`

The main purpose of the class `AtomicAction` is to manage all the records which may be added during a computation's execution, invoking the corresponding record operations as action events occur. For instance, when the `Abort` operation is invoked on an instance of `AtomicAction`, the resulting *abort* event involves the implementation of the `Abort` operation invoking the `AbstractRecord` `abort` operation on all the records held by the `AtomicAction` instance. Since these records will be instances of classes (derived from `AbstractRecord`) that have refined the `abort` operation, this will result in the recovery of, and removal of any concurrency control on, the objects accessed by the computation encapsulated by the atomic action. Similarly, `Commit` invocations result in the corresponding type of commit operation being invoked

on each record. To know what type of commit to invoke, each instance of `AtomicAction` maintains a reference to its parent action. If this reference has no value, then the `AtomicAction` is a top-level action and will invoke the top-level operations provided by the records instead of the nested operations.

To provide a means of discovering whether an atomic action is active, a global variable called `CurrentAtomicAction` is provided and is maintained as a reference to the currently executing atomic action. When the `AtomicAction` `Begin` operation is invoked, the value of this variable is used to determine whether the `AtomicAction` instance is a top-level or nested atomic action. If the variable has a value then the parental reference is set to this value, and the `CurrentAtomicAction` variable set to the new (nested) `AtomicAction`.

As noted previously, the class `AtomicAction` implements the state transitions, and internal states, defined in section 3.2.1. To ensure that incorrect operation sequences do not occur, each operation is checked against the state of the action (based upon the valid transitions described in section 3.2.1). A run-time error occurs if an incorrect sequence takes place, with the severity of the error being definable by a user for each action.

The following sub-section describes a simple example which illustrates how the class `AtomicAction` coordinates the state and access records added by an instance of a class derived from `LockCC`, thereby ensuring the three properties of the action are met. To simplify the description, the computation in this example does not access any remote objects.

3.5.1 A simple example

As an example of the operation of the class `AtomicAction` consider the sequence of operations from an application illustrated in Figure 3.11. In this

```
ExampleClass EC1, EC2;   // with state0
AtomicAction A, B;
....
A.Begin();               // EC1 has state0 at this point
     ....
     EC1.update();       // set a lock and change state to state1
     ....
     B.Begin();          // EC1 has state1 at this point
          ....
          EC1.update();  // change state to state2
          EC2.update();  // change state to state1
          ....
     B.Commit();
     ....
A.Abort();               // state is recovered to state0
....
```

Figure 3.11: An example using an `AtomicAction`

example, two instances of a class called `ExampleClass` (which is assumed to be derived from `LockCC`) have been declared. The state of each instance before the `AtomicAction` A begins may be considered to be *state*$_0$. Inside `AtomicAction` A the object EC_1 is modified so that its state becomes *state*$_1$. Before this modification is allowed to occur, a write lock has been set on the object which guarantees exclusive access to the object. The details of this lock are recorded in an instance of `LockRecord`, and the recovery information (in the form of the old state of the object) is saved in an instance of `ObjectStateRecord`.

The implementation of the `update` operation is shown in Figure 3.12 which

```
void ExampleClass::update()
{
     LockCC::SetLock(new Lock(WRITE));
     Object::modified();

     // now the state can be modified ....
}
```

Figure 3.12: The `ExampleClass update` operation

illustrates the extra operations the implementor of a class must invoke to ensure that the LockRecord and ObjectStateRecord instances are created and added to the currently executing instance of AtomicAction. The LockRecord instance is created and added by the SetLock operation, and the ObjectStateRecord instance is created and added by the modified operation. The objects in existence just before the nested action B begins are illustrated in Figure 3.13.

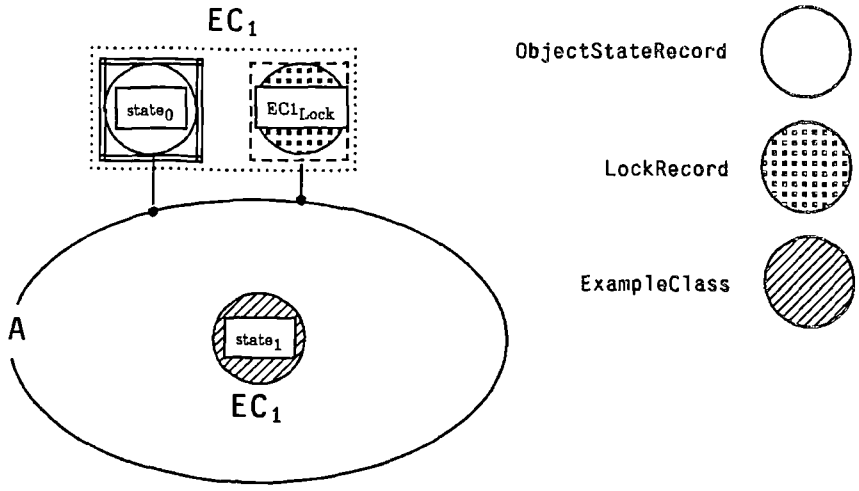

Figure 3.13: Objects created during AtomicAction A

In AtomicAction B, the update operation is invoked on EC_1 so that the state of EC_1 changes from *state1* to *state2*. In the update operation further invocations of SetLock and modified will occur. Since the only other holder of a lock on EC_1 is A (which is the parent of B), the attempt to set another write lock will be allowed and a new LockRecord created to hold the information about the lock added to B. Since the state of EC_1 is modified again, another ObjectStateRecord is created to hold recovery information for EC_1 which contains the *state1*. In addition to the modification to EC_1, another object is modified within the scope of AtomicAction B. This object is EC_2, and its state changes as a result of an invocation of the update operation from *state0* to *state1*, along with the creation of an instance of LockRecord and

`ObjectStateRecord`. The objects in existence just before B commits are illustrated in Figure 3.14.

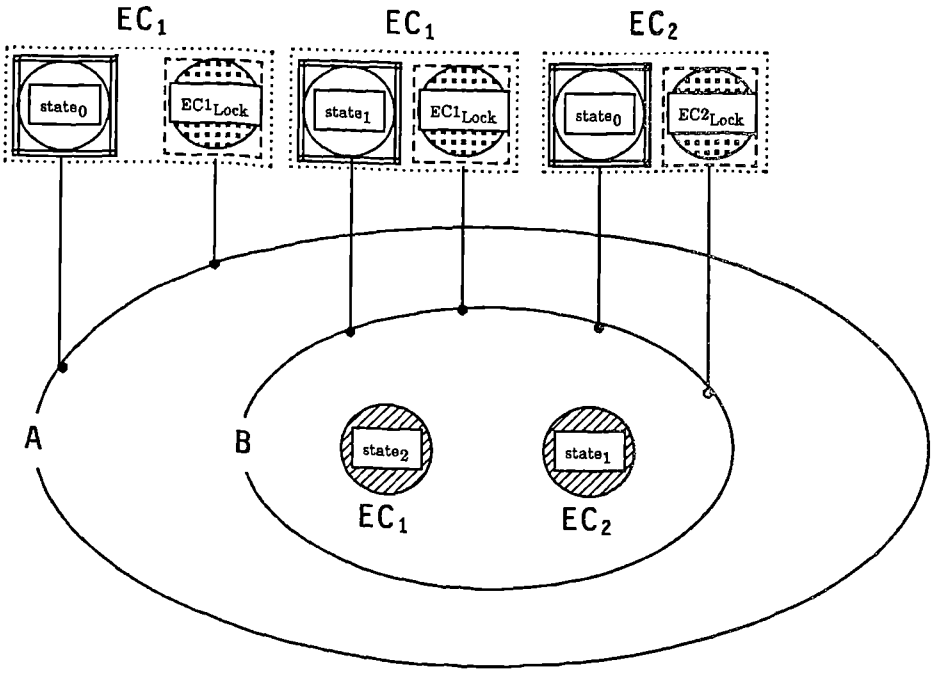

Figure 3.14: Objects created before B commits

When `AtomicAction` B commits, the information contained in B must be merged into its parent action A. The operations provided by `ObjectStateRecord` and `LockRecord` that are invoked by `AtomicAction` to determine what action to take during a merge, return a value which indicates that the most recent version of the same record for an object can be discarded. In the case of an `ObjectStateRecord` object, discarding the newest record object is possible because the recovery supported by `ObjectStateRecord` (and `Object`) relies on the old state of an object which will be held by the parent of a nested atomic action (this technique is an implementation of the recovery cache algorithm [Horning *et al.* 74, Anderson and Kerr 76]). Since A has no record of EC_2, the records for this object will be added to those already held by A, producing the situation illustrated in Figure 3.15.

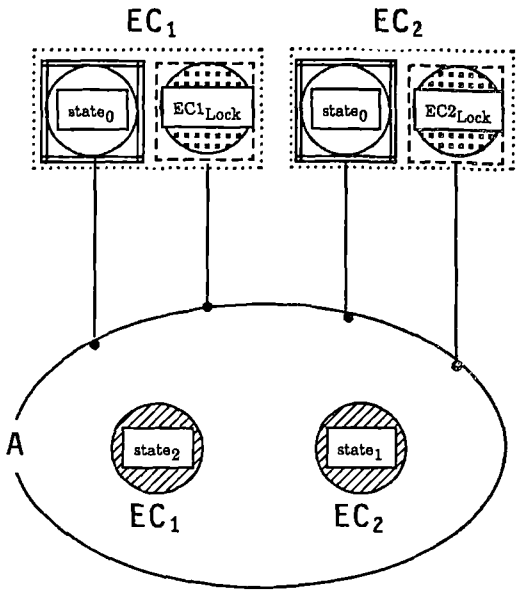

Figure 3.15: Objects in existence after B commits

During the `Abort` operation, the `AtomicAction` A takes the objects it contains (the `LockRecord` and `ObjectStateRecord` instances for EC_1 and EC_2) and invokes the `abort` operation on each. In the `ObjectStateRecord` implementation the old state of each object replaces the current state so that, for example, the state of EC_1 (which was *state*$_2$) is restored to *state*$_0$. Further details of this recovery action are provided in the next chapter. The `LockRecord` implementation employs the details of the lock held on the object to release the lock. Once each record is processed it is deleted.

In this example, if the `AtomicAction` A commits, and the `ExampleClass` objects were declared to be persistent, then the state of the two objects should be made permanent since A is a top-level action. In this case, instances of the class `PersistentRecord` would have been added to the atomic action instead of instances of the class `ObjectStateRecord`. A `PersistentRecord` object operates in an identical manner to an `ObjectStateRecord` object until the top-level commit operation, at which point the new state of each object being

managed is saved in the object store using an operation provided by the class `Object` (called `deactivate`) for this purpose.

The `AtomicAction` implementation also ensures that instances are crash recoverable, and that the two-phase commit protocol is correctly followed. The mechanisms behind crash recovery and commitment are described in the next section.

3.6 Commitment and crash recovery

When a top-level atomic action is to be committed, a commit protocol is required to ensure the atomicity of the commit operation despite the presence of node crashes and communication failures. The commit protocol employed by the `AtomicAction` implementation, and described in this section, is the well known two-phase commit protocol [Gray 78]. During the execution of this protocol, management information must be saved in non-volatile storage to enable a node crash to be tolerated. A node crash before the end of the first phase will result in the commit operation being aborted, leaving any objects modified during an atomic action in the state they held before the atomic action began. If a node crash occurs after the first phase has completed successfully, then protocol management information will be employed by the crash recovery mechanism to mask the node crash and produce the system state that would have been established if the node crash had not occurred.

The object and action model presented in this thesis provides an interesting means of implementing the two-phase commit algorithm and providing crash recovery. Each of the management records which an `AtomicAction` is managing is an instance of a class that is ultimately derived from the class `Object`. Since the class `Object` supports persistence, the persistent operations

may be used to save the management records in non-volatile storage (managed as an object store).

During the first phase of the `Commit` operation, the `AtomicAction` implementation invokes the `deactivate` operation on each management record, thereby saving the state of each record in the object store. After making an individual record permanent, the `top_level_prepare` operation provided by the record may be invoked so that the record can indicate whether the protocol may proceed to the second phase. In the case of the `PersistentRecord` class, the implementation of this operation involves saving the state of the object in the object store. If the object cannot be saved in the object store then a value is returned by the `top_level_prepare` operation which indicates that movement to the second phase should not occur. The *prepare* stage is illustrated in Figure 3.16 using the example given in the last section, assuming that the

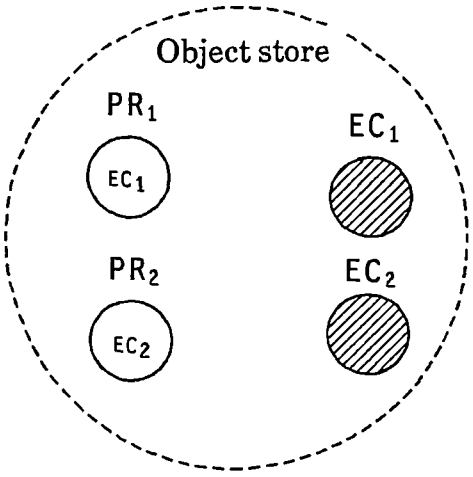

Figure 3.16: During phase one

`AtomicAction` A is committing, that the instances of `ExampleClass` were declared to be persistent, and omitting the `LockRecords` to simplify the example. In the above figure, the two modified objects (EC_1 and EC_2) have been saved in the object store by their respective `PersistentRecords` (labeled PR_1 and PR_2). The state of each `PersistentRecord` consists of the name of each

object the PersistentRecord is managing, hence, PersistentRecord PR_1 has as its state the name of EC_1 and so on.

If this operation succeeds for all records then the first phase has been successfully completed. To indicate this situation, class AtomicAction maintains a private object in which each record's name is saved. This object, which forms part of the state of an instance of AtomicAction, is saved in the object store as the AtomicAction saves itself (the reason for deriving the class AtomicAction from the class Object). This stage is illustrated in Figure 3.17,

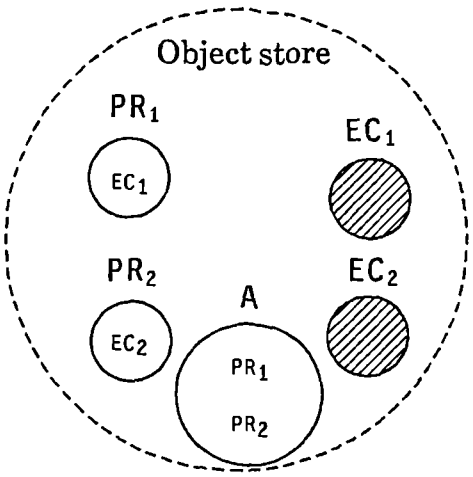

Figure 3.17: At the end of phase one

where the AtomicAction is labeled *A*.

Once the AtomicAction has been saved, the second phase can begin. In this phase each record is processed a second time, with the AtomicAction invoking the top_level_commit operation on each record and removing the record's state from the object store. If the first phase failed then the top_level_abort operation is invoked on each record which had the top_level_prepare operation invoked so that the implementation of each record may undo any effects of the prepare operation. In the case of the PersistentRecord this would take the form of removing the state of the object placed in the object store during the prepare operation. If the top_level_prepare operation was not invoked on a

record, for example as a result of the first phase being aborted before the record was processed, then the record need only have the `abort` operation invoked on it. After the top-level commit or abort operation is invoked on a record, the record is deleted by the `AtomicAction`. On completion of the processing of all records, the `AtomicAction`'s state is removed from the object store when the commit protocol terminates.

If a crash occurs then when the node recovers the crash recovery mechanism (implemented by the program `CrashRecover`) is invoked. This program utilises the two pieces of action management information contained in the object store. One piece, the states of the objects whose classes are derived from `AbstractRecord` (in the previous example PR_1 and PR_2), contains information about objects which were being processed by a top-level action as it was committing (the names of EC_1 and EC_2). The other piece of information is the state of the `AtomicAction`, the existence of which indicates that the first phase of that action successfully completed. The names of any records that are not in the `AtomicAction`'s state must belong to a top-level action that was either in the process of aborting or had not completed the first phase. The information in these records may be used to recover the objects that the records refer to. The records recorded in `AtomicAction` are used to commit the objects thereby completing the second phase.

3.7 The design of a distributed atomic action

The discussion up to this point has concentrated on atomic actions that manage purely local objects. This section briefly describes how the design and implementation of the class `AtomicAction` may be extended to encompass the distributed environment provided by Arjuna, in which a client program contains stub objects that communicate with servers.

Since remote objects are managed by servers, a client program that contains an atomic action needs an equivalent (server) atomic action at each server to manage the state of, and control access to, each object managed by the server. The atomic action in the client program is therefore responsible for coordinating all of these server actions to provide the abstraction of a distributed atomic action. When either the `Commit` or `Abort` operation is invoked on the client action, the same operation should be invoked on each server action, if the object at that server has been accessed during the scope of the client atomic action.

A new class could be defined to implement the functionality required by a server atomic action, but since much of this functionality is already provided by the previously described class `AtomicAction`, inheritance may be employed again so that the extra functionality may be added by simply deriving a new class that represents a server atomic action from `AtomicAction`. To create server actions, and keep the client and server actions in synchronisation, additional functionality is also required by the client stub object and server stub which are produced by the stub generator.

The approach is very simple. The first time an operation is invoked on a stub object, an instance of the class `ServerActionRecord` is added to the active atomic action. This instance contains the address of the server which is managing the object that the stub object represents. To be able to detect changes in the action environment, an instance of a class called `Client_Action_Stub`

(created by the stub object when it is instantiated) is provided. An operation provided by this class, which takes the server address as an argument, is invoked before the stub object makes the RPC to the server (but after the server has been created). This class is responsible for detecting changes in the action environment (by examining the `CurrentAtomicAction` variable). When such changes are detected, `Client_Action_Stub` creates and adds a `ServerActionRecord` instance to the current `AtomicAction`, and returns information to the stub object which is then added to the message that contains the RPC. Each RPC message therefore contains a small amount of action management information. This information describes the action hierarchy at the client, so that the server may ensure that this hierarchy exists before the operation on the (remote) object is invoked.

When an RPC is received by a server, the server stub which is responsible for unpacking the RPC arguments invokes an operation provided by the class `Server_Action_Stub` (instantiated by the server) passing the action information as an argument. Using this information, the `Server_Action_Stub` will create new server atomic actions to reflect the action hierarchy at the client, and set the global variable `CurrentAtomicAction` to point to the current atomic action.

The atomic action created at the server is an instance of the class `ServerAtomicAction`, which is a class derived from `AtomicAction`. The reason for deriving this class from `AtomicAction` is to utilise the support provided by `AtomicAction` for managing the state of, and access to, the object at the server. The additional functionality that `ServerAtomicAction` provides is two operations (a prepare and commit operation) which replace the single `Commit` operation. This enables the client atomic action to correctly control a server action during the two-phase commit protocol.

The client/server model employed by the distributed system allows servers to make further remote procedure calls, thereby creating nested servers. Since the class `ServerAtomicAction` is derived from `AtomicAction`, the invocation of an operation on a remote object by a server will result in the addition of a `ServerActionRecord` to the `ServerAtomicAction` and the creation of a `ServerAtomicAction` at the site of the remote object. The functionality inherited from `AtomicAction` will ensure that the `ServerAtomicAction` correctly manages any `ServerAtomicActions` that are created in this manner.

Each server is an instance of a class generated by the stub generator. The interface that the server class provides (i.e. the operations it recognises) consist of the operations provided by the class of the object, together with the operations provided by `ServerAtomicAction`. This enables the client atomic action, through the `ServerActionRecord`, to invoke the equivalent operations at the various action events during the lifetime of the client action. The implementation of `ServerActionRecord` utilises the server address to make RPCs directly to the server.

The following example may help to clarify how this design provides distributed atomic actions. Consider again the example given in section 3.5.1 (ignoring the nested action B). Assume that the client program is on node N_1. If the instance of `ExampleClass` EC_1 is contained on node N_2 then the declaration of EC_1 will produce a server (S_1) at that node. The sequence of events that occur when the `update` operation is invoked on EC_1 are illustrated in Figure 3.18. Each operation invocation is numbered in the order they occur.

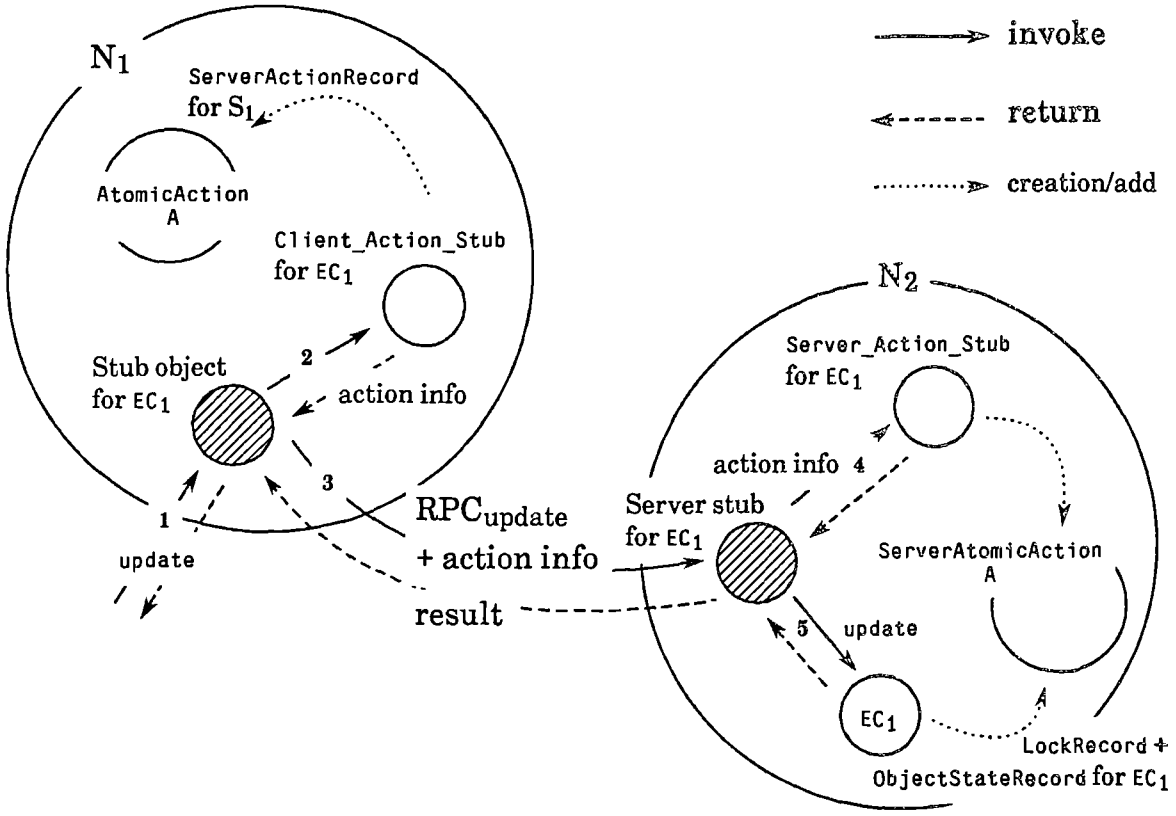

Figure 3.18: A distributed atomic action

When the `update` operation is invoked on the stub object for EC_1 (invocation 1), the stub object in turn invokes the operation provided by the `Client_Action_Stub` (created by the stub object) for EC_1 (invocation 2). The `Client_Action_Stub` operation recognises that the `AtomicAction A` has begun (by checking the value of `CurrentAtomicAction`), and that this is a new action and not an action that returned to being the current atomic action as a result of the termination of a nested action. Since A is a new `AtomicAction` an instance of `ServerActionRecord` containing the address of S_1 is added to A. In addition, the `Client_Action_Stub` operation returns the action information (in this case the name of A). The third invocation is the remote procedure call from the stub object to the server.

On receiving this RPC, the server invokes an operation provided by the Server_Action_Stub passing the action information as an argument (invocation 4). The Server_Action_Stub discovers that the AtomicAction A has no ServerAtomicAction at S_1, so a ServerAtomicAction for A is created, and the value of CurrentAtomicAction assigned to the result. Once control returns to the server stub, the update operation is invoked (invocation 5) on the object EC_1 which in turn creates a LockRecord and ObjectStateRecord and adds these to the current AtomicAction in the manner previously described.

When the Abort operation is invoked on A, the only object in A is the ServerActionRecord for S_1. In the abort operation provided by the ServerActionRecord, an RPC is made to S_1 to invoke the Abort operation on the ServerAtomicAction for A. Since A is still the CurrentAtomicAction the action environment has not changed so that no action information is passed. On receiving this RPC, the server stub invokes the Abort operation on the ServerAtomicAction A. This operation recovers EC_1 in the manner previously described.

An invocation of the Commit operation behaves in a similar manner, with the client atomic action acting as the coordinator of the commit protocol. During the prepare phase, the client atomic action invokes the ServerAtomicAction Prepare operation. If each ServerAtomicAction returns a value to the client atomic action which indicates that the prepare phase has successfully terminated at the server, then the client atomic action invokes the Commit operation on each ServerAtomicAction. If a ServerAtomicAction returns a value that indicates that the prepare phase could not be successfully completed, or the RPC to the server terminates abnormally, then the client atomic action invokes the Abort operation on all ServerAtomicActions that had invoked the Prepare

operation, and/or those `ServerAtomicActions` that had yet to invoke the `Prepare` operation.

The distributed atomic action design presented in this section has not been implemented, whereas the non-distributed atomic action design and record classes described in the previous sections have been implemented and extensively tested. This section and previous sections have described how a distributed atomic action may be provided using the features of an object-oriented language. The resulting atomic actions are objects which raises a number of issues involving defining the scope, and declaring instances, of an atomic action. These issues are described in the next section.

3.8 Atomic actions as objects

When an object-oriented language is used to implement atomic actions in the manner described in this chapter, the declaration and use of the resulting atomic actions differs from language or operating system based implementations. Before the operations provided by a class such as `AtomicAction` can be invoked, an instance of the class must be declared in an application. If atomic actions are provided by a language or operating system then there is generally no need to explicitly declare atomic actions before invoking the operations that they provide.

Another disadvantage of the object-oriented approach (which also applies to an operating system approach) is that an invalid sequence of operation invocations, such as a `Commit` invocation on an atomic action which has not had a previous `Begin` invocation, must be checked at run-time. If a language based approach is employed then the compiler for the language can check the syntax of the atomic action declarations so that such incorrect operation invocations can be discovered at compile, rather than run, time.

An additional disadvantage is defining the scope of atomic actions which are declared in an application. In the section which described the atomic action model, the scope was defined to be the region (or computation) bounded by the `begin` and `commit` operations. When an atomic action is aborted, the implementation of the atomic action should ensure that the scope of the aborted atomic action is the same as a committed atomic action, requiring the `abort` operation to return control to the point immediately after the `commit` operation. When an object-oriented approach is employed extra linguistic constructs must be used, or an atomic action and its computation declared using a procedure/block from which control may be return before the end of the procedure/block, to enforce the scope of an atomic action. Figure 3.19 illustrates these two approaches using

```
AtomicAction A;
    A.Begin();
        ....
        if (operation failed)
        {
            A.Abort();
            goto endofAction;
        }
        ....    // do some more work
    A.Commit();
endofAction:    // end of scope
```

(a)

```
int AtomicOperation()
{
    AtomicAction A;
    ....
    A.Begin();
    if (operation failed)
    {
        A.Abort();
        return (-1);
    }
    ....    // do some more work
    A.Commit();
    return (0);
}
```

(b)

Figure 3.19: Enforcing scope

the language C++. Figure 3.19(a) employs the `goto` statement to move the execution of the program when the atomic action is aborted to the point immediately after the `Commit` operation. Figure 3.19(b) is an example of a function, that returns a value indicating whether the atomic action committed or aborted, which employs the `return` statement to leave the function as soon as the atomic action is aborted.

To avoid having to explicitly declare instances of the class `AtomicAction` and include extra linguistic constructs to enforce the scope of the resulting atomic actions, additional mechanisms such as pre-processor macros may be employed. The technique of using preprocessor macros to add exception handling constructs to C programs was described in [Lee 83], and a similar technique may be used with the class `AtomicAction`. Using such pre-processor macros, the example in 3.19(a) may be changed to that illustrated in Figure 3.20. The macros used in this

```
BEGIN
    ....
    if (operation failed)
        ABORT
    ...         // do some more work
END
```

Figure 3.20: Enforcing scope using macros

example define the scope of the resulting atomic action to be the computation between the `BEGIN` and `END` operations. If the computation reaches the `END` operation then the atomic action is committed. When the atomic action should be aborted the `ABORT` operation may be invoked, with control returning to the point immediately after the `END` operation.

3.9 Concluding remarks

This chapter described the issues involved in the design and implementation of atomic actions in a distributed environment using the support provided by an object-oriented language. The approach was to design a non-distributed atomic action and then extend this design to enable remote as well as local objects to be controlled by the resulting distributed atomic action. The reason such an approach is practical, is the way the implementation of the non-distributed atomic actions is structured. During the design, common functionality between the various activities an atomic action must provide was recognised. This led to use of inheritance to define a base class that provided the common functionality,

with each activity being provided by classes (termed records) derived from this base class. The implementation of this scheme to control local objects was realised, and the design then extended to enable access to remote objects to be controlled.

To manage each record, the class `AtomicAction` was defined. The implementation of this class provides the properties that characterise an atomic action. In the next two chapters, further examples of the use of this class will be given. The objects controlled by these atomic actions must provide recovery and concurrency control, and have associated record classes to manage these properties. The next chapter discusses the issues involved in providing recoverable objects, and describes a design, along with its implementation, which addresses these issues. Chapter five describes how the design may be extended to ensure objects are persistent.

Chapter 4

Recoverability

The last chapter described how nested atomic actions may be constructed. During this discussion, a number of implementation techniques were considered, and an approach based on employing the features of an object-oriented language adopted. The advantage of this approach is that it avoids the expense of modifying, or implementing new, languages or operating systems. The disadvantage however, is that the support for objects generally provided by a language compiler, or operating system, that directly supports atomic actions must be provided by the objects themselves. The support required of an object is that it is recoverable, persistent, and provides concurrency control. The subject of this chapter concerns the first of these properties: *recoverability*.

To support the failure atomicity property of an atomic action requires the addition of recoverability to an unrecoverable class of objects. There are two aspects to providing recoverability, the first is the construction of recoverable objects, the second is the management of these objects so that recovery occurs when an atomic action is aborted.

The recovery issues discussed in this chapter are those that relate to the recovery of the volatile state of an object. The recovery techniques employed are well known, the purpose of this chapter being the description of a new approach to the addition of these techniques to an unrecoverable class of objects to produce recoverable objects. To manage recoverable objects, this chapter also describes how a *record* (the abstraction by which the atomic action manages the various properties) may be used to coordinate the type of recovery an object provides with the control exercised by an atomic action. The advantage of the record abstraction is that a variety of different management techniques can be employed

to construct recoverable objects, and a recoverable object may consist of a number of recoverable objects each providing alternative forms of recovery.

The chapter begins by discussing recoverability and two common recovery techniques. To add recovery mechanisms to an unrecoverable class of objects, using the features of an object-oriented programming language, a number of approaches are considered in the following section. The section after discusses how the resulting recoverable objects may be managed by an atomic action, so that they are recovered if the atomic action is aborted. The two sections following describe how recoverable objects may be implemented using the two recovery techniques, and are followed by a discussion of the issues involved in providing new abstractions from unrecoverable and/or recoverable objects. The final section assesses the technique developed in this chapter for constructing a recoverable class of objects.

4.1 Providing recoverability

When an atomic action is aborted, all the objects modified by the computation encapsulated by the atomic action must be restored to their previous state. This capability is supported by the property which is termed *recoverability*. This section discusses how this property may be added to unrecoverable objects.

A recoverable object is an instance of a *recoverable class*, which is a class that includes recovery mechanisms. Producing a recoverable object from an unrecoverable object therefore involves adding recovery mechanisms to the class of the unrecoverable object. During the invocation of operations on instances of a recoverable class, the recovery mechanisms must create management information to enable an atomic action to control the recoverable object. The construction and management of recoverable objects are discussed in detail in the

next two sections, this section provides an overview of the main issues surrounding recoverability.

The section on object-oriented languages in chapter two described how an object is an abstraction that may consist of a collection of internal objects. This abstraction ensures that the state of an object viewed by a user is the *abstract state* of the object. The implementation of an object's abstract state is termed the *concrete state*, and is supported by the implementation language and the underlying hardware. Because of this abstraction, the concrete state of an object may change in such a way that the abstract state presented to a user of the object remains unchanged. A recoverable object is therefore defined to be an object that can be restored to a previous abstract state. This definition allows the concrete state of an object to differ from a previous concrete state when the abstract state of a recoverable object is restored.

Techniques which provide the abstraction of recovery generally take one of two forms: either *state* or *operation* based. A state based recovery technique takes a copy (or *snapshot*) of the state of an object before the object is modified. During recovery, the current object state is replaced by the old state or snapshot. An operation based recovery technique records the operations invoked on an object, enabling the state to be recovered by sequentially invoking the inverse of each operation recorded. Implementations of both techniques have been made in a number of ways, for example [Schwarz 84] describes how a log may be used to hold the recovery information, whereas [Horning *et al.* 74, Anderson and Kerr 76] describe how the *recovery cache* (a stack) may be used.

Both recovery techniques require recovery information to be created during operations that modify the state of an object, which consists of the recovery data required to restore the state of the object and the management information required by the atomic action. The amount of recovery data created and

maintained by each technique varies. Operation based recovery techniques generally require more recovery data since recovery is related to the number of operations invoked on an object, whereas with a state based recovery technique, the recovery data need only be recorded the first time the object is modified within an atomic action.

Each time an object is modified within a new atomic action, recovery information must be created and added to the atomic action. When a nested atomic action commits, the recovery information maintained by the nested atomic action must be merged into the parent atomic action (as described in the last chapter). The recovery information maintained when a state based approach is used can be discarded during a merge if the parent atomic action has more suitable (i.e. older) recovery information for that object. In contrast, when an operation based approach is employed, the recovery information maintained in a nested atomic action must be added to that held by the parent atomic action.

The recovery information maintained about a recoverable object will be dependent upon the class of the object, since the recovery mechanism employed by an atomic action must know what operations to invoke on an object to effect recovery. Some objects are inherently more recoverable than others and more suitable for a particular recovery technique. For example, objects that provide an assignment operation (such as integers) may be recovered using a state based approach that involves saving a copy of the object and reassigning this copy during recovery of the object. An object that provides the abstraction of a stack however, may be better suited to an operation based approach, as a record of the push and pop operations (and arguments) could be used to recover the stack by sequentially invoking the inverse operation of each operation recorded in the recovery data.

If an object provides no suitable operations to support recovery, or more efficient mechanisms are required, then additional recovery mechanisms may be added. A means of adding such mechanisms are described in the next section. Not all unrecoverable objects however, are suitable for the pure state or operation based recovery techniques described above. For instance, objects which affect the external environment of the system may be difficult, if not impossible, to recover. In such circumstances the abstraction of recovery may be provided by performing compensation operations on the environment affected by the operations invoked on the object. The compensation required will be specific to a class and a function of the operations invoked upon an instance of the class, so that this approach is an extension of the operation based approach (an example of class-specific compensation is described in [Shrivastava and Banâtre 78]). Compensation highlights the fact that recovery need only restore the *abstract* state of an object. The *concrete* implementation of the object does not necessarily have to return to the previous state held for recovery to be effective.

4.2 Constructing recoverable objects

To support recoverability an object must be capable of restoring a previously held abstract state. This section describes how this capability may be added to an unrecoverable class using the features of an object-oriented programming language.

To construct a new recoverable class from an unrecoverable class requires the addition of recovery mechanisms that correctly manage the abstract state of the unrecoverable class. When atomic actions are supported by a language or operating system the compiler for the language, or object management provided by the operating system, generally provide the recovery mechanisms that ensure an object is recoverable and controlled by an atomic action. The disadvantage of language or operating system support however, is that they are almost

exclusively based on managing the state of an object. Providing objects that may be recovered by invoking inverse operations relies on knowledge of the semantics of the class of the object, and as a result are difficult to generate automatically.

If a system does not support recovery then the recovery mechanisms must be explicitly added to an unrecoverable class. The burden of this task lies with the implementor of the recoverable class, but can be approached in a number of ways using the features of an object-oriented language.

One approach is to produce a new class that contains an instance of the unrecoverable class and provides a set of operations that are equivalent to those provided by the unrecoverable class, but which create suitable recovery information when an instance of the recoverable class is modified. To support recovery, the recoverable class should also provide an operation that may be invoked during an atomic action abort to recover an instance of the class using the recovery data. This approach (which will be termed the *container* approach) may be illustrated by considering how to implement a class that provides the abstraction of a recoverable integer (i.e. integers that can be used within the scope of an atomic action). Assuming that integers are instances of a class called `Integer`, and that the recoverable interface is provide by an operation called `recover` (the operation that will be invoked by the atomic action to recover the object), then a recoverable integer may be implemented in the manner illustrated in Figure 4.1. This example is written in the language C++, which allows the implementor of a class to declare operations such as the assignment operation (as `operator=`). The disadvantage of this approach, is that the implementor of the recoverable class has to provide all the operations provided by the unrecoverable class. In addition, since a `RecoverableInteger` is a new class, each operation must be *overloaded* to take both `RecoverableInteger` and `Integer`

```
class RecoverableInteger
{
    Integer value;
public:
    ....
    operator=(Integer);
    operator=(RecoverableInteger);
    operator+(Integer);
    operator+(RecoverableInteger);
    operator*(Integer);
    operator*(RecoverableInteger);
    .... // and all the other Integer operations
    recover();
};
```

Figure 4.1: The *container* approach

arguments so that instances of `RecoverableInteger` may be used in place of instances of `Integer`.

A superior approach is to employ inheritance to derive a new recoverable class from the unrecoverable class. In this way, the implementor need only refine those operations which modify the inherited unrecoverable state, so that suitable recovery information may be created before the modifications take place. This approach will be termed the *unrecoverable inheritance* approach. In a similar manner to the *container* approach, the new recoverable class should provide an operation that constitutes the *recoverable interface*. An alternative implementation of a recoverable integer class using this approach is illustrated in Figure 4.2. The advantage of the *unrecoverable inheritance* approach is clear from

```
class RecoverableInteger : public Integer
{
    ....
public:
    ....
    operator=(Integer);
    recover();
};
```

Figure 4.2: The *unrecoverable inheritance* approach

the class declaration in Figure 4.2. The number of operations the implementor of the class must re-implement is substantially reduced to only those operations

that modify the object. In addition, since `RecoverableInteger` is a sub-type of `Integer` the operations do not need to be overloaded as each operation may be defined to take an argument of class `Integer`, but may be invoked with an argument of class `Integer` or `RecoverableInteger`.

Implementing recoverable objects in the manner described above is dependent upon semantic knowledge of an unrecoverable object for which recovery is being provided. In the examples illustrated in Figures 4.1 and 4.2, each class represents an integer, the semantics of which are well known, so that the implementor of the recoverable class may employ this knowledge to refine the operations that modify the state of the object (in this example the assignment operation). If the semantics of a class are unknown then constructing recoverable objects in this manner will not be possible, but if the semantics are not known then it will not be possible to use the unrecoverable objects in any case.

Using the features of an object-oriented language in this manner, the recovery mechanisms added to an unrecoverable class will be largely dependent on the semantics of the unrecoverable class, but may share common functionality due to the support the recoverable class provides for one of the recovery techniques described in the last section. A general technique is needed to add this common functionality and thereby avoid unnecessary duplication.

During the discussion on the design of the management records in the last chapter, the advantages of inheritance for adding new functionality to an existing class was outlined. In the record class design this took the form of defining a base class with common functionality and deriving new classes to provide additional class-specific functionality. An approach of this sort is also suitable for adding recovery mechanisms to an unrecoverable class of objects, by defining a base class that implements the recovery mechanisms and deriving new classes from this *base recoverable* class.

If an object-oriented language only supports sub-typing inheritance, then each new recoverable class must be implemented in the manner described in the first of the above two approaches, and will effectively act as a *container* for the unrecoverable object, managing the abstract state of the unrecoverable object by employing the inherited functionality. This *recoverable inheritance* approach is similar to the type generator *mutex* provided by the *Argus* programming language [Liskov 84] for generating user-defined *atomic* (recoverable) *types* [Weihl 84]. Another declaration of a recoverable integer class is illustrated in Figure 4.3, with the recovery mechanism class assumed to be implemented by the

```
class RecoverableInteger : public RecoveryMechanism
{
    Integer value;
public:
    ....
    operator=(Integer);
    operator=(RecoverableInteger);
    operator+(Integer);
    operator+(RecoverableInteger);
    operator*(Integer);
    operator*(RecoverableInteger);
    ....    // and all the other Integer operations
};
```

Figure 4.3: The *recoverable inheritance* approach

class `RecoveryMechanism`. The advantage of this approach over the *container* approach is that recoverable interface (in the form of the `recover` operation) is provided by the `RecoveryMechanism` class. In addition, it is assumed that the `RecoveryMechanism` class provides an operation called `record` which may be invoked to maintain the recovery information required by a recoverable object.

Both inheritance approaches rely on the provision of additional functionality by the implementor of the recoverable class to correctly manage the unrecoverable class (or object). The advantage of the unrecoverable inheritance approach is that the recoverable class need only refine those operations that modify the object (ignoring concurrency control aspects). There is no need to refine the remaining operations, which may be contrasted with the recoverable

inheritance approach, where all the operations provided by the unrecoverable class must be re-implemented since the recoverable class acts as a container for the unrecoverable object. The principle advantage of the recoverable inheritance approach however, is that all recoverable classes will be sub-types of the `RecoveryMechanism` class, enabling the recoverable objects to be managed by a common management mechanism. With the unrecoverable inheritance approach, each recoverable class will be a new type and require a management mechanism that is specific to that particular class.

The above discussion has been considering sub-typing inheritance, where a class may only have a single super-class. The disadvantages of the two inheritance approaches can be removed if the implementation language supports multiple-inheritance, where a class can have multiple super-classes. By employing multiple-inheritance the functionality of the unrecoverable class may be mixed with the `RecoveryMechanism` class, requiring only a subset of the operations provided by the unrecoverable class to be refined, and enabling the new recoverable class to be treated as a sub-type of both the `RecoveryMechanism` class and the unrecoverable class. Figure 4.4 illustrates

```
class RecoverableInteger : public Integer, RecoveryMechanism
{
public:
    ....
    operator=(Integer);
};
```

Figure 4.4: The *multiple inheritance* approach

another implementation of the recoverable integer class (in C++) that employs multiple inheritance. This *multiple inheritance* approach is therefore the most suitable approach to adding recovery mechanisms to an unrecoverable class, and is the approach assumed whenever recoverable classes that are constructed using inheritance are discussed.

Employing inheritance to construct recoverable objects is a powerful technique that has not been previously exploited. By providing suitable support, in the form of flexible recovery mechanisms, the construction of recoverable classes from unrecoverable classes using inheritance greatly eases the burden of the implementor of a recoverable class. This technique, which was first described in [Dixon and Shrivastava 87], will be described in greater detail in the remainder of this chapter.

To manage the recoverable objects requires the addition of management information to an atomic action. The next section describes how this management information may be used to ensure an object is recovered when an atomic action is aborted, and how the recovery data needed by the recoverable object may be managed.

4.3 Managing recoverable objects

There are two aspects to the management of a recoverable object, the first involves notifying an atomic action that a recoverable object has been modified, the second, recording sufficient recovery data to enable the abstract state to be restored. The previous section describes how it was assumed that the class `RecoveryMechanism` provides an operation called `record` for this purpose, which is used to maintain the data needed to recover an object and record suitable management information with the current atomic action. Notifying an atomic action that an object has been modified need only occur on the first modification, but recording recovery information may occur on each modification (particularly if an operation based recovery technique is employed).

When an atomic action is aborted, the recoverable objects recorded in the management information maintained by the atomic action should be recovered. If a recoverable object provides an operation such as `recover` then an atomic action simply has to invoke this operation to restore its abstract state. If all

recoverable objects provide an identical recoverable interface then the implementation of the recovery management mechanisms will be greatly simplified.

The discussion so far has assumed that the management of recovery is a part of a recoverable object. As a result, a recoverable object must take part in the commitment of an atomic action, since the recovery data maintained for each atomic action has to be merged into the parent atomic action. If instead, the recovery of an object is directly managed by the atomic action, along with the maintenance of the recovery data, then the atomic action could merge the recovery information, and only involve the recoverable object when the atomic action is aborted. This separation of management from the basic recovery properties will simplify the class that provides the recovery mechanism. This approach has been adopted by the implementation described in the next section. This implementation of recoverable objects using inheritance operates in the atomic action framework described in the last chapter.

4.4 Implementing recoverable objects

The separation of the management of recovery from the basic recovery property is possible using the record classes described in the last chapter. A record class is responsible for managing a particular property, so that new record classes may be defined to manage various recovery techniques. The following sections describe the implementation of the two recovery techniques described earlier in this chapter, along with the corresponding record classes that are responsible for managing each technique.

The technique described in the next section provides recovery based on the old state of an object. To support state based recovery a class called `Object` has been implemented. This class provides a recoverable interface that supports operations which may be invoked to retrieve and restore the internal state of a

recoverable object. Instances of a record class called `ObjectStateRecord` are created by `Object` to add to an atomic action to indicate that a recoverable object (constructed using `Object`) has been modified. An `ObjectStateRecord` instance also maintains the recovery data required to restore a recoverable object, which is contained in an instance of another class called `ObjectState`. The design and implementation of each of these classes are described in the next section.

Two examples of operation based recovery techniques follow the state based approach. The first example adopts a similar approach to the state based approach, by defining a base recoverable class called `Operation` that may be inherited to construct a recoverable class. To manage the resulting recoverable objects constructed using the class `Operation`, a record class called `OperationRecord` is provided. The recovery data that the `OperationRecord` objects manage are instances of classes derived from an abstract class called `OperationLog`. The second example of a recoverable class that employs an operation based recovery technique simply defines a new record class, and is only derived from the unrecoverable class. This alternative implementation is provided to illustrate the flexibility of the record class abstraction, and describe how compensation may be employed to provide the abstraction of recovery.

In the following two sections, the discussion is limited to how to add recovery mechanisms to an existing unrecoverable class to produce a recoverable version of the unrecoverable class. The recovery techniques described in these sections may also be used to construct new recoverable classes that contains instances of both recoverable and unrecoverable classes. The issues involved in constructing new recoverable classes rather than adding recovery mechanisms to existing classes will be discussed in detail in a section following the next two sections.

4.5 Implementing state based recovery

This section describes how a state based approach to recovery may be implemented within the atomic action framework described in the last chapter. A suitable starting point is to consider the interface that a recoverable class should provide. As the discussion in a last section concluded, the management of a recoverable object is best left to an atomic action. As a result, the recoverable interface may consist of an operation that returns the state of a recoverable object (called `save_state`), and a complementary operation (called `restore_state`) that restores a previous state which is passed as an argument. When a recoverable class is constructed from the (base recoverable) class which provides the `save_state` and `restore_state` operations, these two operations should be refined to save and restore the state of the recoverable class in a form defined by the base recoverable class. The way the recovery data is represented defines the form required.

While the mechanics of state based recovery can be considered to be independent of the class of an object, the recovery data required is not. If inheritance is to be used to add the basic functionality then the recovery mechanisms and data should be independent of the class to which they are being added. One approach is to employ an abstraction in the form of a class that may be refined to manage class specific recovery data, but may be treated as an instance of the base abstract class by the inherited recovery mechanisms.

Consider a class called `AbstractState` which constitutes the recovery data for the `save_state` and `restore_state` operations. Each new recoverable class may derive a class from `AbstractState` to contain the recovery data that the new recoverable class requires. For instance, when constructing a recoverable integer a class may be derived from `AbstractState` (say `IntegerState`) to maintain an `Integer` instance that is a copy of the value of

the unrecoverable integer. When recovery occurs, the recovery management will invoke the base recoverable class `restore_state` operation passing an instance of `AbstractState` as an argument. In practice however, the `restore_state` operation invoked will be the refined version in the recoverable integer class, and the argument passed will be an instance of `IntegerState`.

The disadvantage of the above approach to providing the recovery data is that for each recoverable class, an associated class that maintains the recovery data must be defined. An alternative approach, is to provide a means by which the recovery data is managed in a class-independent manner.

The method adopted by the implementation which will be described in the rest of this section, is to provide a class called `ObjectState` that maintains a snapshot of the state of an object as a *bit-image*. To convert the state of an object into a bit-image, the class `ObjectState` provides an operation (called `pack`) which copies the state of an object (in terms of the storage associated with the underlying hardware). If a newly defined recoverable class consists of a number of objects then each object may be copied into an `ObjectState` instance so that the state of the recoverable object will consist of a contiguous block of storage. In effect, the `pack` operation behaves in a similar manner to the similarly named procedure required to marshal parameters for remote procedure calls.

To restore the state of an object, the `ObjectState` class provides a complementary operation called `unpack`. This operation copies the state from the `ObjectState` buffer into the volatile storage that constitutes the state of the object being restored. The function of the `Object` operations (`save_state` and `restore_state`) and the `ObjectState` class is therefore to provide a class independent assignment operation. The advantage of this approach is that it is more efficient than employing a class specific assignment operation if the state of an object consists of a number of internal objects that are contiguous (such as an

array). In addition, the class enables objects that are inherently unrecoverable (i.e. which have no operations that may be used to recover them) to become recoverable.

The base recoverable class that provides the recoverable interface has been briefly mentioned in the last chapter when describing the way the atomic action subsystem is structured. A skeleton declaration of this class, which is called `Object`, is illustrated in the language C++ in Figure 4.5. In addition to the two

```
class Object
{
    ....
protected:
    void modified();
    ....
public:
    ....
    virtual ObjectState* save_state(ObjectState*);
    virtual ....         restore_state(ObjectState*);
    ....
};
```

Figure 4.5: The class `Object`

operations previously described, the class `Object` also provides an operation called `modified` (the label `protected` in Figure 4.5 is a C++ encapsulation mechanism that ensures only derived classes may invoke operations that follow this label). The function of the `modified` operation is to behave in a manner similar to the operation `record` described in a previous section. That is, before an object is modified the `modified` operation should be invoked to record management information with an atomic action and create the recovery data required to restore the recoverable object. A more detailed description of this sequence of events will be given later in this section.

To illustrate how a recoverable class may be declared, Figure 4.6 is the

```
class RecoverableInteger : public Object,Integer
{
public:
    RecoverableInteger();

    virtual ObjectState* save_state(ObjectState*);
    virtual void         restore_state(ObjectState*);

    Integer operator=(Integer&);

};
```

Figure 4.6: The class `RecoverableInteger`

declaration (in C++) of a class that provides the abstraction of a recoverable integer. In this class declaration it is assumed that C++ provides multiple inheritance and that integers are represented by the class `Integer`. The operations refined are the `save_state` and `restore_state` operations, and the assignment operation (`operator=`). An implementation of the assignment operation is illustrated in Figure 4.7 which explicitly names the inherited

```
Integer RecoverableInteger::operator=(Integer& newvalue)
{
    Object::modified();
    return (Integer::operator=(newvalue));
}
```

Figure 4.7: `RecoverableInteger` assignment

operations that are invoked. An implementation of the `save_state` operation is illustrated in Figure 4.8 which employs the C++ `sizeof` operation to return the

```
ObjectState* RecoverableInteger::save_state(ObjectState* newstate)
{
    newstate->pack((Integer*)this,sizeof(Integer));
    return (newstate);
}
```

Figure 4.8: `RecoverableInteger save_state` operation

size of an `Integer` object in bytes. In this implementation, the pseudo-variable provided by the C++ compiler (called `this`) which points to the state of an object in volatile storage is used to access the storage associated with the inherited state from `Integer` by casting the pointer. The complementary operation

`restore_state` is illustrated in Figure 4.9 (the size is not required for this

```
void RecoverableInteger::restore_state(ObjectState* oldstate)
{
    oldstate->unpack((Integer*)this);
}
```

Figure 4.9: `RecoverableInteger restore_state` operation

operation as it is maintained by the `ObjectState` object).

The discussion so far has concentrated on how to construct a recoverable class from an unrecoverable class. The discussion may now move on to how to manage the resulting recoverable objects. To ensure that instances of the recoverable class are correctly managed by an atomic action requires that management information is added to an atomic action. In the last section, the manner in which the recovery information is implemented was described. The approach is to employ an abstraction termed a *record*, and to derive classes from the base record class to produce the required functionality.

Given the base record class `AbstractRecord`, a suitable record class to manage the recoverable classes constructed in the manner described in this section may be provided. The record class for state based recovery, which is called `ObjectStateRecord`, manages an instance of the class `ObjectState` that in turn manages the recovery data for a recoverable object. When a recoverable object is modified an instance of the class `ObjectStateRecord` is created to manage and hold the newly created `ObjectState` instance (which contains the current object state), and this is then added to the current atomic action.

The operation which creates the `ObjectState` and `ObjectStateRecord` instances is the `modified` operation provided by the class `Object`. This operation should be invoked in the implementation of each operation in a recoverable class that modifies the inherited state. When invoked, the `modified` operation begins by creating an `ObjectState` instance and invoking the

save_state operation to save the current state of the recoverable object in the newly created ObjectState instance. An ObjectStateRecord is then created, with the ObjectState instance being passed as an argument, and is added to the current atomic action. The modified operation may be called a number of times as operations are invoked on a recoverable object, but this sequence of events only occurs the first time the modified operation is invoked in each atomic action.

To abort an atomic action the Abort operation provided by the class AtomicAction may be invoked. The implementation of this operation involves invoking the abort operation implemented by each record instance that the AtomicAction instance is maintaining. In the implementation of the ObjectStateRecord abort operation (illustrated in Figure 4.10) the

```
void ObjectStateRecord::abort()
{
    object_addr->restore_state(state);
}
```

Figure 4.10: ObjectStateRecord abort operation

ObjectState instance is used as the argument to the restore_state operation, thereby recovering the state of the recoverable object. Each ObjectStateRecord object contains a pointer (called object_addr) to the recoverable object that created the ObjectStateRecord instance, and a variable (called state) that points to the ObjectState instance for the recoverable object, enabling the ObjectStateRecord implementation to invoke the restore_state operation in the manner shown in Figure 4.10.

This section has described how a recoverable class that implements state based recovery may be constructed from an existing unrecoverable class and a base recoverable class using inheritance. The next section describes how an

operation based approach to recovery may be used, and is followed by a discussion on how to construct recoverable classes that provide new abstractions.

4.6 Implementing operation based recovery

This section describes how recovery mechanisms which are based on recording the operations invoked on an object may be implemented within the atomic action framework described in the last chapter. To be suitable for such an approach, the semantics of the unrecoverable class must be known by the implementor of the recoverable class so that the inverse of an operation may be invoked when recovery is required. Hence, each unrecoverable class must have a set of operations that have suitable inverse operations.

Providing an unrecoverable class is suitable, then a new recoverable class may be constructed in a similar manner to the state based recovery technique where class dependent recovery data is employed. Since the purpose of an operation based recovery technique is to undo the operations invoked on an object, a suitable base class called `Operation` may be defined that provides two operations: `undo` and `record`. The skeleton declaration for such as class is illustrated in Figure 4.11. Both operations defined by the class `Operation` take

```
class Operation
{
    ....
protected:
    void record(OperationLog*);
    ....
public:
    ....
    virtual void undo(OperationLog*);
    ....
};
```

Figure 4.11: The class `Operation`

instances of the class `OperationLog` which is provided to log the operations invoked on a recoverable object. The `OperationLog` class is an abstract class

provided so that new classes may be derived from it to provide class-specific recovery data.

To illustrate the basic mechanism (the management issues being described later in this section) consider the construction of a class that provides the abstraction of a recoverable stack of integers. The unrecoverable class that implements a stack of integers is called `Stack` and provides two operations: `push` and `pop`. The class declaration for the recoverable stack is given in Figure 4.12,

```
class RecoverableStack : public Operation, Stack
{
    ....
public:
    ....
    void    push(Integer);
    Integer pop();
    virtual void undo(OperationLog*);
    ....
};
```

Figure 4.12: The class `RecoverableStack`

illustrating how the recoverable class refines the `push`, `pop`, and `undo` operations. To provide suitable recovery data the class `StackOperationLog` is also provided (illustrated in Figure 4.13). This class is derived from

```
enum StackOperation {PUSH, POP};
class StackOperationLog : public OperationLog
{
    StackOperation Op;
    Integer argument;
public:
    StackOperationLog(StackOperation,Integer);
    StackOperation get_operation();
    Integer        get_argument();
};
```

Figure 4.13: The class `StackOperationLog`

`OperationLog` and (to simplify issues) contains the name of a single operation (as a value of type `StackOperation`) that was invoked on an instance of the

recoverable class, along with the argument needed when the inverse operation is the push operation.

When either the push or pop operation is invoked on a RecoverableStack instance, an instance of StackOperationLog is created and the record operation provided by the Operation class invoked. The implementation of the pop operation is illustrated in Figure 4.14. The record operation performs a

```
Integer RecoverableStack::pop()
{
    Integer temp = Stack::pop();
    if (CurrentAtomicAction
          && CurrentAtomicAction->Status() == RUNNING)
       Operation::record(new StackOperationLog(POP,temp));
    return temp;
}
```

Figure 4.14: RecoverableStack pop operation

similar service to the modified operation provided by the class Object, adding an instance of a record class (called OperationRecord) to the current atomic action. Before invoking this operation, the implementation of the push and pop operations determine the status of the currently executing atomic action. If the status is anything other than RUNNING then the record operation is not invoked. The reason for this test is that each of these operations will be used to undo the inverse operation during recovery (when the atomic action status will be ABORTING), at which point recovery information should not be created and added to the atomic action.

When the atomic action is aborted, the abort operation implemented by the OperationRecord class simply invokes the undo operation passing the OperationLog instance as an argument. In the case of the RecoverableStack, the undo operation invoked is the refined version and the OperationLog instance passed is actually an StackOperationLog instance. The implementation of the undo operation (illustrated in Figure 4.15) provided by the RecoverableStack class employs the StackOperation field in the

```
void RecoverableStack::undo(OperationLog* Log)
{
    StackOperationLog *Slog = (StackOperationLog*) Log;
    if (Slog->get_operation() == POP)
        Stack::push(Slog->get_argument());
    else
        Stack::pop();
}
```

Figure 4.15: The RecoverableStack undo operation

StackOperationLog instance to invoke the inverse operation, passing the argument if this operation is the push operation.

The simple operation based recovery technique described above can be optimised in a number of ways, for example by maintaining a single log of operations rather than a single operation per log object, and a single record class instance per atomic action rather than a record class instance per operation. The description of the simple recovery technique illustrates however, the flexibility of employing inheritance to construct a recoverable class from an unrecoverable class.

To further illustrate the power of this technique, consider a class that manages a physical resource such as a printer. Once a printout is sent to a printer, the act of sending the printout cannot be recovered, but the abstraction of recovery can be provided by performing a compensation operation [Shrivastava and Banâtre 78]. To illustrate an alternative method of implementing operation based recovery, the following example employs a class specific record class to provide the necessary functionality. This approach may be contrasted with the approach described above (which employs the class Operation) since the recoverable class is only derived from the unrecoverable class and relies on the record class to provide the recovery mechanism rather than a base recoverable class such as Operation.

To provide the capability of producing a hard copy of the state of an object during its lifetime, consider the class `Printer`. This class is an abstraction of a physical printer typical to many systems, and is an unrecoverable object that provides three operations. The first is `print` which sends the output from a recoverable object (the result of invoking the `printOn` operation implemented by the class `Object`) to the physical printer, returning the job number of the printout. This job number may be saved by an application and used to stop the printout by invoking the operation `kill`. The final operation is called `status` which, as its name suggests, returns the status of a particular job as an enumerated value of type `PrinterStatus`. The class definition for `Printer` is illustrated in Figure 4.15. In addition to the class declaration, the `Printer`

```
enum PrinterStatus {QUEUED,PRINTING,PRINTED,KILLED};

class Printer
{
    ....

public:
    Printer (String);
    ~Printer();

    Integer       print (Object&);
    PrinterStatus kill  (Integer);
    PrinterStatus status(Integer);
};

inline void operator<<(Printer& P,Object& O)
{
    P.print(O);
}
```

Figure 4.15: The class `Printer`

definition includes an `inline` function that enables an instance of `Printer` to be used in the same way as a C++ output stream. Given an instance of `Printer` called `tweedmouth`, statements of the form

```
tweedmouth << instance_of_class_derived_from_Object;
```

can be made to print the state of an object.

To provide a recoverable printer that automatically kills print jobs if the atomic action within which the `print` operation was invoked is aborted, the class

RecoverablePrinter is provided. This class is derived from Printer and provides a single operation, a refined version of the print operation. To ensure that instances of this class are recoverable, the implementation of the print operation adds an instance of the class PrinterRecord (using the AtomicAction add operation) to the currently executing AtomicAction (accessed through the CurrentAtomicAction variable). The body of the print operation is illustrated in Figure 4.16 (the method used to add the

```
void RecoverablePrinter::print(Object& o)
{
    int job_number = Printer::print (o);
    if (CurrentAtomicAction)
        CurrentAtomicAction->add (new PrinterRecord (this, job_number));
}
```

Figure 4.16: The RecoverablePrinter print operation

PrinterRecord being similar to that used to add ObjectStateRecords during the modified operation provided by Object). The state of a PrinterRecord consists of a reference to the Printer object (printer) and the job number (p_id) passed as arguments to the class's constructor (as this and job_number respectively in the above example).

When the enclosing AtomicAction is aborted the abort operation provided by the PrinterRecord is invoked (illustrated in Figure 4.17). This operation

```
void PrinterRecord::abort ( )
{
    PrinterStatus status = printer->status(p_id);
    if (status == PRINTING || status == QUEUED)
        printer->kill (p_id);
    if (status == PRINTING || status == PRINTED)
    {
        Message message(form("Discard printout with job number %d", p_id));
        printer->print(message);
    }
}
```

Figure 4.17: The PrinterRecord abort operation

first finds the status of the job. If the job is either waiting to be printed, or is in

the process of being printed, then recovery occurs as the operation invokes the `kill` operation on the `RecoverablePrinter` object (passing the job number as the argument). If part, or all, of the job has been printed then compensation occurs in the form of a message sent to the printer indicating that the printout should be discarded. The compensating message is provided by the creation of an instance of the class `Message` (a class derived from `Object`) that contains the message text. Since the `Message` class is a subtype of `Object`, the print operation provided by `Printer` may be directly invoked with a `Message` instance as an argument.

This section has described how operation based recovery techniques may be used to add recovery mechanisms to an unrecoverable class. The previous section described how a state based approach may be used, and the next section discusses how both these approaches may be used to construct new recoverable classes that contain recoverable and/or unrecoverable objects.

4.7 Constructing a new recoverable class

The discussion so far has concentrated on constructing simple recoverable classes that consist of a single unrecoverable class. This section discusses the issues involved in constructing a recoverable class that consists of more than one recoverable and/or unrecoverable class to provide a new abstraction.

A newly defined recoverable class can be constructed in one of three ways. The first is a recoverable class that consists of objects that are already recoverable, the second a class where all the objects are unrecoverable, and the third a class where there is a mixture of recoverable and unrecoverable objects. In each of these cases, if the new abstraction requires more than one instance of a particular class then inheriting that class (in the manner described in previous sections) cannot be used, since inheritance effectively provides only a single instance of the class that is inherited. The rest of this section therefore concentrates on how to

construct recoverable classes that instantiate other classes. The classes instantiated by a recoverable class are termed the *internal* objects of the *containing* recoverable class.

To illustrate the different approaches, the following discussion will employ a simple class that provides the abstraction of a string, which may be represented by a class that consists of a variable that references a block of volatile storage and two integers that maintain the size of, and an index into, the volatile storage. The unrecoverable volatile storage is implemented by the class `VolatileStorage` and the recoverable version by `RecoverableVStorage`. The unrecoverable integer is represented by the class `Integer` and the recoverable version by the class `RecoverableInteger`.

Starting with the `RString` class that is implemented using objects which are already recoverable, illustrated in Figure 4.18. As the class declaration given in

```
class RString
{
    RecoverableVStorage storage;
    RecoverableInteger  storage_size:
    RecoverableInteger  storage_index;
public:
    ....    // various operations typical to strings
};
```

Figure 4.18: The class `RString`

Figure 4.18 shows, no extra recovery mechanisms are needed by the class `RString` if all the internal objects are already recoverable. When the contents of the volatile storage are changed, the `RecoverableVStorage` class will create suitable recovery data and management information and add these to the current atomic action. Similarly, when either the size or index changes, recovery data and management information will be added to the current atomic action. If this atomic action aborts then the objects modified will be recovered.

The opposite approach is to construct a recoverable class from unrecoverable objects producing the string implementation illustrated in Figure 4.19. In this

```
class UString : public Object
{
    VolatileStorage storage;
    Integer         storage_size:
    Integer         storage_index;

public:
    ....

    virtual ObjectState* save_state(ObjectState*);
    virtual void         restore_state(ObjectState*);

    ....    // various operations typical to strings
};
```

Figure 4.19: The class UString

class (UString) the recovery technique chosen is the state based approach that is supported by the class Object, so that UString refines the inherited save_state and restore_state operation to save and restore the state of a resulting string object. In the implementation of the save_state and restore_state operations, the state (which consists of the three unrecoverable objects) must be packed into, and unpacked from, a single instance of the recovery data class ObjectState. The disadvantage of this approach is therefore apparent, in that a change to a single unrecoverable object results in all the objects being saved even if they are not modified at the same time.

The final approach is to construct a recoverable class from a mixture of recoverable and unrecoverable objects. Assuming that the RecoverableInteger class is not available, then the string class may be represented in the manner shown in Figure 4.20. The refined versions of the save_state and restore_state operations provided by the URString class simply have to manage the unrecoverable Integer objects, since the volatile storage object is already recoverable.

```
class URString : public Object
{
    RecoverableVStorage storage;
    Integer             storage_size:
    Integer             storage_index;

public:
    ....

    virtual ObjectState* save_state(ObjectState*);
    virtual void         restore_state(ObjectState*);

    ....    // various operations typical to strings
};
```

Figure 4.20: The class URString

Of the three approaches described above, the most useful are the RString and URString classes where the recoverable class consists either entirely, or partly, of instances of recoverable objects. As a result, when an operation such as retrieving a sub-string of the string is invoked then only the index object need be saved. Similarly when the size of a string object is changed then only the volatile storage object and and size object need be saved. If the entire string is changed in an assignment operation however, all three objects will be modified and recovery information recorded for each object. In this situation, the approach of constructing a recoverable class entirely from unrecoverable objects will be the most efficient, as only one set of recovery information is be needed. To optimise recovery for a recoverable class that contains recoverable objects, a fourth approach is possible if the recoverable class inherits extra recovery support from the class Object. Figure 4.21 illustrates such as class. The optimisation that

```
class MRString : public Object
{
    RecoverableVStorage storage;
    RecoverableInteger  storage_size:
    RecoverableInteger  storage_index;

public:
    ....

    virtual ObjectState* save_state(ObjectState*);
    virtual void         restore_state(ObjectState*);

    ....    // various operations typical to strings
};
```

Figure 4.21: The class MRString

inheriting further recovery provides is the ability to recover all three objects

together (as if they were unrecoverable). This capability saves management information (only one copy being required instead of three) and instances of the recovery data object `ObjectState`.

A potential disadvantage with deriving a recoverable class that contains recoverable objects from a base recoverable class is that if all of the recoverable objects are modified, then recovery information will be recorded by the mechanism the recoverable class inherits, in addition to the recovery information that each recoverable object will record as it is modified. Clearly, recording recovery information twice defeats the purpose of this optimisation. To rectify this situation, the solution is to provide a means by which the recoverable class can override the recovery management provided by the recoverable objects contained in the recoverable class.

The method used to override the recovery mechanisms of the internal recoverable objects is shown in Figure 4.22 which illustrates the assignment

```
void MRString::operator=(MRString& oldstring)
{
    Object::modified();
    Recovery = Off;
       storage = oldstring.storage;
       storage_size = oldstring.storage_size;
       storage_index = oldstring.storage_index;
    Recovery = On;
}
```

Figure 4.22: The `MRString` assignment operation

operation operation provided by the `MRString` class. In this implementation, the `modified` operation inherited from `Object` is invoked to save the entire state of the `MRString` object. The implementation of the `modified` operation in turn invokes the `save_state` operation provided by `MRString` for this purpose. The implementation of the `save_state` operation (illustrated in Figure 4.23) simply

```
ObjectState* MRString::save_state(ObjectState* newstate)
{
    storage.save_state(newstate);
    storage_size.save_state(newstate);
    storage_index.save_state(newstate);

    return newstate;
}
```

Figure 4.23: The `MRString save_state` operation

invokes the `save_state` operation provided by each recoverable object to save the state of that object in a single instance of `ObjectState`.

To override the recovery mechanisms the recoverable objects provide, the value of the global variable `Recovery` is set to `Off`. The value of this variable should be checked in each operation that records recovery information with an atomic action. The implementation of the inherited recovery operations (the `modified` operation in the case of `Object` and the `record` operation in the case of `Operation`) act in this manner, returning when invoked without recording any recovery information if the value of `Recovery` is `Off`.

The above approach to controlling when recovery information is recorded has been discussed in terms of the *inclusive* and *disjoint* recovery models proposed by [Anderson *et al.* 78]. When the recovery environment established by a computation is also used by the objects the computation invokes operations on (and any objects that have operations invoked as a result), then the type of recovery is termed *inclusive.* Inclusive recovery is therefore the recovery model implicitly employed throughout the discussion of recovery in this chapter. If the recovery environment of a computation is not employed when an operation is invoked on an object by the computation, then the recovery environments of the main computation and the computation that implements the operation are *disjoint.* An example of how disjoint recovery may be implemented, and when it is required, is therefore illustrated in the assignment operation provided by the `MRString` class, enabling the assignment of the individual recoverable objects to

be considered to be disjoint from the main assignment, thereby avoiding two sets of recovery information from being recorded.

This section has discussed how a recoverable class may be constructed from unrecoverable and recoverable objects. During this discussion, the advantages of the various approaches were briefly described. The next section discusses the advantages and disadvantages of constructing recoverable classes using inheritance in the manner described in this chapter.

4.8 An assessment of constructing recoverable classes using inheritance

This section assesses the technique of exploiting inheritance to construct a recoverable class of objects. To summarise, the basic technique is to derive a new recoverable class from two base classes. One is a previously unrecoverable class and the other a class which provides the recovery mechanisms that enable the newly constructed class to correctly manage the inherited unrecoverable state, and in turn be managed by an atomic action. By exploiting inheritance in this manner, the recoverable class will be a sub-type of the unrecoverable class on which it is based and as a result, instances of the recoverable class may be used in place of instances of the unrecoverable class. In addition, only a single recovery management mechanism is required (for the base recoverable class) since the newly constructed recoverable class will be a sub-type of the base recoverable class.

When the programming language used to construct a class supports the declaration of recoverable objects, then the compiler for the language will produce the recovery mechanisms. With the approach described in this chapter, the base recoverable class provides class-independent support which must be refined by the implementor of a recoverable class to correctly manage that class. To refine the inherited support, the implementor must know the semantics of the

unrecoverable class and, in the case of the state based recovery technique, what constitutes the state of an instance of the unrecoverable class. Without such knowledge, recoverable classes will not correctly manage the unrecoverable state and functionality they inherit. If the state of an unrecoverable class is unknown, but the unrecoverable class provides suitable operations, then a recoverable class can be constructed using the operation based recovery technique which also exploits inheritance.

The principal advantage of an operation based approach to recovery is that the implementation of the recovery mechanisms provided by the recoverable class should be independent of the implementation of the unrecoverable class. For instance, the stack used as an example earlier in the chapter could be implemented as a linked list, and at a later date changed to an array of pointers without affecting the implementation of the recovery mechanisms provided by recoverable class. With the state based approach, recovery is dependent upon the state of the unrecoverable class and as a result changes to the internal representation of the unrecoverable class will effect the state that must be managed by the recovery mechanisms, and must result in changes to the implementation of the recovery mechanisms provided by the recoverable class.

Since the recovery mechanisms are provided by a class that may be inherited, a recoverable class that consists entirely of instances of existing recoverable classes may inherit extra recovery mechanisms to optimise the recovery of the internal recoverable objects. The flexibility offered by providing recovery through a class that may be inherited enables the granularity of recovery to be chosen by the implementor of a recoverable class. An implementor can choose to override the recovery mechanisms provided by a recoverable object so that optimisations are possible when more than one internal object that constitutes the state of the recoverable class is modified. Since recovery mechanisms can be added to existing recovery mechanisms (in the case of a recoverable class that

contains recoverable objects which inherit recovery), and recovery mechanisms can be overridden, employing inheritance to add recovery proves to be very flexible.

4.9 Concluding remarks

When an atomic action is aborted, failure atomicity requires that the objects modified during the atomic action are restored to the state held at the start of the atomic action. This chapter described how the property that supports this capability (recoverability) may be added to a class of objects. The chapter began by considering how recovery mechanisms may be added to an unrecoverable class to provide recoverability. Two alternative recovery techniques were described during this discussion; a state based technique which manages the state of an object, and an operation based technique which records the operations invoked on an object.

To add recovery mechanisms that support either a state or operation based recovery technique, this chapter described a novel approach that exploits the inheritance property of an object-oriented programming language. By producing a class that implements the basic recovery technique (the base recoverable class), a new recoverable class may be constructed by deriving the new class from the base recoverable class and an unrecoverable class. The result is a recoverable class that has the functionality of both the unrecoverable and base recoverable classes. Implementations of both recovery techniques which exploit inheritance for constructing recoverable classes were described.

During the description of each approach, the issues involved in managing instances of the resulting recoverable classes were discussed. The advantage of inheriting recovery is that the resulting recoverable classes are sub-types of the base recoverable class, requiring a single management mechanism for each base recoverable class. This management mechanism was implemented for each base

recoverable class using the record abstraction described in the last chapter. To illustrate the flexibility of exploiting inheritance and the record abstraction, an implementation of a class that provides the abstraction of recovery by performing compensation operations was described.

A later section of this chapter discussed how new abstractions that are recoverable may be constructed from instances of recoverable and unrecoverable classes. The examples discussed in this section illustrated the flexibility of employing inheritance to add recoverability, as the granularity of recovery of internal objects contained in a recoverable class may be defined by the implementor of the recoverable class.

In summary, the technique of using inheritance to construct recoverable classes began partly out of necessity, the alternative being to modify a language compiler or operating system, but the outcome has been a flexible and elegant method of adding a property such as recovery to a previously unrecoverable class of objects.

The next chapter describes how the base recoverable class that implements state based recovery technique may be extended to enable an instance of a class, derived from this base recoverable class, to persist beyond the lifetime of an application that created it.

Chapter 5

Persistence

The previous chapter described how the objects accessed by the computation an atomic action encapsulates may be constructed in such a way that they can be restored to a previously held state should the atomic action abort. This capability is provided by the recoverability of the object and supports the failure atomicity property of the atomic action. This chapter discusses another property of atomic actions, that of *permanence of effect.* The permanence of effect property ensures that the system state established by a successfully terminated top-level atomic action will be unaffected by subsequent system failures. To provide the permanence of effect property, newly established system state must be saved in non-volatile storage which will not be corrupted by system failures. Since the system state is represented by objects this involves ensuring the permanence of those objects.

To move the state of an object to and from non-volatile storage requires a mechanism for mapping the volatile state into, and out of, the form expected by the non-volatile storage system. To simplify the implementation of this mapping mechanism, non-volatile storage may be organised as an *object store* thus providing a suitable interface for the management of objects in non-volatile storage. When a programming system supports *persistence*, the automatic movement of objects to and from an object store, and the mapping mechanisms, are provided for each class of objects.

This chapter describes how the concept of persistence may be incorporated into the object and action model described in earlier chapters of this thesis. The aspects of persistence that will be discussed are those that directly relate to the construction, storage, and naming of a persistent object. Since the purpose of the

work described in this thesis is to address reliability, issues which concern researchers in the field of persistent programming, such as managing different versions of a class of a persistent object (e.g. [Barman and Crawley 87]) or providing database functionality (e.g. [Bloom and Zdonik 87]), are orthogonal to the problem being considered and as a result are not discussed in this thesis.

This chapter begins by discussing when persistence is required and how the implementor of an application may define which of the objects used during the application are existing persistent objects, or new objects that must persist. The following sections discuss how persistence may be added to a class, and how the support for this property may be controlled by the state and access components of an atomic action to move the state of a persistent object to and from the object store. The final sections of the chapter discusses the problems that arise from saving, retrieving, and organising object states in non-volatile storage, and how these issues may be resolved by the design and implementation of an object store.

5.1 Permanence of effect and persistence

The permanence of effect property requires that the system state modified by the computation (encapsulated by an atomic action) becomes permanent when the atomic action commits. In the nested atomic action model employed in this thesis, this occurs when the outermost atomic action (the top-level atomic action) commits, all other (nested) commits simply involve the propagation of management information to the enclosing atomic action.

When a programming system supports *persistence* [Atkinson *et al.* 83a, Atkinson and Buneman 87], an object may be declared that exists beyond the lifetime of the application program in which the object was created. If persistent objects are used to model the permanent system state then the permanence of effect property may be provided by ensuring that all persistent objects, which

have been modified during the top-level atomic action, persist once the top-level atomic action commits.

The motivation behind the concept of persistence is to remove the two views of storage (volatile and non-volatile) supported by conventional programming systems. Normally, an object accessed by an application exists in volatile storage and will be deallocated when the application terminates. In a conventional programming system, if the state the object represents is required to exist beyond the execution of the application then the state must be converted into a form that can be stored in the non-volatile storage supported by the system (which is typically a file).

The aim of a persistent programming language is to remove the burden of mapping the state of an object between volatile and non-volatile forms and provide a uniform view of state where no distinction is made between short and long-term state. Atkinson *et al.* [Atkinson *et al.* 83a] view persistence as an orthogonal and independent property of a class, so that the persistence of an object is not a function of the class of the object or the way the object is used. Systems which support orthogonal persistence are considered to be more flexible than systems that support limited forms of persistence where only certain classes of object may persist, or persistent and nonpersistent objects must be accessed in different ways.

Programming systems that address reliability issues all provide some form of support for the permanence of data, but do not attempt to provide this permanence in an orthogonal manner. For instance, the *Aelous* [Wilkes and LeBlanc 86] language produced by the *Clouds* project [Allchin and McKendry 83] allows a class to be declared as *autorecoverable* which results in the automatic recovery and persistence of the entire state of instances of the class when modified within an atomic action. Another form of declaration supported by Aelous allows

a class to be declared as *recoverable*, but only the variables declared within *recoverable areas* are persistent. In both cases, the persistence of an object is a function of the class of the object, hence, persistence is not an orthogonal property as it is not possible to use an instance of a such a class without it persisting should the top-level atomic action commit.

The research described in this chapter attempts to address persistence from a reliability point of view by building upon the recovery mechanisms described in the last chapter. The penultimate section of this chapter assesses whether the persistence mechanisms that have been developed support orthogonal persistence. The rest of this section however, discusses how persistence may be employed in the object and action model assumed by the environment described in this thesis, beginning with the terminology that will be used to describe persistent objects.

While an application is using a persistent object, the object is said to be *active*. Since the lifetime of a persistent object must extend beyond that of an application, the persistent object must be maintained in non-volatile storage between activations. When a persistent object is stored in non-volatile storage the object is said to become *passive* as it is *deactivated*, the passive form being referred to as the *persistent data*. To *activate* a passive object, a new instance may be created with the default value and the persistent data held in non-volatile storage used to give the new instance the identity and value of the passive persistent object.

The lifetime of a persistent object may be represented by a number of state transitions (illustrated in Figure 5.1). When an object is newly created, the state

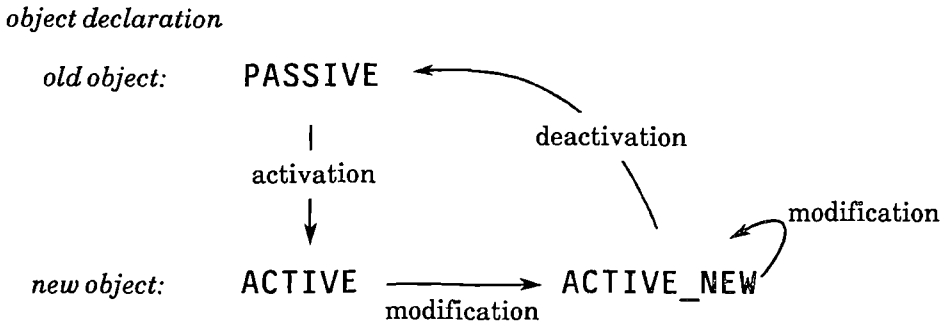

Figure 5.1: Persistent object state transitions

of the new object may be considered to be ACTIVE. When an existing persistent object is declared it is initially PASSIVE so that when the object is *activated* the state will change from PASSIVE to ACTIVE acquiring the old persistent value and identity. As an object is modified the state is assumed to change from ACTIVE to ACTIVE_NEW. If an object has the state ACTIVE_NEW when an application terminates the current volatile object state should be saved in non-volatile storage to ensure the persistence of the object. When an object is saved in non-volatile storage it reverts to the PASSIVE state.

The rest of this section describes how a persistent object may be activated and deactivated, within the framework of the above model. Before invoking operations on an existing persistent object, the object must be active. The activation of the object could occur any time between the declaration of the object in the application and the invocation of the first operation on the object. When an object is created the initialisation operation provided by the class of the object will be invoked, thus providing one method of activating a persistent object. To function consistently within the environment assumed by this thesis however, an existing persistent object must only be activated after concurrency control has been applied. Without concurrency control an object might be activated (in

conflicting access modes) in disjoint applications resulting in inconsistencies when the objects are deactivated.

In the object and action environment, concurrency control is internal to an object, since it is only the implementor of a class who knows which operations modify the internal state of an instance of the class, and hence what type of concurrency control should be placed on an instance. As a result, the implementation of each operation must involve a concurrency control operation. When the first operation is invoked on an object, the first act of the implementation of the operation will be to invoke a concurrency control operation, providing a suitable point at which to activate the object. It is therefore assumed that the concurrency control mechanism is responsible for activating an object. More precise details of the interaction between the persistence and concurrency control mechanisms will appear later in this chapter.

Persistent and nonpersistent objects therefore differ, since access to a persistent object must involve concurrency control as it can be accessed by concurrent computations. Another important difference is that persistent objects do not obey the scope rules normally associated with the programming language used to construct the application in which they are declared. All programming languages have some concept of scope which is defined by an operation, procedure, or block during which the declaration of an object is valid. Once the scope ends all objects declared within that scope may be deallocated since they cease to be accessible.

The way that scope is supported differs from language to language but the basic effect is the same. In a programming system such as Smalltalk [Goldberg and Robson 83] objects are accessed through either temporary or global variables which are explicitly assigned to new or old objects contained on the system *heap*. When an operation or block terminates, the temporary variables declared within

the scope defined by the block, or operation, are automatically deallocated. Objects are never explicitly deallocated, as the language employs a *garbage collector* to deallocate objects when an object is determined to be inaccessible.

The method of object management supported by Smalltalk may be contrasted with that of a procedural object-oriented programming language such as C++ [Stroustrup 86]. The variables declared in C++ can also be temporary or global, but may reference an object in one of two ways. When an object is referenced by an *automatic variable*, the variable and the object will be deallocated when the block, within which the variable was declared, terminates. The compiler for the language provides the support for the creation and deallocation of objects referenced by an automatic variable. The second method of accessing an object is through a *pointer variable*. Objects accessed through a pointer variable function in a similar manner to those supported by Smalltalk as they must be explicitly created and, in the case of C++, deallocated. To create a new instance of a class, C++ provides the *new* operator. To deallocate an existing instance, the *delete* operator is provided. With both types of declaration, the objects are allocated on the program heap, but since the language does not provide a garbage collector the implementor of an application is expected to deallocate objects that are accessed using a pointer variable, before they become inaccessible.

The persistence of an object beyond its normal scope may therefore be achieved by ensuring that either the persistent object is not deallocated, or the persistent data for the object is maintained, until the end of the application (at which point it may be deactivated). The former approach may be implemented by ensuring that all persistent objects are maintained on the program heap and referenced in a global object which is known as the *persistence root*. The *persistence root* can then be processed when the application terminates to ensure all persistent objects are deactivated. An approach of this type is used by the persistent programming language PS-Algol [Atkinson *et al.* 83b]. It is important to note however, that

persistence only affects the state of the object, not the accessibility, since the variable used to access the object will be deallocated.

Deactivation involves saving the persistent data for the persistent object in non-volatile storage. Since this operation is susceptible to node or process crashes, persistent programming systems provide atomic actions to ensure that the deactivation of a persistent object is failure atomic. Generally, only single level atomic actions are supported, so that an atomic action encompasses the application and controls the deactivation of all persistent objects. If the atomic action is not committed then the persistent objects will not be deactivated, so that the deactivation of a persistent object becomes a function of the outcome of the atomic action declared in an application.

If atomic actions can be nested then the persistence of an object becomes a function of the outcome of the atomic action (and any containing atomic actions) within which the object was modified. The management of persistence is therefore more complex when atomic actions are nested, since a single global persistent root cannot be used. The solution is to maintain information about the persistent objects modified during nested atomic actions, and merge this information into the information held by the containing atomic action (in the manner described in the previous chapter for recovery information). If all atomic actions in an application which have control over the persistence of an object commit, then the newly established state of a persistent object will outlive the application.

This section has described how the permanence of effect property of an atomic action may be supported, and how this persistence is a function of the atomic actions declared in an application. The next section describes how the mechanisms that support persistence may be added to a class of objects.

5.2 Supporting persistence

There are a number of issues, which were covered in the last section, that must be considered when constructing mechanisms that support persistence. The first is how to declare that an object should be persistent, or name an existing persistent object so that the persistent data may be located and the object activated. Once active, any scope associated with the object's declaration must be overridden so that the object persists until the outcome of the containing atomic actions are determined. If all atomic actions up to the top-level atomic action commit then the modified persistent objects must be automatically saved in non-volatile storage, requiring mechanisms to move the state of an object (the persistent data) from volatile to non-volatile storage. Finally, the means of organising the non-volatile storage must be considered so that when a persistent object is required by a subsequent application, the persistent data can be located. This section discusses how these issues may be addressed in the object and action environment after briefly considering the approaches taken by other research projects.

To provide support for persistence, most research projects have concentrated on producing persistent programming languages. One approach has been to extend the type system of an existing language to provide new types that support persistence. Examples of this approach are *E*, the language produced by the *EXODUS* [Richardson *et al.* 87] project which is based on C++, and *Trellis/Owl* [Bullis *et al.* 86, O'Brien *et al.* 86]. Another approach, based on extending an existing language, is taken by *PS-Algol* [Atkinson *et al.* 83b] where the mechanisms required are provided as functional extensions to the language *S-Algol* [Morrison 79]. Rather than extend an existing language, new languages have also been defined, for example the *Napier* [Atkinson and Morrison 87] programming language.

To define which objects are persistent, one of two approaches are adopted by these persistent programming languages. Either the class of the object is a special persistent class (as in EXODUS and Trellis/Owl), or any type of object may persist but must be explicitly retrieved from, and placed in, the persistent environment (which defines the objects that are moved to and from the object store - the PS-Algol and Napier approaches). When the persistent programming language is based upon an existing language, the normal scope rules must be overridden. The approach taken by PS-Algol relies on maintaining extra references to a persistent object held in the heap, so that the garbage collector does not deallocate the object.

The mechanisms required to map the state of an object between volatile and non-volatile storage, are generally implemented in one of two ways: by providing general mechanisms that can manage all classes using class structure information (the PS-Algol approach), or by producing the mapping mechanisms specifically for a class during the compilation of the class (the E approach). To manage the state in non-volatile storage an object store is generally employed, which may be constructed specifically for the language (as in PS-Algol), or may be a general purpose object store. An example of the latter is the use of an object store called *ObServer* [Skarra *et al.* 86] by an implementation of persistence in the language Trellis/Owl described in [Moss 87].

The rest of this section discusses an approach to persistence, designed to operate in the object and action environment described in this thesis, beginning with how the recovery mechanisms, described in the last section, may be used as the basis of an implementation of persistence. The mechanisms needed to support persistence are similar to those that support recoverability, when recoverability is based upon saving and restoring the state of an object. Both must be capable of saving and restoring the state of an object, but will differ in where the state is saved to or restored from. For recoverability, the state of an object is usually

maintained in temporary (volatile) storage, whereas with persistence the state of an object is always maintained in permanent (non-volatile) storage.

Given the support for retrieving and restoring the state of an object, supplied by the recovery mechanisms, a persistent class of objects may be constructed by ensuring that the state of an object is saved in non-volatile rather than volatile storage. The additional functionality required is described in the rest of this section.

The last chapter described how a recoverable class may be constructed and how instances of the class that are declared in an application which employs atomic actions are restored when the atomic action aborts. In this implementation, the declaration of an instance of the recoverable class produces a recoverable object. A similar approach could be taken by persistent objects, in that the declaration of an instance of a persistent class results in the persistence of that object when the top-level atomic action commits, the effect being equivalent to the creation of a file in a conventional programming (or operating) system. This effect is undesirable however, since all instances of a persistent class will persist by default. It should therefore be possible to declare an instance of a persistent class in a manner that ensures it is either persistent or nonpersistent, with the default being a nonpersistent object.

Another difference between the persistence and recovery mechanisms lies in the data required by each, and the way that the data is used. If the state of an object is unstructured, for example a simple contiguous character buffer, then the persistent and recovery data for the object will be identical (a copy of the buffer). If the state of an object is structured into a number of other (internal) recoverable/persistent objects however, then the data required by the recovery and persistence mechanisms will differ, as will the mechanisms.

Consider the recoverable/persistent object A shown in Figure 5.2, which

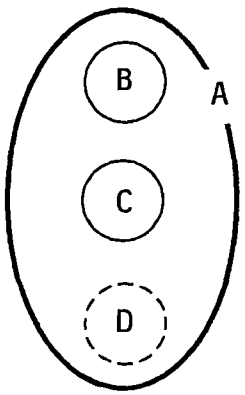

Figure 5.2: A structured object

contains two internal recoverable/persistent objects (B and C) and one internal unrecoverable object (D). When either B or C are modified (assuming the *inclusive* recovery model described in the last chapter), then the recovery of both objects may be managed independently of A. The only time the object A has to create recovery data and record management information is when the internal object D is modified, since D is unrecoverable. When persistence is considered, a similar approach may appear to be sufficient, in that the persistence of B and C are managed independently of A. Since B and C are only accessible through A however, there would be no way of activating B and C when A is activated by a subsequent application. To ensure the persistence of the entire object therefore requires that the names of B and C are recorded in the persistent data for A. Hence the persistent data for A will differ from the recovery data for A. This difference between recovery and persistent data highlights the fact that a recoverable class can be implemented by ensuring that all internal objects are recoverable, whereas a persistent class must always provide additional mechanisms to manage the internal objects, even if all the internal objects already support persistence.

In addition to the differences in data, the persistence mechanisms must handle the creation of existing internal persistent objects in a different manner to new internal persistent or nonpersistent objects. In the case of the above example, when A is newly created, the internal persistent objects (B and C) may be created by the initialisation operation implemented by the class of the object A. When A is an existing persistent object however, the creation of the internal persistent objects should not occur until the state of A is restored using the persistent data. The reason being that it is only during the restoration of A that the names of the internal persistent objects will be known, and the names must be known before an existing persistent object is created since the identity and persistence of an object is determined when it is created. As a result, the initialisation operations provided by a persistent class for creating persistent instances will differ from those that create nonpersistent instances.

To summarise, persistence can be considered to be an extension to recoverability requiring that the state of an object is saved in non-volatile rather than volatile storage. Under certain circumstances, the persistent data and mechanisms may differ from the recovery data and mechanisms, and persistent objects must be declared in a special manner. The next section describes how the recovery mechanisms provided by the class `Object` may be extended to support persistence.

5.3 Implementing persistence

This section describes how the state based recovery mechanisms, implemented by the class `Object`, may be used as the basis for new functionality to enable instances of a class to persist. The approach taken in this implementation employs the class that is used to hold the recovery data (`ObjectState`) also to hold the persistent data. By adding extra operations (`activate` and `deactivate`) to the class `Object`, and implementing an object store (as the class

`ObjectStore`) that manages instances of `ObjectState`, the support for constructing a persistent class of objects may be provided.

To control the persistence of an object in an atomic action environment another *record* class (`PersistentRecord`) may be implemented. In effect, the atomic action acts as the *persistence root*, with each `PersistentRecord` instance referencing an active persistent object. During the commitment of the top-level atomic action each `PersistentRecord` will deactivate the persistent object it references.

A previous section described how one of two approaches may be taken to override the scope of the programming language. Rather than rely on the persistent objects being kept on the program heap (the approach adopted by PS-Algol), the implementation described in this section takes the alternative approach. To guarantee the persistence of an object only requires that the persistent data is available when the top-level atomic action commits. By providing a means of saving the persistent data before the persistent object is deallocated, and a special *record* class to manage the persistent data, the scope defined by the language may be overridden. Instances of the special *record* class (`CadaverRecord`) are created to hold the persistent data, immediately before the object is deallocated. Each `CadaverRecord` only takes part in the top-level commit of an atomic action, saving the persistent data in the object store. If an atomic action is aborted then the `CadaverRecord` instances will be deleted, achieving the desired result and persistence. The rest of this section, beginning with the extensions to the class `Object`, describe in greater detail the implementation of these classes which operate in the atomic action environment described in chapter three.

To recap, the class Object provides an operation called save_state that returns the state of an object as an instance of the class ObjectState. The corresponding operation is called restore_state which restores the state of an object using the ObjectState instance passed as an argument. To manage the recovery data (implemented by the ObjectState class), and perform recovery on an object, a *record* class called ObjectStateRecord is provided. To uniquely name objects the class Uid is provided. An instance of this class is declared in the class Object so that all recoverable/persistent objects may be named in a uniform manner using the value of the Uid instance. To read the value of this variable, Object provides the operation get_Uid. One example of the use of unique identifiers is to determine whether two instances of the class ObjectStateRecord refer to the same object.

Given the unique identifier contained in the state implemented by the class Object, a method of defining whether an object should persist is required. The solution chosen in this implementation is to provide an operation called persist. Invoking this operation changes the type of the object from recoverable (the default) to recoverable/persistent. To represent the type of an object, a variable called Type of type Object_Type is defined to be part of the state of Object. An invocation of the persist operation changes the value of Type from RECOVERABLE to PERSISTENT. The persist operation also takes an argument which is an instance of the class Uid and if this argument has a value then the object is an existing persistent object, and the Uid instance maintained by Object is set to the value of the argument. The method of naming persistent objects is therefore based upon the value of the unique identifier maintained by the class Object. Each class ultimately derived from Object may provide class-specific methods of defining whether an object is persistent or not, as long as the persist operation is invoked by the constructor when an instance of the class is persistent. It is also possible, given this approach, to define a class that always

results in the persistence of instances of the class by invoking `persist` in all the constructors the class provides.

To directly support a persistent interface, the class `Object` has been extended by adding two new operations: `activate` and `deactivate`. A more complete declaration (than that given in Figure 4.5) of the class `Object` is shown in Figure 5.3. In addition to `activate` and `deactivate`, three of the objects that

```
enum Object_Status {ACTIVE, PASSIVE, ACTIVE_NEW};
enum Object_Type   {RECOVERABLE, PERSISTENT};

class Object
{
    Object_Status Status;
    Object_Type   Type;
    Uid           object_uid;
    ....

protected:
    void modified();
    void persist(Uid*);

public:
    Object();
    ~Object();
    ....

    virtual ObjectState* save_state(ObjectState*,Object_Type);
    virtual void         restore_state(ObjectState*,Object_Type);

    Uid* get_Uid();

    Outcome activate();
    Outcome deactivate();
    virtual void destroy();
};
```

Figure 5.3: The class `Object`

represent the state of an `Object` instance are illustrated. The first is a variable of type `Object_Status`, the value of which follows the state changes described in an earlier section of this chapter. The second is of type `Object_Type` which is initialised to `RECOVERABLE` but may be changed to `PERSISTENT` when the `persist` operation is invoked. The third variable (`object_uid`) is the `Uid` object used to name all objects which has already been mentioned.

The new operations, `activate` and `deactivate`, each require access to an object store to save or restore the persistent data. An object store is assumed to be implemented by the class `ObjectStore` (see Figure 5.12) which provides two

operations that save and return instances of the class `ObjectState`, called `write_state` and `read_state` respectively. To simplify issues, there is always assumed to be a global object store which may be accessed through the variable `Current_Store`. Figure 5.4 illustrates how the operations provided by the

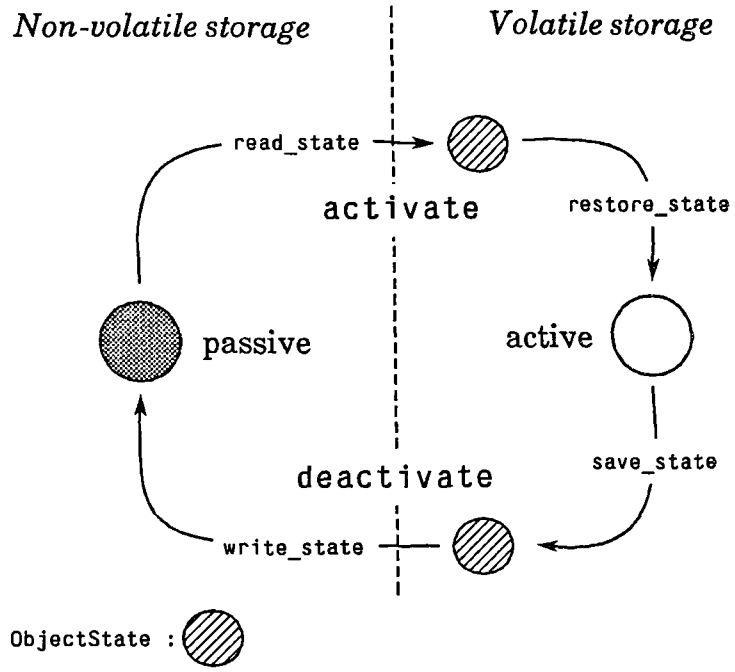

Figure 5.4: Movement of persistent data

classes `Object` and `ObjectStore` move the state of a persistent object between volatile and non-volatile storage.

The implementation of the `activate` operation (illustrated in Figure 5.5)

```
Outcome Object::activate()
{
    if (Type == PERSISTENT && Status == PASSIVE)
    {
        ObjectState *oldstate = Current_Store->read_state(&object_Uid);
        if (oldstate)      // was a valid ObjectState returned ?
        {
            restore_state(oldstate,PERSISTENT); // restore the old state
            Status = ACTIVE;
            return(SUCCESS);
        }
    }
    return(FAILURE);
}
```

Figure 5.5: The `Object activate` operation

shows how the read_state operation provided by the ObjectStore (which takes a unique identifier as an argument) may be used to return the persistent data for an existing persistent object. The implementation of the complementary operation deactivate is illustrated in Figure 5.6.

```
Outcome Object::deactivate()
{
    if (Type == PERSISTENT && Status == ACTIVE_NEW)
    {
        ObjectState newstate;
        if (save_state(&newstate,PERSISTENT))    // get the persistent data
        {
            if (Current_Store->write_state(&newstate))   // storage successfull ?
            {
                Status = PASSIVE;
                return(SUCCESS);
            }
        }
    }
    return(FAILURE);
}
```

Figure 5.6: The Object deactivate operation

During the activate and deactivate operations, the state of a persistent object is saved and restored using the Object operations save_state and restore_state. Since the data and mechanisms required for recovery and persistence may differ (as described in the last section) the save_state operation must return, and the restore_state operation must expect, persistent data when invoked by the activate and deactivate operations to function correctly. Rather than provide two operations for recovery and two for persistence, the class Object uses the save_state and restore_state operations to perform both functions, relying on the value of a second argument (of type Object_Type) to determine which type of data to expect or return and corresponding mechanisms to execute.

To support the automatic deactivation of an instance of a persistent class when a top-level atomic action commits requires additional functionality beyond that provided by the class Object. Chapter three described how the atomic action implementation provide by the class AtomicAction maintains instances of

record classes (classes derived from the class AbstractRecord). When an operation such as Commit or Abort is invoked on an AtomicAction instance, the implementation of the AtomicAction invokes the corresponding (top-level or nested) operation on all the record instances it holds. For each recoverable object that is an instance of a class derived from Object, there will be an instance of the ObjectStateRecord class in the atomic action. The only operation this class implements is the abort operation, which simply invokes restore_state, provided by Object, to recover the state of an object using the recovery data maintained by the ObjectState instance.

To support persistence therefore, a new *record* class is needed. This *record* class, called PersistentRecord, is derived from ObjectStateRecord to provide the functionality associated with a recoverable object. To control persistence, the PersistentRecord class refines the operations that are invoked when a top-level commit occurs. The implementation of the PersistentRecord takes an optimistic view of the commit operation. The first phase of the two-phase commit protocol results in a *top_level_prepare* event, so that the atomic action invokes the AbstractRecord top_level_prepare operation for each AbstractRecord in the action. In the implementation of this operation provided by PersistentRecord (illustrated in Figure 5.7), the

```
int PersistentRecord::top_level_prepare()
{
    if (object_addr->deactivate() == SUCCESS)
        return (1);    // deactivating the object suceeded
    else
        return (0);    // deactivating the object failed
}
```

Figure 5.7: The PersistentRecord top_level_prepare operation

persistent object is deactivated by directly invoking the deactivate operation using the reference to the persistent object called object_addr maintained by the PersistentRecord. The operation is optimistic, since the second phase of the two-phase commit protocol need not perform any extra work. If the first phase

fails, due for instance to the object store failing to save the persistent data resulting in the `deactivate` operation returning `FAILURE`, then for all the objects for which the operation succeeded, the new state must be removed from the object store. For each `PersistentRecord` this is the function of the `top_level_abort` operation which invokes an operation called `delete_state` (provided by the `ObjectStore`) to remove the `ObjectState` saved by the `write_state` operation.

The `ObjectStore` implementation employed by the above commit protocol is implicitly assumed to maintain versions of an object's persistent data so that compensation may be performed by removing the last version added. One approach is to store the versions of the persistent data as a *log*. An alternative approach is to provide a form of *careful replacement* [Verhofstad 78]. This could take the form of an operation to add the persistent data for an object, with the second phase of the protocol determining whether the old or the new persistent data is removed. To simplify issues however, the approach described above will be employed when the design and implementation of an object store is detailed in later sections of this chapter.

The last chapter described how the first time `modified` is invoked in an atomic action, an instance of the `ObjectStateRecord` class is created, and added to the current atomic action, to hold the recovery data. To support persistence therefore, `modified` must be enhanced so that the value of the `Object_Type` variable maintained by `Object` (which will be either `RECOVERABLE` or `PERSISTENT`) defines whether an `ObjectStateRecord` instance or `PersistentRecord` instance is added to the current atomic action.

The remaining component required to support persistence is the mechanism to override the scope imposed by the programming language. Depending upon the

language, an object may be deallocated when the number of references to the objects are determined to be zero or when the scope ends.

If the language supports the former approach then the `PersistentRecord` created during the invocation of the `modified` operation will be sufficient to ensure that at least one reference to the object is kept until the atomic action terminates. Hence, the object will not be deallocated.

When an object declared in C++ (which supports the latter approach) is deallocated, the *destructor* for the class of the object is invoked. To effectively override the deallocation of the persistent object, the class `Object` provides an operation called `terminate` which may be invoked in the destructor. The `terminate` operation creates an instance of the *record* class `CadaverRecord` and invokes the `save_state` operation to create the persistent data (an instance of `ObjectState` containing the current state of the object). The `CadaverRecord` is then added to the current atomic action (replacing any `PersistentRecords` for the object), thus ensuring that the persistent data for the object is available when the atomic action terminates. To ensure the persistence of an object, the `top_level_prepare` operation implemented by the `CadaverRecord` class must operate in a similar manner to the `deactivate` operation, directly placing the persistent data held by the `CadaverRecord` in the object store (using the `write_state` operation).

The remaining operation that `Object` provides, which is of relevance to persistence, is the operation `destroy`. The function of this operation is to remove an existing persistent object from the object store. Since the object store already provides an operation called `delete_state` (detailed during the description of the commit protocol implemented by `PersistentRecord`), the `destroy` operation may appear to be unnecessary. The reason for providing this operation however, is the same as the reason why the persistent data for an object may

differ from the recovery data. As the last section explained, an object may contain other internal objects. If an internal object is persistent then the containing object need only save the name of the persistent object as part of the containing object's persistent data. If the persistent data for an object of this type was simply deleted from the object store then the persistent data for the internal persistent objects would be left in the object store.

To solve this problem therefore, the operation `destroy` is implemented by the class `Object`. If an object is a structured object that references other persistent objects then the class for the object must refine the `destroy` operation so that the persistent data for the internal objects are destroyed when the containing object is destroyed. In the case of the example given earlier (Figure 5.2), the class of the object `A` would be responsible for invoking the `destroy` operation on the objects `B` and `C` in addition to invoking the inherited `destroy` operation to remove the persistent data for `A` (which contains the state of the object `D`). If the class is unstructured then the default implementation provided by `Object` may be used.

To guarantee that the `destroy` operation is recoverable, the `delete_state` operation is not directly invoked by the `destroy` operation. Instead, the `destroy` operation creates an instance of a *record* class called `DeleteRecord` for each internal object, and the containing object, and adds these instances to the current atomic action. The only operation implemented for this *record* class is the `top_level_commit` operation. Since this operation is only invoked during the second phase of the commitment of a top-level atomic action, the `ObjectState` instance for the object the `DeleteRecord` is managing may be safely deleted. If any atomic action before the top-level action is aborted then the `DeleteRecord` will be discarded, thereby recovering the `destroy` operation.

This section described how and when an object becomes persistent. The next sub-section describes the implementation of a simple persistent class to illustrate how easily a persistent class may be defined. In addition, the support that a class must provide to ensure that instances of the class are persistent is contrasted with the support required to ensure recoverability.

5.3.1 A simple persistent class

This section describes the design and implementation of a simple persistent class that provides the abstraction of a file. The class, which is called `File` (illustrated in Figure 5.8), provides two operations (`read` and `write`) that take

```
class File : public Object
{
    ....
public:
    File();
    File(String);
    ~File();

    Integer read(Buffer*,Integer);
    Integer write(Buffer*,Integer);

    virtual ObjectState* save_state(ObjectState*,Object_Type);
    virtual void         restore_state(ObjectState*,Object_Type);
};
```

Figure 5.8: The persistent class `File`

instances of the class `Buffer` (a class that implements a character buffer).

In a C++ class, when an instance is declared with no arguments the default constructor is invoked. To provide the capability of declaring persistent and nonpersistent instances of the `File` class, the constructor is *overloaded* by providing a constructor that takes no arguments and one that takes a `String` argument. When an instance of the `File` class is declared with no arguments, the default constructor will be invoked. The implementation of this constructor only initialises the `File` object. To declare a persistent instance of the class `File`, an instance of the class `String` (a character string) is needed.

To map the `String` passed as an argument to the unique identifier needed by the class `Object` to name the persistent data, the `File` constructor (illustrated in Figure 5.9) employs a name server which is assumed to be implemented by the

```
File::File(String string)
{
    Uid *id = Current_NameServer->lookup(string);
    persist(id);
    if (id == 0)                          // is this a new object ?
    {
        Current_NameServer->add(string,get_Uid()); // yes so add string and Uid
        .... // create any internal persistent objects
    }
    // else leave the creation of internal persistent objects to restore_state
    .... // rest of the initialisation for this class
};
```

Figure 5.9: The `File` constructor

class `NameServer`, an instance of which is accessible using the global variable `Current_NameServer`. The `lookup` operation implemented by the class `NameServer` returns the unique identifier associated with the `String` if the `String` instance is held in the name server. If the `String` is not found then the object is new, so the `String` and the `Uid` for the `File` objects are added to the name server (using the `NameServer add` operation). The `NameServer` class is also assumed to be a recoverable/persistent class, so that if the containing atomic action is aborted, the new `File` instance will not be added to the object store and the `String` to `Uid` mapping will be removed from the name server. In addition, if the object is an existing persistent object then the constructor does not create any internal persistent objects, leaving this to the `restore_state` operation which will be invoked when the object is activated using the persistent data.

This example illustrates the functionality a persistent class must provide in addition to that required by a recoverable class. These additions are a constructor that invokes `persist` and leaves the creation of internal persistent objects to the `restore_state` operation if the object is an existing persistent object, and refined versions of the `save_state` and `restore_state` operations that are

capable of returning persistent data or restoring the object (and creating any internal persistent objects) using the supplied persistent data. Given the extensions to the `Object` class described in the previous section, these are the only differences between a class which is declared so that all instances are recoverable and a class which is declared so that an instance may be persistent.

This section described how the persistence mechanisms may be used to construct a simple persistent class. To support persistence an object store is required. The following section discusses the design of an object store which best reflects the organisation of classes provided by the programming system described in this thesis.

5.4 The design of an object store

During the description of the implementation of the mechanisms that support persistence in the last section, the assumption was made that the data for a persistent object could be saved in, and restored from, non-volatile storage. Furthermore, it was assumed that the non-volatile storage was organised as an object store implemented by the class `ObjectStore` which provides three operations: `read_state`, `write_state`, and `delete_state`. This section describes the design of the `ObjectStore` class, concentrating on the logical rather than physical organisation of the underlying non-volatile storage.

Since an object store must be capable of managing the persistent data for a variety of different classes of object, a class-independent storage format is needed. Earlier sections of this chapter described how the persistent data may be managed in a class-independent manner using the `ObjectState` class, so that the `ObjectStore` implementation need only manage instances of this class.

It is assumed that persistent data is only added to the object store during the commitment of a top-level atomic action. To ensure the atomicity of the commit operation, the atomic action implementation employs a two-phase commit protocol which saves management information in the object store so that a node crash may be tolerated, and a consistent system state established when the node restarts execution.

The non-volatile storage used by the object store is also assumed to be implemented as *stable storage* [Lampson and Sturgis 76]. A stable storage system uses replicated hardware and carefully designed fault tolerance strategies to provide the abstraction of non-volatile storage which has a high probability of remaining uncorrupted despite media or node failures. The object store therefore, does not require any special reliability mechanisms for managing the addition of persistent data, since the atomic action implementation is responsible for ensuring that any inconsistencies, which would occur if the commit operation was interrupted, are recoverable. To discover and recover any inconsistencies, an application that implements a *garbage collector* is assumed to be available and invoked when a node recovers from a node crash. Given these assumptions, the rest of this section concentrates on the design of an object store, beginning with the organisation of the persistent data so that location and access is both fast and efficient.

The simplest organisation is a flat name space that maps an object's unique identifier (the means by which all objects are assumed to be named) to the location of the persistent data for the object in non-volatile storage (illustrated in Figure 5.10). This location would be implementation dependent and could be a block number on a hard disk, an address in non-volatile memory, or an index into a sequential file. In this organisation, if an object has internal persistent objects

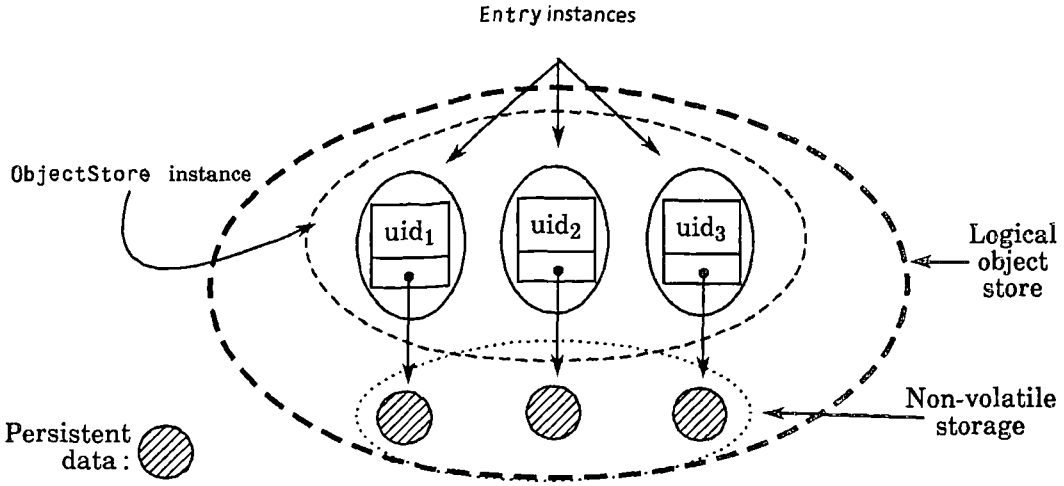

Figure 5.10: A single level object store

then the persistent data for the containing object must contain the unique identifiers of the internal objects.

To manage the mapping from unique identifier to non-volatile storage location, a class such as `Entry` (Figure 5.11) may be defined. For each persistent

```
class Entry : public Object
{
    Uid* object_uid;
    Location data_locn;

public:
    Entry(Uid*,Location);
    ~Entry();
    ....
};
```

Figure 5.11: The class `Entry`

object, an instance of `Entry` may be used to maintain the value of the object's `Uid`, and the location of the persistent data for the object in non-volatile storage (the type `Location` being implementation dependent).

To provide the overall abstraction of an object store the `ObjectStore` class (illustrated in Figure 5.12) may be defined to manage the location of each `ObjectState` object using an instance of `Entry`. Figure 5.10 illustrates an active instance of the class `ObjectStore` which is maintaining the persistent

```
enum store_type {ROOT,USER};

class ObjectStore : public Object
{
    Location store_locn;
    Integer  no_entries;
    Entry    **entries;
    ....

public:
    ObjectStore(store_type =ROOT);
    ObjectStore(String);
    ~ObjectStore();

    ObjectState* read_state(Uid*);
    Outcome      write_state(ObjectState*);
    void         delete_state(Uid*);

    virtual ObjectState* save_state(ObjectState*,Object_Type);
    virtual void         restore_state(ObjectState*,Object_Type);
};
```

Figure 5.12: The class `ObjectStore`

data for three objects using instances of the class `Entry`. When an object store is not being used, the mappings of unique identifier to location for the persistent data in non-volatile storage must also persist. Hence, the `ObjectStore` class must also be a persistent class. The persistent data for an `ObjectStore` instance may be organised in one of two ways: as the unique identifiers of the `Entry` objects (since each `Entry` object may also persist), or as the persistent data for all the `Entry` objects (by declaring the `Entry` objects to be nonpersistent and using the `save_state` operation to retrieve the state of each `Entry`). The latter approach has been chosen since it is more efficient, requiring fewer non-volatile storage accesses.

Since instances of the `ObjectStore` class are persistent, they should be activated and deactivated using an `ObjectStore` instance. Given this approach however, the activation of the `ObjectStore` instance will require an `ObjectStore` instance to locate the persistent data for the `ObjectStore`. To break this circularity, the `ObjectStore` may be implemented so that the persistent data for the object store can be recognised as such, and saved in a location known to the `ObjectStore` implementation. In this way, an

`ObjectStore` instance may be activated without the need for an `ObjectStore` to locate the persistent data for the `ObjectStore` instance.

Since objects stores are simply instances of the class `ObjectStore`, more than one object store may be declared. Rather than attempt to define the location of all possible object stores in the `ObjectStore` implementation, the distinction may be made between a single *root* object store and all other *user* object stores. The location of *root* object may be defined by the `ObjectStore` implementation, but the *user* object stores may be treated as just another persistent object. To define the type of object store when an instance of the class `ObjectStore` is declared, the constructor for the class takes an argument of type `store_type` (illustrated in Figure 5.12). To locate a *user* object store therefore involves activating and using the *root* object store to locate and access the persistent data required for the *user* object store.

In the design presented above, if the object store contains many objects then its state will consist of a large number of `Entry` objects, which will affect both the speed to locate the `Entry` for a given `Uid`, and the time taken to save the persistent data for the object store when a new object (and hence `Entry`) is added. To improve the ease with which the `Entry` for a persistent object is found, an alternative approach may be employed, utilising the class of the object in the organisation of persistent data in the object store to provide a two level name space. In this way the class of the object will locate a flat name space that contains the `Entry` objects for all the persistent data of that class.

When there are more instances of classes than classes, an organisation of the type described above will reduce the number of `Entry` objects that must be searched to locate the persistent data for an object of a given class. An added advantage of the above approach is that since the class of an object must be used to locate the persistent data, it will not be possible to subvert the language's type

system by activating an instance of a class using the persistent data for a different class. To function however, a means of naming the class of an object is required. To simplify issues, it will be assumed that a constant called `CLASSNAME` is available which consists of a string with the same name as the class of the object.

To locate the persistent data for a persistent object given the class name and unique identifier, the `Entry` objects for a particular class must be located and searched until the unique identifier of the object is found. To manage the extra level of indirection that the class name adds, another class could be defined. However, this is not necessary since the root object store can contain the location of other object stores. Each class may therefore be represented by an object store, with the root object store containing the `Entry` object for each class's object store. In this situation, the root object store will be the object store for the class `ObjectStore`. An organisation of this type is illustrated in Figure 5.13. In this

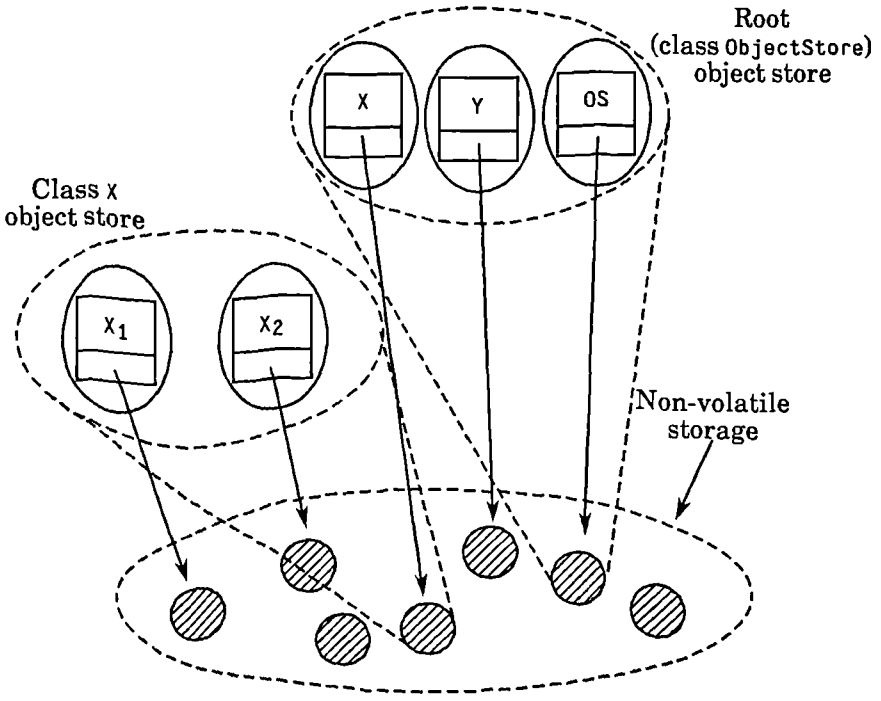

Figure 5.13: A two-level object store

figure, there are two active `ObjectStore` instances, the root (or `ObjectStore`)

instance labeled OS and the instance for a class X. The root object store contains the Entry objects for two classes (X and Y), in addition to the Entry object for the root object store. The active object store for the class X also contains two Entry objects which locate the persistent data for the objects X_1 and X_2. The two unreferenced sets of persistent data in the above figure are those for two instances of the class Y, the references to which are contained in the persistent data of the object store for the class Y.

This section has described how the abstraction of an object store may be structured into a collection of object stores that each maintain the persistent data for instance of a particular class. The next section describes how this object store design may be implemented using the support of the operating system UNIX.

5.5 Implementing an object store

This section describes a simple implementation of the object store design presented in last section. The aim of this implementation was to test the soundness of the ideas presented, and the abstractions developed, in this thesis, rather than provide the definitive object store implementation. To this end, the object store has been implemented on top of the UNIX operating system. Consequently, the performance of the implementation could be greatly improved by alternative (lower-level) implementations. The advantages of the abstraction developed using an object-oriented language however, will ensure that subsequent object store implementations can be used by the rest of the programming system described in this thesis (providing that the interface defined by the class ObjectStore is met).

This implementation provides no tolerance against media failures, and it is assumed that previous work at Newcastle [Anyanwu 85, Anyanwu 86] which extended the *stable storage* mechanisms described in [Lampson and Sturgis 76] to provide the abstraction of a reliable, crash resistant, UNIX file system could be

employed at a later date. Anyanwu utilised standard magnetic disk technology, but there are other approaches to the implementation of non-volatile storage which is tolerant to media failures. One example is the stable storage implementation provided by the Enchère system [Banâtre *et al.* 86] which uses stable random access memory instead of magnetic disk technology.

There are a number of different approaches to implementing the two-level object store design described in the last section using the support provided by UNIX. For instance, each object store could be implemented as a single UNIX file, with the `Location` of the persistent data being the offset from the beginning of the file. Whilst this approach is simple, as the number of persistent objects grow the size of the file will increase, which may result in the time taken for an object to be located being dependent upon when the object was created. A superior approach would be to employ the directory structure supported by the UNIX file system.

The UNIX file system is hierarchical in nature, with a directory containing files and other (sub-)directories. Using the directory structure, an object store may be represented by a directory, with each entry in the directory being an instance of the class which the directory represents. In this way, separate `Entry` objects are not required since an entry in a UNIX directory is equivalent. Each entry in a directory therefore maintains the location (or name) of a file which contains the persistent data for an object. To name each file, the unique identifier may be used (when converted into a string). To name each directory, the `CLASSNAME`, which is a string that names the class of an object, may be used. Figure 5.14 illustrates three classes (and directories) that contains the names of a number of instances of each class.

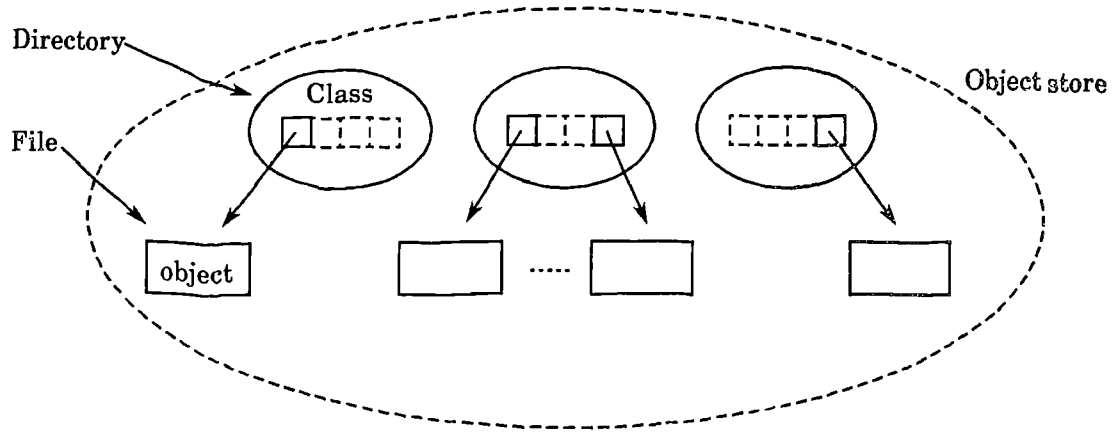

Figure 5.14: Object store implementation

This implementation differs from the design presented in the last section since a distinction is made between `ObjectStore` objects (which are UNIX directories) and other types of object (which are UNIX files). To provide tolerance against corruption due to a node crash during the update of the persistent data for an object, the persistent data is stored as a log of versions. Each version contains redundant information that enables the integrity of the data to be determined (this information would not be required if the non-volatile storage was implemented as stable storage).

Given that a directory is used to represent a class, and the unique identifier converted into a string is the name of the file that contains the persistent data for that object, then all the directories may in turn be contained in a single directory which is the object store. This directory is a constant defined in the `ObjectStore` implementation, so that to locate the file for a persistent object, a pathname may be constructed from the `ObjectStore` constant with the classname, followed by instance name, appended. The object store implementation described in this section is currently in use, and is one area where the performance of the programming system described in this thesis may be improved. The next section describes how the concurrency control mechanisms

that may be inherited from the class LockCC affect the persistence of objects and the object store design.

5.6 Concurrency control and the object store

Apart from a brief mention of the class LockCC during the description of Arjuna in chapter two and a discussion earlier in this chapter, the issues involved in concurrency control have not been discussed. In Arjuna, concurrency control is provided by the class LockCC [Parrington 88], so that a class derived from LockCC will be able to utilise inherited operations to set locks on its instances. The default locking scheme provided by LockCC is the well known pessimistic strategy of *single writer/multiple readers* which allows an application exclusive access to an object if a write lock is set.

If an object is newly created its existence will not be known to other users until it is saved in the object store. When a persistent object is activated, the concurrency control mechanisms come into play, ensuring that only a single instance of the object may be activated if a write lock is set. As the previous discussion outlined, the activation of an object (involving the retrieval of the object's persistent data from the object store and the recreation of the persistent state using the persistent data) must not occur until a lock has been set on the object. If this discipline is not adhered to, multiple versions of the object may be in existence, producing inconsistencies when the persistent data in the object store is updated.

To assist the concurrency control mechanisms the class Object provides an operation called activate (see Figure 5.5). The first time this operation is invoked, the persistent data for the object is retrieved from the object store and the state established. Subsequent invocations return immediately since the internal state of the object will have changed from PASSIVE to ACTIVE. The

operation provided by `LockCC` which sets a lock on the object (`SetLock`), only invokes the `activate` operation once a lock as been granted.

When a class is derived from the class `LockCC`, and the support this class provides for setting locks employed, the concurrency control provided by `LockCC` will ensure that two instances of the same object are never created and accessed in conflicting access modes. Since the concurrency control is a part of the object and cannot be overridden, two versions of the same object will not be available for modification. As there will only ever be one version of an object available for modification, there need be no access control on the object store.

Since the object store is itself a persistent object however, concurrency control should be utilised to ensure that modifications to the object store, such as deleting the persistent data for an object, do not occur concurrently with other operations on the same part of the object store. If the class `ObjectStore` was derived from `LockCC` it could utilise the pessimistic locking provided by this class, but the degree of granularity provided by `LockCC` would be overly restrictive. For instance, deleting an object would involve locking the `ObjectStore` instance that represents that object's class using a write lock. Such a lock would restrict access to all instances of that class, even though an entirely different instance may be required.

To solve this problem *type specific locking* is required which reduces the granularity of a lock enabling an entry in an `ObjectStore` to be locked, rather than the entire object store. An implementation of type specific locking for directory type structures is described in [Parrington and Shrivastava 88]. It is intended that this approach will be utilised by the `ObjectStore` class to increase the amount of concurrency in subsequent implementations.

5.7 An assessment of constructing persistent classes using inheritance

This chapter has described how the support required to ensure an object is recoverable can be extended to guarantee the persistence of the object. This section assesses this technique, discussing the advantages and disadvantages of this approach. To recap, by extending the interface provided by the class `Object`, and implementing an object store that manages the storage of persistent data, new classes of object may inherit the basic functionality required to ensure that instances of these classes persist when accessed within an atomic action. The operations provided by the class `Object` ensure that a persistent object will be automatically activated and deactivated. Activation is controlled by the concurrency control mechanisms, deactivation by a instance of the *record* class which is added to an atomic action the first time a persistent object is modified.

The principal advantage of the implementation of persistence described in this chapter is that it builds upon the support provided to ensure the recoverability of an object. A class, derived from the base recoverable class `Object`, that implements the operations required to ensure that instances of the class are recoverable, needs little extra functionality to ensure that instances also persist. When a persistent class is constructed in this manner, the property of persistence is independent of the way that the objects are used. For individual instances of a class to persist however, the class must be derived from the class `Object` so that persistence is dependent upon the class of an object. As a result, the implementation of persistence described in this chapter is not orthogonal.

Another advantage of this implementation is the flexibility in the ways of constructing persistent classes from instances of other persistent classes. For example, an instance of a class (the containing object) can instantiate other persistent classes (the internal objects) in such a way that they persist when

modified, with the containing class saving the names of the internal objects as part of its state. This is the situation described earlier in the chapter when illustrating the possible differences in recovery and persistent data.

An alternative implementation of the class of the containing object could declare the internal objects to be nonpersistent. To ensure that the internal objects do persist however, the containing object may use the operations provided by `Object` (`save_state` and `restore_state`) to directly save and restore the state of the internal objects, holding the persistent data for the internal objects in the persistent data for the containing object. The main advantage of this approach is that the number of non-volatile storage accesses is decreased, thereby increasing the performance of the activation and deactivation, of a group of objects.

Another alternative is also possible, where the internal objects are declared to be persistent, but one or other of the two approaches described above effectively occurs, with the decision which being determined dynamically during the execution of operations on the containing object. This approach is possible using the mechanism (described in the last chapter) to override the recovery of internal objects. By switching recovery off, the `PersistentRecord` objects will not be created so that the internal objects will not persist, leaving the containing object to manage the persistence of the internal objects. If recovery is not switched off then the first of the above approaches to the persistence of the internal objects will occur.

The next chapter describes an example that consists of a number of classes that may be managed in this manner. To illustrate how the performance may be increased by managing both the recovery and persistence of the internal objects as a part of the containing object, a number of simple performance tests are described.

The disadvantages of this implementation of persistence, apart from not meeting the requirement for orthogonal persistence, is that the implementor of a class must supply operations that correctly manage the persistent data for their class. In addition, the implementor must overload the constructors if instances of the class are allowed to be either persistent or nonpersistent. It is perhaps worth noting however, that the implementor of a class can also ensure that all instances of the the class persist by not providing a constructor that allows nonpersistent instances to be declared.

If the implementation language provides class structure information (e.g. the *clsTypes* field produced for an Objective-C [Cox 86] class), then a set of mechanisms could be implemented by a class that would manage all classes using this information. Inheritance could therefore be used to add persistence without the implementor of a class having to add the operations that save and restore the persistent data. They would however, still have to invoke an operation to add a persistent object to the persistent root. The disadvantage of this approach would be that the implementor would have no control over which parts of an object persist, as typically not all of an object's state need persist. The fact that the implementor of a class has to provide operations to save and restore the persistent data is therefore not such a burden as might first appear.

In summary, the technique described in this chapter is sufficient to support the permanence of effect property of an atomic action, but not to support orthogonal persistence. Classes constructed using these persistence mechanisms are more flexible than those constructed when the permanence mechanisms conventionally used to support reliability are adopted. The end result therefore, is beneficial, even though full support for orthogonal persistence is not possible.

5.8 Concluding remarks

This chapter described how the permanence of effect property of an atomic action may be supported by ensuring that permanent system state is modelled using objects which are persistent. The lifetime of a persistent object extends beyond the lifetime of the application in which the object was created, thereby ensuring the permanence of the state the object represents.

The chapter began by considering the requirements of persistence. These are a means of overcoming the scope associated with an object's declaration, and a mechanism for moving the state of an object to and from non-volatile storage. Since much of this functionality is already provided by the recovery mechanisms which were described in the last chapter, the approach described in this chapter was to extend these mechanism so that an object may be persistent as well as be recoverable. To organise the state of a persistent object in non-volatile storage, the design and implementation of an object store was described.

The mechanisms described in this chapter, allow the object state that persists to be under the control of the implementor of a class, and be determined either statically or dynamically. In addition, by effectively overriding the persistence of any internal objects, performance can be increased as the storage overhead is decreased. The next chapter describes the design and implementation of a set of classes that illustrate the possible optimisations.

Chapter 6

Performance and Optimisations

The last three chapters have described how atomic actions, and objects which are recoverable and persistent, may be constructed using the features of an object-oriented programming language. This chapter develops an example which illustrates how the recovery and persistence mechanisms described in earlier chapters can be used to construct classes that may be used within atomic actions. One of the advantages of the way that classes may be constructed, using these mechanisms, is that the recovery and persistence of internal objects may be optimised. To illustrate these optimisations, this chapter describes the results of a number of tests which were made to determine the performance of applications that employ atomic actions and instances of the example classes.

The chapter begins by describing the design of the example which is a banking system. This design is structured into a number of classes, each class being derived from the class `Object` (described in chapters four and five). By using the state based recovery and persistence provided by `Object`, instances of the classes which make up the banking system may be controlled using atomic actions implemented by the class `AtomicAction` (described in chapter three). The remainder of this chapter describes a number of simple applications which were constructed to measure the performance of the atomic action implementation, and the recovery/persistence mechanisms. The performance figures given are not intended to provide an authoritative measure of the implementation, which at the time of writing is an experimental prototype, developed to test the soundness of the ideas presented in this thesis and provide a testbed for experimentation. The performance of the prototype can be optimised in numerous ways, and a number

of these optimisations are described in the final chapter which discusses future areas of work.

6.1 A banking system

Assuming that a bank consists of a number of customers, with each customer holding a current account and perhaps a deposit account, then a number of classes can be declared to represent the bank, the customers, and the accounts each customer holds. This section begins by defining the classes required, not taking into consideration any recovery or persistence aspects. Once these classes are defined, the changes needed to add recoverability and persistence to the classes will be briefly described.

The basic functionality required by an account may be represented by the class `Account` (illustrated in Figure 6.1). To provide the current and deposit

```
class Account
{
    Money amount;
public:
    Account();
    ~Account();

    Money deposit(Money);
    Money withdraw(Money);
    Money balance();
};
```

Figure 6.1: The class `Account`

accounts, the classes `CurrentAccount` and `DepositAccount` are derived from the class `Account`. The class `CurrentAccount` adds extra functionality such as an overdraft limit, whereas the class `DepositAccount` adds extra functionality that enables the interest due on the amount maintained by the `Account` to be calculated on a weekly basis. To simplify the following discussion however, it will be assumed that the current and deposit accounts are instances of the class `Account`.

To model a customer, the class Customer (Figure 6.2) simply maintains

```
enum account_type {CURRENT, DEPOSIT};

class Customer
{
    Account *Current;
    Account *Deposit;

public:
    Customer();
    ~Customer();

    void OpenDepositAccount();

    Money deposit (account_type, Money);
    Money withdraw(account_type, Money);
    Money balance (account_type);
    Money transfer(account_type, Money);
};
```

Figure 6.2: The class Customer

references to the current and deposit accounts held by the customer. The deposit account reference may be null as the deposit account must be explicitly created using the OpenDepositAccount operation (the CurrentAccount instance being automatically created when the Customer object is created). The three main operations (deposit, withdraw, and balance) provided by the Customer class correspond to those provided by the Account class, but take an extra argument (of type account_type) that defines which Account (either CURRENT or DEPOSIT) is required by the invoked operation. The extra operation (transfer) enables an amount to be transferred from one account to the other, the recipient account being passed as the account_type argument.

To model a bank given the Customer and Account classes, a class such as the class Bank is defined. This class (illustrated in Figure 6.3) maintains the Customer instances, and provides operations that enable the accounts maintained by a Customer to be opened or closed. When a new Customer is added to the Bank using the OpenAccount operation, a CustomerId is returned which must be used to access the accounts held by the customer, so that all access to both the Customer and Account instances must pass through a Bank instance.

```
typedef Integer CustomerId;

class Bank
{
    Customer **customers;
    Integer no_customers;

public:
    Bank();
    ~Bank();

    CustomerId OpenAccount();
    void       CloseAccount(CustomerId);
    void       OpenDepositAccount(CustomerId);

    Money deposit (CustomerId, account_type, Money);
    Money withdraw(CustomerId, account_type, Money);
    Money balance (CustomerId, account_type);
    Money transfer(CustomerId, account_type, Money);

    Money Assets();
};
```

Figure 6.3: The class Bank

Declaring an instance of the class Bank in an application, and invoking operations to create customers and their accounts will produce the relationship between objects shown in Figure 6.4 (which illustrates a Bank with two

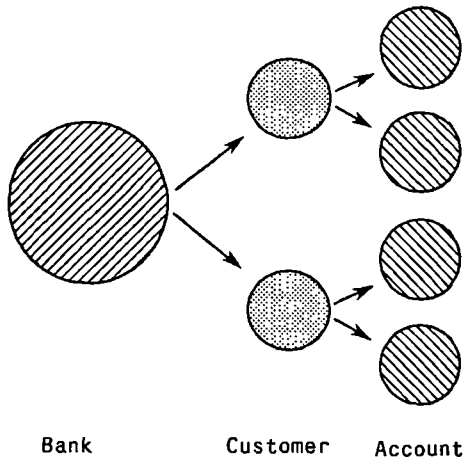

Figure 6.4: The relationship between objects

customers, each having both a current and deposit account).

This very simple banking system provides no support for tolerating system crashes and maintaining the integrity of the objects that represent the banking system. In addition, the persistence of the objects created is not supported, so that

the scope of the objects is that of the application program in which they are created.

To enable atomic actions to be used in an application the classes described above must be recoverable, persistent and provide concurrency control. To provide the first two of these three properties, the mechanisms described in the previous two chapters can be used (concurrency control aspects being ignored as they are not part of the work described in this thesis). This involves adding recovery and persistence mechanisms to the three classes described above, and declaring instances of the class `AtomicAction` to control the outcome of operations, and provide tolerance against system crashes. These additions are easily achieved and are described in the remainder of this section.

To add recoverability and persistence to a class, that class can be derived from the class `Object` (described in chapter four). When deriving a new class from the class `Object`, three (or more) additional operations must be implemented to correctly support the functionality `Object` provides. Two of the three operations are concerned with managing the state of the object so that a bit-image may be taken (in the case of `save_state`) or restored (in the case of `restore_state`).

To support recovery, recovery data must be created before an object is modified. The class `Object` provides an operation (`modified`) that should be invoked in the implementation of each operation that updates the state of the object. When the `modified` operation is invoked, management information is created to hold the recovery data which is created as the `save_state` operation is invoked. Each class must therefore refine the `save_state` (and corresponding `restore_state`) operation to return (or expect) suitable recovery data which enables an instance to be recovered should the containing atomic action be aborted.

To ensure the persistence of instances of the Bank, Customer, and Account classes, each class must also refine the save_state and restore_state operations so that they return or expect persistent data. Since the Bank and Customer classes are structured, the recovery and persistent data (and mechanisms) will differ, and both classes must refine the destroy operation to ensure that the persistent data for the internal objects is removed from the object store when the destroy operation is invoked on the containing object.

To enable instances of the three classes to persist, each class must implement a constructor that invokes the persist operation, passing the unique identifier needed to name the persistent data as an argument when the object is an existing persistent object. The simplest form of constructor is one that takes a unique identifier argument, and directly invokes the persist operation using this argument. A constructor of this type can be added to both the Customer and Account classes, since they are only accessed by a Bank object. The class declaration for the Customer class is shown in Figure 6.5 to illustrate the

```
enum account_type {CURRENT, DEPOSIT};

class Customer : public Object                          // new super-class
{
    Account *Current;
    Account *Deposit;

public:
    Customer();
    Customer(Uid*);                                     // new constructor
    ~Customer();

    void OpenDepositAccount();

    Money deposit (account_type, Money);
    Money withdraw(account_type, Money);
    Money balance (account_type);
    Money transfer(account_type, Money);

    virtual ObjectState* save_state(ObjectState*);      // new operation
    virtual void         restore_state(ObjectState*);   // new operation

    virtual void destroy();                             // new operation
};
```

Figure 6.5: The new declaration of class Customer

changes required.

To provide a convenient way of naming a Bank instance, a constructor may be added that takes a String argument and employs a name server to map the String to the unique identifier (in the manner described in the example given in section 5.3.1 of the last chapter). A new declaration of the class Bank that has the four extra operations is illustrated in Figure 6.6.

```
typedef Integer CustomerId;

class Bank : public Object                              // new super-class
{
    Customer **customers;
    Integer no_customers;

public:
    Bank();
    Bank(String);                                       // new constructor
    ~Bank();

    CustomerId OpenAccount();
    void       CloseAccount(CustomerId);
    void       OpenDepositAccount(CustomerId);

    Money deposit (CustomerId, account_type, Money);
    Money withdraw(CustomerId, account_type, Money);
    Money balance (CustomerId, account_type);
    Money transfer(CustomerId, account_type, Money);

    Money Assets();

    virtual ObjectState* save_state(ObjectState*);      // new operation
    virtual void         restore_state(ObjectState*); // new operation

    virtual void destroy();                             // new operation
};
```

Figure 6.6: The new declaration of class Bank

Given these changes to the class, declarations of the form:

```
Bank MyBank("NatWest");
```

are all that is required to create a new, or name an existing, instance of Bank (in this case one called NatWest).

In addition to declaring an atomic action in an application to control the outcome of operations invoked on instances of these classes, atomic actions may also be declared within the implementation of an operation. For instance, the class Customer provides an operation called transfer that transfers an amount from one type of account to another. To perform this task, the

implementation must invoke the `withdraw` operation provided by the relevant account. In the implementation provided by the class `Account`, this operation returns a value that is the maximum amount that can be withdrawn up to the value requested. If this amount held in the source account is insufficient for the `transfer` operation, then the amount withdrawn must be replaced in the source account and the `transfer` operation abandoned. To simplify the implementation of this operation an instance of the class `AtomicAction` can be declared in the implementation of the `transfer` operation, so that recovery of the `Account` object may be performed automatically by simply aborting the atomic action. An implementation of transfer that operates in this manner is illustrated in Figure 6.7.

```
Real Customer::transfer(account_type to, Real amount)
{
    if (Deposit && (amount > 0)) // deposit account and valid amount ?
    {
        AtomicAction transfer_Action;
        transfer_action.Begin();

        Account *in  = (to == DEPOSIT) ? Deposit : Current;
        Account *out = (to == DEPOSIT) ? Current : Deposit;

        if (out->withdraw(amount) < amount) // amount insufficient ?
        {
            transfer_action.Abort();
            amount = 0;
        }
        else
        {
            in->deposit(amount);
            transfer_action.Commit();
        }
    }
    else
        amount = -1;

    return (amount);
}
```

Figure 6.7: The `Customer transfer` operation

In addition to declaring and using instances of `AtomicAction` in the implementation of an operation, instances can also be declared to be part of the state of a class. One example of a situation where this might be useful is the class `Bank`, where the `Begin` operation can be invoked in the constructor for the class,

and a corresponding `Commit` in the destructor, resulting in all operations on the `Bank` occurring within an atomic action environment.

The description of the classes that make up the banking system, and changes necessary to ensure that the classes are recoverable and persistent, illustrate how easy it is to add this extra functionality to a class. Because the class `Object` and the various record classes provide most of the necessary recovery and persistence mechanisms, the implementor of a class only has to add three (or four in the case of `Bank` and `Customer`) operations in the manner described for instances of the class to be recoverable and persistent. The burden of this task is greatly eased by the support these classes provide, so that this approach to providing recoverability and persistence is a truly practical approach.

The next section describes the performance of a number of simple applications that employ instances of the `Bank`, `Customer`, and `Account` classes. The purpose of the tests described in the next section was to measure the overhead imposed when adding recovery and persistence to a class using inheritance, and how this performance may be optimised by altering the granularity of recovery and persistence.

6.2 Measuring the performance

This section describes a number of simple tests which illustrate the performance of the experimental, unoptimised, implementation of the system described in this thesis. Since distributed atomic actions remain to be implemented, all tests were made using objects and atomic actions that were local to the process and node of the test program. The tests were performed on a Sun 3/160 workstation that has four megabytes of memory and runs the Sun implementation (version 3.5) of the Berkeley BSD4.2 UNIX operating system. The execution time of an operation, system call, or application, was determined

using the UNIX *getrusage* system call, taking the average user time for a large number of executions.

To illustrate the performance of this environment, Figure 6.8 lists the

1.	UNIX *gethostid* system call	25 µs
2.	UNIX file *open* system call	50 µs
3.	UNIX file *write* + *fsync* calls	60 µs
4.	Creation of `Uid` object	150 µs

Figure 6.8: The performance of the environment

execution times for a number of basic operations. The first time given in Figure 6.8 is for a typical minimal system call (*gethostid*). The second is another system call (*open*) which involves access to secondary storage, measuring the time taken to open an existing file in the same directory as the test program. To illustrate the input/output performance of the secondary storage provided by the workstation, the third figure gives the average time taken to write a 512 byte buffer to a file and flush the buffered write to the disk (i.e. a *write* system call followed by an *fsync* system call). The final figure is the time taken to create a typical small object, in this case an instance of the class used to represent unique identifiers (the class `Uid`). The creation of this class involves two system calls (*gethostid* and *gettimeofday*) and the use of the free store operator (called *new*) provided by the language C++, that in turn involves the execution of the C/UNIX library function *malloc*.

When the state of a recoverable/persistent object is updated, management information must be added to an atomic action so that the object may be recovered if the atomic action aborts, or made persistent when the top-level atomic action commits. The first performance tests described in this section were made to determine the overhead of invoking an operation that records management

information with an atomic action. Both this test, and the rest described in this section, are limited to the mechanisms provided by the class `Object`, which provides an operation called `modified` for recording management information.

To measure the overhead of recording (or attempting to record) management information using the `modified` operation, tests were made on instances of the class `Account` (described in the last section) which is a sub-type of `Object`. To provide a basic comparison, the time taken to execute the `deposit` operation on an instance of the unrecoverable (i.e. not derived from `Object`) implementation of this class was measured. Several measurements were also made on the recoverable implementation: when in a non atomic action environment; when in a new atomic action environment; and when in an existing atomic action environment. The results of these tests are listed in Figure 6.9. The overhead of

1. Unrecoverable `deposit`	100 µs
2. Recoverable `deposit`, non-action environment	110 µs
3. Recoverable `deposit`, new action environment	1.4 ms
4. Recoverable `deposit`, old action environment	150 µs

Figure 6.9: The `modified` operation overhead

executing the `modified` operation when no atomic actions are active is approximately 10%. The only time the `modified` operation creates recovery data and records management information is the first time the object is modified. The third timing is a measure of this situation. As the time for the third test illustrates, the overhead imposed by creating recovery data and recording management information with an atomic action is approximately ten times slower than when there is no atomic action executing. This overhead is partially dependent upon the amount of the recovery data required by the recoverable object, but in the above example the majority of the overhead is in the creation of the management information and the addition of this information to the active

atomic action. When the `modified` operation is invoked in an action environment, and the object has already recorded management information, the overhead is approximately 50% of the unrecoverable object execution time as the fourth result listed in Figure 6.9 shows.

To determine the execution overhead of an atomic action, tests were made to measure the time taken to perform a null atomic action that involves invoking the `Begin` operation followed immediately by either a `Commit` or `Abort` operation. To test for any difference between top-level and nested atomic actions, four sets of tests were made, but the result of the all tests gave the execution time of a null atomic action to be approximately 70 μs.

To examine the performance of atomic actions and the recoverable/persistent classes described at the start of this chapter, a number of tests were made which involved timing the execution of an atomic action within which an instance of `Bank` is created, one or more `Customers` added to the `Bank` (using the `OpenAccount` operation) and an amount deposited in the current `Account` of each `Customer` (using the `deposit` operation). The number of `Customers` added to a `Bank` varied from one to one hundred per atomic action. The time taken for both nested and top-level atomic actions to commit and abort these modifications are listed in Table 1.

Atomic Action Type	1 `Customer` (seconds)	10 `Customers` (seconds)	100 `Customers` (seconds)
nested commit	0.005	0.06	1.9
nested abort	0.004	0.07	2.3
top-level commit	0.04	0.2	4.3
top-level abort	0.004	0.07	2.3

Table 1: Atomic action execution times

As the results in Table 1 show the times for the abort of nested and top-level atomic actions are the same. The commit of a nested atomic action is fractionally slower for a single `Customer` object, as this involves merging management information held by the nested atomic action with that held by the parent atomic action. As the number of `Customers` increase however, the cost of recovery becomes greater than the cost of merging information so that a nested commit becomes faster than either a nested or top-level abort.

The results for the commitment of a top-level atomic action illustrate the cost of the two-phase commit protocol and the storage of the persistent data for the modified objects in the object store. As the number of objects increase however, the overhead of storage reduces as the amount of management information that must be merged (in the case of a nested commit) becomes correspondingly greater. From being almost eight times slower with one `Customer`, the top-level commit becomes four times with ten, to twice as slow as a nested commit with one hundred.

To increase the performance of an application, changes could be made to the implementation of the `AtomicAction` class to improve the method of managing the records. A high proportion of the atomic action overhead is due to the way that the records are managed, the overhead becoming greater as the number of records held by the atomic action increase. Even so, applications that use atomic actions, and classes constructed using the experimental system, produce acceptable results. In addition, performance optimisations are possible if the implementor of a class manages the persistence and recoverability of any internal objects.

Chapter four described how a class that instantiates instances of other recoverable classes (which are termed *internal* objects) may override the recovery management provided by the class of the internal recoverable objects to directly

recover the internal objects. In this manner, the granularity of recovery may be altered to suit the situation in which the internal objects are modified. The set of classes described in the previous section are an ideal candidate for such a technique, as instead of each class being responsible for managing its own recovery the containing classes (the classes `Customer` and `Bank`) may override the recovery of the internal objects (instances of the class `Account` and `Customer` respectively).

In addition to directly managing the recovery of internal objects, the persistence of the internal object may also be controlled in the manner described in chapter five. To illustrate three possible levels of recovery and persistence, consider a newly created instance of `Bank` that consists of three `Customers`, each `Customer` having a current `Account`. Assuming a similar sequence of operations to those which produced the results given in Table 1, then the recovery/persistence data that will be created when the classes that make up the banking system are structured in the manner illustrated in Figure 6.4 are shown in Figure 6.10.

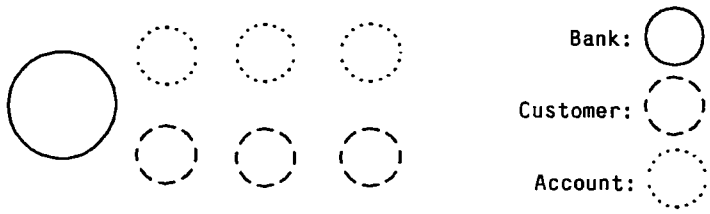

Figure 6.10: Recovery/persistent data for 3 `Accounts`

If the same number of `Customers` and `Accounts` are created by an application, but the `Customer` class assumes the management of recovery and persistence for the `Account` objects, then the recovery/persistent data will be structured in the manner shown in Figure 6.11.

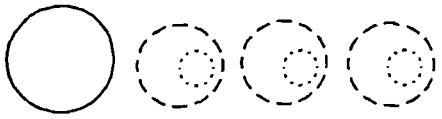

Figure 6.11: Customer optimisations

The final organisation considered is one where the Bank class manages the recovery and persistence of the Customer objects, with each Customer object managing the Account objects, the resulting structure being shown in Figure 6.12.

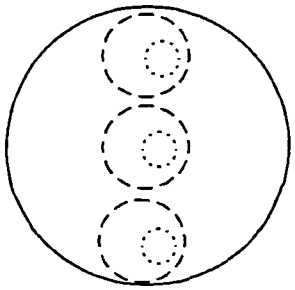

Figure 6.12: Bank and Customer optimisations

To determine the effect that these changes in granularity have on execution time, the application that produced the results in Table 1 was used with alternative implementations of the Customer and Bank classes.

The results for the first situation, where a Customer object assumes the recovery of the Account objects, are listed in Table 2. The results in this table may be contrasted with those given in Table 1, illustrating how the performance may be significantly improved by such optimisations. In this situation, two sets of recovery data and management information (instead of three) will be created and recorded with an atomic action (the situation illustrated in Figure 6.11). The first set is for the Customer objects (which includes the recovery data for the Account objects), and the second for the Bank object, whereas previously there had been a set of recovery data for each class. The effect of this optimisation

Atomic Action Type	1 Customer (seconds)	10 Customers (seconds)	100 Customers (seconds)
nested commit	0.003	0.03	0.6
nested abort	0.003	0.03	0.7
top-level commit	0.02	0.1	1.5
top-level abort	0.003	0.04	0.7

Table 2: Customer class optimisations

becomes larger the greater the number of objects created, from taking around a two thirds the original time to execute with a single object (since there are only two objects being managed instead of three), to half as long with ten objects (a reduction from 21 to 11 objects), to a third of the time with one hundred objects (201 to 101 objects). The difference in times between the top-level and nested atomic actions also reflect those observed in the previous table.

Table 3 gives the results for the situation where the Bank class assumes the

Atomic Action Type	1 Customer (seconds)	10 Customers (seconds)	100 Customers (seconds)
nested commit	0.002	0.01	0.06
nested abort	0.003	0.01	0.08
top-level commit	0.01	0.03	0.2
top-level abort	0.003	0.01	0.08

Table 3: Bank + Customer class optimisations

recovery of the Customer class (which is also assuming the recovery of the Account objects). This situation produces only one set of recovery data and management information per Bank object. When only one Account object is modified, the execution time is almost the same as that for the previous example (Table 2). As the number of objects increase however, the performance increases so that for ten objects the execution time is a third of that for the previous case, to

a tenth when one hundred objects are modified. Comparing the results in Table 3 with those in Table 1, the optimisations available are clear, as the performance may be increase by thirty times for one hundred objects when aborting the atomic action, or committing a nested atomic action. When the commitment of a top-level action is considered, the increase in speed of around twenty times illustrates how the speed of the non-volatile storage becomes the limiting factor.

6.3 Concluding remarks

To illustrate how the recovery/persistence mechanisms may be added to a class, the first section of this chapter described the design of an unrecoverable and nonpersistent set of classes that provide the abstraction of a banking system. The remainder of the section then described how the recovery/persistence mechanisms may be added by deriving the existing classes from the class `Object`, and implementing three (or four) extra operations. Since the classes developed to support recovery and persistence provide much of the necessary support, the addition of recovery/persistence mechanisms to an existing class was easily achieved.

To show that the example classes (developed in the first section) operate in the manner described in earlier chapters, the second section of this chapter described the performance of a number of tests that employ instances of these classes. As the results show, the performance of the system is adequate, despite the fact that the current implementation has been constructed to test the soundness of the ideas presented in this thesis and provide a flexible testbed rather than address performance issues. One of the major advantages of the mechanisms described in this thesis is the way that the recovery and persistence of objects can be altered to improve the performance.

When an instance of a recoverable/persistent class consists of instances of other recoverable/persistent classes then the containing class may be implemented in such as way that the speed of recovery and storage is tailored to the way that the containing class is used. For example, with the banking system described in this chapter, if a bank has a large number of customers who infrequently use the bank then the best form of recovery and storage would be on a per customer basis. In the opposite case, a small number of customers who frequently use the bank, the optimum granularity of recovery and storage would be the entire bank.

The results given in the previous section showed how both the granularity of recovery and persistence may be altered at the same time. In a banking system, recovery should be on a per `Account` basis. The results described in this chapter show that optimisations are possible but do not conform to the ideal abstraction of a banking system (recovery being on a `Customer` or `Bank` basis). Since instances of a persistent class can be declared to be nonpersistent and still return persistent data, a more suitable set of classes could be defined. The `Bank` class may be constructed so that it is responsible for the persistence of nonpersistent `Customers`, and the persistent data returned by the `Customer save_state` operation may include the persistent data for the `Account` objects (the result being the situation illustrated in Figure 6.12). Given this approach, the `Customer` and `Account` objects will be grouped into the persistent data for the `Bank` object. This example illustrates the flexibility of the recovery and persistence mechanisms, and the advantages of being able to declare nonpersistent instances of a persistent class. Approaches of this type are not possible when the compiler for the implementation language, or underlying operating system, automatically provides the necessary support for recovery and persistence.

Clearly, the execution overhead imposed by the addition of these mechanisms and the use of atomic actions in an application is considerably greater than when no reliability or persistence mechanisms are employed. The advantages of being able to declare a persistent object, and not have to explicitly manage its storage, in addition to the added recovery benefits, are difficult to quantify. The tests described in this section have all been concerned with measuring the execution overhead and performance of the recovery/persistence mechanisms and the atomic action implementation. The addition of these mechanisms to a class, and the use of atomic actions, also increases the non-volatile storage (disk space) required by an application, and any persistent objects created, in addition to the amount of volatile storage (memory) needed to execute the application. Given that the cost of both types of storage are rapidly decreasing, and the processing power of computers rapidly increasing, then the benefits of adding recovery and persistence to support reliability, greatly outweigh the increasing storage requirements and execution overhead that results.

The performance figures given in the last section are for local objects and non-distributed atomic actions. When objects are remote, operation invocations must involve remote procedure calls which will add to the execution overhead. With the distributed atomic action design described in chapter three, the overhead that results from operating in an atomic action environment will be slightly higher (ignoring the RPC overhead) than that for a non-distributed atomic action, as each invocation involves calls on the components that manage the action environment at both the local and remote node. When the a cost of a remote procedure call is considered (approximately 10ms for a null RPC in the environment used for the tests) then, until the atomic action is terminated, the greatest overhead will be the RPC itself.

To further improve the performance of an application that uses instances of classes constructed using the mechanisms described in this thesis, the implementation of the mechanisms and supporting components (such as the object store) may be optimised. A number of optimisations are described in the next chapter which summarises the work described in this thesis, and discusses future areas of work.

Chapter 7

Conclusions

The construction of reliable distributed applications is a difficult task that can be aided by the use of programming constructs such as atomic actions. The concepts and practise behind the use of atomic actions for maintaining the integrity of databases are well understood and are beginning to be employed in distributed environments. Their use in distributed object-oriented systems is relatively new however, with current approaches concentrating on providing support through special linguistic constructs or operating system calls. The research described in this thesis differs from previous research in that support for atomic actions is provided using only the features of an object-oriented language. Using this approach, atomic actions may be implemented as objects, and the support required to ensure the failure atomicity and permanence of effect properties added to a class of objects by exploiting inheritance. The result is a programming system for constructing reliable distributed applications that is the closest so far to being classified as object-oriented, since the principle components provided by the programming system are modelled as objects. This chapter summarises the work presented in this thesis and speculates on future areas of work.

7.1 Thesis summary

As distribution becomes commonplace the need for a programming system that may be used to construct distributed applications, becomes greater. Distribution introduces a new domain of problems however, as the independent failure of components of a distributed computation can lead to the abnormal behaviour of the computation. Unless such failures are addressed by a

programming system, then the applications produced using the programming system will not be reliable.

One approach towards supporting the reliability of an application is to use the computing abstraction known as the *atomic action*. The properties of an atomic action ensure that the system state, modified during the execution of the application, will be failure atomic, unaffected by concurrent computations, and permanent when successfully terminated. To provide these properties, the system state accessed within the scope of an atomic action must be managed by the containing atomic action. To model the system state, the object paradigm may be used.

The object paradigm is currently one of the best programming methodologies for managing state and modelling behaviour. The properties provided by a language that supports the object paradigm are: *data abstraction*, *encapsulation*, and *inheritance*. Data abstraction and encapsulation enable behaviour and state to be implementing using a *class*. Instances of a class share common behaviour and are known as *objects*. The state that each object represents is defined by the class of the object, and consequently may be managed by an atomic action if suitable mechanisms are added to each class of objects.

To take advantage of the support provided by the object paradigm, the programming system known as *Arjuna* (of which the work in this thesis forms part) employs an object-oriented language. Arjuna supports nested atomic actions which may be used to control the objects declared in an application program. To support distribution, Arjuna employs *remote procedure calls* and the *client/server* execution model, and these mechanisms are hidden through the use of *stubs*. The stub objects required by the client program, and the server which manages a remote object, are produced using a stub generator from a class definition.

The research presented in this thesis has addressed two distinct issues surrounding the production of objects and atomic actions. The first has concerned the construction of atomic actions, the second the mechanisms required to ensure that a class of objects may be controlled by an atomic action so that the properties an atomic action provides may be met. To support atomic actions, objects must be recoverable (to ensure the failure atomicity property), provide concurrency control (to support the serialisability property), and persist (to meet the permanence of effect property).

A class which provides recoverability, persistence and concurrency control is known as an *Arjuna class*. When an instance of an Arjuna class is modified, recovery data is recorded so that the old object state may be restored, thereby supporting the failure atomicity property. Arjuna objects may also be persistent and are stored in a *passive* state in an *object store* until required by an application. When an application that has invoked operations on a persistent Arjuna object terminates successfully, the persistent state of the object is saved in the object store if the object has been modified, ensuring the permanence of effect property. As operations are invoked on an Arjuna object, concurrency control in the form of locking occurs ensuring that the serialisability property is met.

Of the above three properties, this thesis only addresses those associated with managing the state of an object: the recoverability and persistence properties. The rest of this section summarises how the work described in this thesis has approached the provision of these properties and the abstraction of atomic actions.

In chapter two the system model, along with the fault-tolerance terminology used throughout the thesis, was defined. In addition, the features and properties of the object paradigm were described using the language C++, and the programming system Arjuna detailed. Since the Arjuna project is addressing

similar issues to a number of other projects, chapter two also included a review of the more important projects. The major difference between Arjuna and other research projects is the way that the support for programming reliable distributed applications using objects and atomic actions is provided. Most research projects have produced new programming languages or operating systems to provide the necessary support. Arjuna is unique however, in that the support required has been provided using the features of an object-oriented programming language. The result is a number of classes that provide the abstraction of atomic actions and the mechanisms required to ensure that a class of objects is recoverable, persistent, and supports concurrency control.

Chapter three described how the abstraction of atomic actions may be provided using the features of an object-oriented language. The chapter began by describing a model of atomic actions and the way that the execution of an atomic action may be considered in terms of a sequence of event and state changes. This model led to a description of the functionality required by an atomic action, detailing what information is required to meet the properties associated with an atomic action, and the way that this information must be managed when atomic actions are nested within one another. To manage the information an abstraction termed a *record* was defined that provides operations which may be invoked at each action event. Three types of record have been produced: the state management record which manages the recovery and persistence of an object, the access management record which manages the concurrency control applied on an object, and the distribution management record which coordinates the state and access management records maintained on remote nodes to manage remote objects.

To provide the abstraction of atomic actions, the design and implementation of the class `AtomicAction` was described. This approach of implementing atomic actions as objects is unique, as other research projects have concentrated on

providing atomic actions through special linguistic constructs or operating system calls. The `AtomicAction` class provides operations that enable the outcome of a computation that accesses Arjuna objects to be controlled. Since much of the functionality is provided by the record abstraction, the `AtomicAction` class simply invokes record operations in response to the various action events that occur during the execution of a computation. During the description of the implementation, an example was given to illustrate how the various records are managed, and how each record in turn manages a specific property to ensure that the objects accessed by a computation provide the properties associated with an atomic action.

In the atomic action model described in chapter three, it was assumed that only the volatile state of an object is accessed during a computation, with the non-volatile state only being updated should the top-level action in an application successfully commit. As a result, a node crash during an application will result in the loss of all volatile state and the effective abort of the atomic action. If the atomic action is performing a top-level commit however, a node crash during the commit operation must not result in an inconsistent system state being established. To ensure the atomicity of the commit operation in such circumstances, chapter three described how a commit protocol may be employed. An implementation of the two-phase commit protocol was described that took advantage of the fact that the management information, the record objects and the atomic action, are persistent objects, enabling their state to be saved in the object store, thereby providing tolerance against node crashes.

To extend the design and implementation of an atomic action to a distributed environment, a later section of chapter three described the distribution management record, and changes to the stub objects needed to support distributed atomic actions. The chapter ended by discussing the disadvantage of modelling

atomic actions as objects, which is the inability to enforce the scope or boundaries of the resulting atomic action. A simple solution to this problem was described.

Chapters four and five described how the mechanisms that support recoverability and persistence, respectively, may be provided. Chapter four began by considering how recoverability may be added to a class of objects by adding suitable mechanisms. Two forms of recovery were considered: *state* and *operation* based. A state based approach relies on saving a snapshot of an object so that the object may be recovered to the state held in the snapshot. An operation based approach involves recording the operations invoked on an object, so that the old state of an object may be recovered by undoing the operations.

Chapter four discussed the alternative methods of adding recovery mechanisms to an existing class, given that modifications to the compiler for the implementation language, or underlying operating system, were not available. Using the features of an object-oriented programming language, a number of approaches were considered. The first was termed the *container* approach which consisted of a new class that contains an instance of the unrecoverable class, providing an identical interface but ensuring that sufficient recovery data is recorded should the internal object be updated. The second approach was the *unrecoverable inheritance* approach, which relies on exploiting inheritance to inherit the functionality of the unrecoverable class, with refined versions of the operations that update the unrecoverable object being provided to record suitable recovery data. The third approach, termed the *recoverable inheritance* approach, was a mixture of the previous two, with a class containing an instance of an unrecoverable class, but inheriting recovery mechanisms from a *base recoverable class.* The advantages and disadvantages of these three approaches were considered, leading to the final approach, which was termed the *multiple inheritance* approach. Using this approach, a new class is constructed by inheriting from a base recoverable class, and the unrecoverable class. In this

way, the new recoverable class is a sub-type of both classes, enabling instances to be treated as if they are instances of the unrecoverable class, but requiring only a single management class (the record class) as all instances are also sub-types of the base recoverable class.

Given such an approach, chapter four described how the base recoverable class may be implemented. Two implementations were described, one providing *state based* recovery, the other *operation based* recovery. The base recoverable class that implements state based recovery is called `Object`, and provides mechanisms that enable a snapshot (or bit-image) of the state of an object to be taken in a class-independent manner. The base recoverable class that supports operation based recovery, called `Operation`, relies on class-dependent recovery information and greater support from the implementor of a new recoverable class. During the description of each approach, a number of examples were given illustrating the flexibility of each approach. In addition, during the description of the operation based approach a method of performing compensation operations was detailed. Compensation enables unrecoverable objects, such as a hard-copy printer, to provide the abstraction of recovery.

As the discussion in the earlier sections of chapter four had been concerned with adding recoverability to an existing class, a later section described how recoverable classes that provide new abstractions may be constructed. The discussion concerned the various methods that may be used, such as constructing a new class using existing recoverable objects or, alternatively, unrecoverable objects with the class inheriting recovery in the manner described in earlier sections of the chapter. The differences is granularity were described, allowing a degree of flexibility in the amount of state recovered when an object is restored. The final section of the chapter assessed the approach of constructing recoverable objects based upon inheriting from a class that provides recovery mechanisms. The conclusions that may be made from this chapter are that exploiting

inheritance to add a property such as recoverability to a class of objects, either an existing class or a class that provides a new abstraction, is both simple and flexible.

Chapter five began by describing how the permanence of effect property of an atomic action may be met by modelling the permanent system state as persistent objects. The concept of persistence was described, and the way that the scope rules of a programming language must be overridden to ensure that an object is not deallocated before being saved in non-volatile storage to effect persistence. Because the state of an object is saved in non-volatile storage, persistent programming language provide atomic actions to ensure the failure atomicity of the storage of new object state. The chapter described how the persistence of an object therefore becomes a function of the outcome of an atomic action.

Since the persistence of an object is in effect large grained recovery, the mechanisms that must be added to a nonpersistent class were considered to be extensions to those developed for state-based recovery. As a result, the operations provided by the class `Object` to support recovery were extended and modified so that they could also support persistence, and two new operations (`activate` and `deactivate`) provided. The chapter described how the concurrency control mechanisms were responsible for activating an object, and how a new record class was derived from the recovery record class to ensure that a persistent object is deactivated when the top-level atomic action commits.

To organise the persistent data in non-volatile storage, the design and implementation of an object store was described. To collect instances of the same class together, the object store is structured into two-levels. The first level is designed to be the *root* object store, with each class having its own object store in this root object store. The second level is the object store for each class which

contains all instances of this class. An implementation of this organisation was described using the UNIX file system.

One advantage of the object store organisation is that since the location of an object is based upon the class of the object, it is not possible to subvert the type system by creating an object using the persistent data of another class. A further use of this organisation is made by the program that recovers the object store after node crash. This crash recovery program scans the object stores of particular classes of object to discover the outcome of any top-level atomic actions which were in the process of committing. Since all instances of a class are collected together, the crash recovery program simply employs the natural organisation of the object store, simplifying the commit protocol and crash recovery mechanism.

The penultimate chapter of this thesis described the construction of a simple example, and used this example to make tests on the performance of the implementation of atomic actions and the class `Object`. The ease with which recovery and persistence may be added to an existing class was illustrated as the example was first developed without thought to such mechanisms. Once the classes were designed to support the example, the way in which the persistence/recovery mechanisms may be added by inheriting the support that the class `Object` provides, was described. By structuring the example into a number of classes, the way that the recovery granularity may be altered to increase the performance of an application containing atomic actions was described and the optimisations supported by the test figures. Given that the current implementations of atomic actions, the object store, and the recovery/persistence mechanisms are all unoptimised prototypes, the performance figures achieved illustrated the practicality of this approach to constructing reliable applications that use atomic actions and objects.

7.2 Future work

During the description of the mechanisms required to provide atomic actions and objects that are recoverable and persistent, a number of areas of future work have become apparent, and are described in this section.

The first area of future work concerns the distribution of atomic actions, the design of which was described in chapter three. A full implementation of this design remains to be realised, although the a number of tests have been made to verify the soundness of the design. To fully support distribution, the simple naming scheme described in this thesis must be expanded so that information about the server for a particular class of objects is stored along with the mapping from user name to unique identifier for the persistent objects. Given such a design, a distributed name server is needed which is both replicated and fault-tolerant. As a first step towards this aim a simple name server is being implemented, and once fully distributed atomic actions are available, a more complex name server may be implemented using the support provided by the Arjuna programming system.

Another extension to the atomic action design which will be possible once objects and atomic actions can be distributed is concurrency within an atomic action. Concurrency within an atomic action enables nested atomic actions that have a common parent to execute concurrently. One advantage of concurrent atomic actions is that a number of nested atomic actions may be invoked to perform an operation, with the first nested atomic action to commit resulting in the abortion of the outstanding atomic actions. Such an approach is useful in a distributed environment where resources are replicated. A concurrent nested atomic action design has also been made and the design tested, but concurrency is not possible until both the objects and atomic actions are fully distributed since objects cannot be shared between processes unless the objects are managed by a

server. This restriction is due to the fact that multi-threading is not provided by the process model of the underlying operating system. If the underlying operating system supported multi-threading then concurrency within atomic actions could be implemented without the need for distribution.

The lifetime of atomic actions was implicitly assumed throughout this thesis to be short, yet there is growing interest in, and need for, long-running atomic actions that span days rather than seconds. If a node crash occurs during the execution of a long-running atomic action constructed in the manner described in this thesis, then a considerable amount of work may be lost as a result of the failure atomicity of the atomic action design and the fact that permanence of effect is only associated with committing top-level atomic actions. To reduce the amount of work lost, concepts such as *top-level nested* atomic actions have been introduced. A top-level nested atomic action is a special type of nested atomic action that has the permanence of effect associated with its commitment, even though it is nested with another atomic action. The problem with such an approach however, is that serialisability is weakened as dependencies between atomic actions that employ objects committed by a top-level nested atomic action are not maintained, and are therefore not controlled by the outcome of the parent action of the top-level nested atomic action. Other approaches to long-running actions are possible, for instance the Arjuna project is considering the concept of *glued atomic actions* [Wheater 88b] which are top-level atomic actions that appear to execute immediately after one another. By this means, the commitment of an atomic action acts as a checkpoint but it remains to be seen whether applications can be structured in such a way that glued actions may be used.

Implementing atomic actions using a class introduces problems not normally associated with atomic actions. In particular, instances of the class `AtomicAction` must be declared before the operations the class provides can be

used, and the boundaries of the atomic action are not enforceable. A simple solution to these problems based upon the use of pre-processor macros was described in chapter three. A superior solution to the boundary problem would be possible however, if the implementation language provided exception handling constructs, enabling the atomic action class to define an exception context.

The work described in this thesis has been adopted by the Arjuna project, which is currently developing a multicast remote procedure call mechanism and set of multicast primitives [Hedayati 88]. A multicast primitive allows a group of objects to be controlled by a single primitive invocation. The intention of the project is to employ the multicast primitives to manage the replication of objects and commitment of atomic actions. To ensure that the group management operations provided by the multicast primitives are recoverable, a suitable management record has been designed. Once the multicast primitives are available, this design, and an atomic action design that employs the multicast primitives for controlling the commit or abort, will be implemented.

A current limitation of the Arjuna programming system is that it provides no support for recovering from an application program (process) crash. Given the nested atomic action model employed by Arjuna, the only time a process crash will result in inconsistencies is during the commitment of a top-level atomic action. The mechanisms described in this thesis that support the commitment of a top-level atomic action have been designed to handle node crashes, not process crashes. As a result, the implementation of the commit protocol and crash recovery program would require changes, in addition to the ability of the system to detect when a process has crashed. This is an area of development that the project is currently considering.

Earlier in this section, a limitation of the environment used to support the classes described in this thesis was discussed. The limitation was the inability to execute concurrent lightweight processes within the main process executing an application. In addition to this limitation, there are a number of other limitations which could be resolved by changing the base programming environment (the language C++ and the UNIX operating system). One is the lack of support for the storage of objects, since the operating system supports a conventional file system. The resulting programming system is therefore not uniform because components of the system which should be persistent objects (such as the source and executable forms of a program) must be stored and managed in files. A more suitable environment would be a persistent object system where all permanent state is maintained as persistent objects, such as that supported by the REKURSIV architecture (described in chapter two). Alternatively a more suitable operating system, which uses conventional hardware and supports objects, such as Ameoba [Mullender and Tannenbaum 85] or Mach [Jones and Rashid 86], could be used.

The advantages of the language C++ is that it is an efficient implementation language, and can be easily ported to different hardware configurations. The disadvantages lie in its ancestry, since it is a superset of the language C. The result, is a non-uniform type system, were variables may be objects (instances of classes constructed using the language) or primitive types. Libraries are becoming available so that all variables may be objects, a good example being the OOPS library [Gorlen 88] which is effectively an implementation of the Smalltalk-80 class hierarchy. Even so, the static binding and lack of dynamic loading ensure that an extensible system cannot be constructed. An example of this situation is the object store. To construct a true object store (i.e. one that takes and returns objects rather than persistent data) is not possible in C++ because the implementation of the object store would have to know all possible

classes as the language does not support dynamic loading, or full dynamic binding. It is recognised [Morrison *et al.* 88] that to support persistent object systems, languages that support flexible binding mechanisms (both static for efficiency and dynamic for extensibility) are needed.

Given the above limitations, a more suitable environment would be an object-oriented language that has a flexible binding mechanism supported by an operating system that supports objects. If the mechanisms described in this thesis were limited to a particular language or operating system then changes in the programming environment would not be possible. Fortunately, one of the aims of this thesis was to produce these mechanisms is a manner that could be generalised to other environments. Since this aim has been met, movement to a more suitable environment is currently being considered.

References

[Allchin 83]

J.E. Allchin, "An Architecture for Reliable Decentralized Systems", Ph.D Thesis, Technical Report GIT-ICS-82/23, School of Information and Computer Science, Georgia Institute of Technology, September 1983.

[Allchin and McKendry 83]

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

[Almes *et al.* 85]

G.T. Almes, A.P. Black, E.D. Lazowska, and J.D. Noe, "The Eden System: A Technical Review", *IEEE Transactions On Software Engineering*, Vol. SE-11, No. 1, pp. 43-59, January 1985.

[Anderson *et al.* 78]

T. Anderson, P.A. Lee, S.K. Shrivastava, "A Model of Recoverability in Multilevel Systems", *IEEE Transactions on Software Engineering*, Vol. SE-4, No. 6, pp. 486-494, November 1978.

[Anderson and Kerr 76]

T. Anderson and R. Kerr, "Recovery Blocks in Action: A System Supporting High Reliability", *Proceedings of the 2nd IEEE International Conference on Software Engineering*, pp. 447-457, October 1976.

[Anderson and Lee 81]

T. Anderson and P.A. Lee, *Fault Tolerance, Principles and Practice*, Prentice-Hall, 1981.

[Anderson and Lee 82]

T. Anderson and P.A. Lee, "Fault Tolerance Terminology Proposals", *IEEE Digest of papers, FTCS-12*, pp. 29-33, June 1982.

[Anyanwu 85]

J.A. Anyanwu, "A Reliable Stable Storage System for UNIX", *Software-Practice and Experience*, Vol. 15, No. 10, pp. 973-990, October 1985.

[Anyanwu 86]

J.A. Anyanwu, "A Crash Resistant Unix File System", *Software-Practice and Experience*, Vol. 16, No. 2, pp. 107-118, February 1986.

[Atkinson *et al.* 83a]

M.P. Atkinson, P.J. Bailey, K.J. Chisholm, P.W. Cockshot and R. Morrison, "An Approach to Persistent Programming", *The Computer Journal*, Vol. 26, No. 4, pp. 360-365, 1983.

[Atkinson *et al.* 83b]

M.P. Atkinson, P.J. Bailey, K.J. Chisholm, W.P. Cockshot and R. Morrison, "PS-Algol Papers", Persistent Programming Research Report 2, Dept. of Computer Science, University of Edinburgh and Dept. of Computational Science, University of St. Andrews, May 1983.

[Atkinson and Buneman 87]

M.P. Atkinson and O.P. Buneman, "Type and Persistence in Database Programming Languages", *ACM Computing Surveys*, Vol. 19, No. 2, pp. 105-190, June 1987.

[Atkinson and Morrison 87]

M.P. Atkinson and R. Morrison, "Polymorphic Names, Types, Constancy and Magic in a Type Secure Persistent Object Store", *Proceedings of the Workshop of Persistent Object Systems: their design, implementation and use*, Appin, Scotland, pp.1-12, August 1987.

[Balter *et al.* 88]

R. Balter, D. Decouchant, A. Freyssinet, S. Krakowiak, M. Meysembourg, C. Roisin, X. Rousset de Pina, R. Scioville, and G. Vandôme, "Guide: an object-oriented distributed operating system", Technical Report, Centre de Recherches Bull, 1988

[Banâtre *et al.* 86]

J-P. Banâtre, M. Banâtre, G. Lapalme, and F. Ployette, "The Design and Building of Enchère, A Distributed Electronic Marketing System", *Communications of the ACM*, Vol. 29, No. 1, January 1986.

[Barman and Crawley 87]

H.J. Barman and S.C. Crawley, "Flexibility in a Persistent Object-based Type System", *Proceedings of the Workshop of Persistent Object Systems: their design, implementation and use*, Appin, Scotland, pp.233-245, August 1987.

[Birman 86]

K.P Birman, "ISIS: A System for Fault-Tolerant Distributed Computing", Technical Report TR 86-744, Department of Computer Science, Cornell University, April 1986.

[Birman and Joseph 87]

K.P Birman and T.A. Joseph, "Exploiting Virtual Synchrony in Distributed Systems", Technical Report TR 87-811, Department of Computer Science, Cornell University, February 1987.

[Birrell and Nelson 84]

A.D. Birrell and B.J. Nelson, "Implementing Remote Procedure Calls", *ACM Transactions on Computer Systems*, Vol. 2, No. 1, pp. 39-59, February 1984.

[Black 85]

A.P. Black, "Supporting Distributed Applications: Experience with Eden", *Proceedings of the 10th ACM Symposium on Operating Systems Principles*, Special Issue *ACM Operating Systems Review*, Vol. 19, No.5, December 1985.

[Black *et al.* 86]

A. Black, N. Hutchinson, E. Jul, and H. Levy, "Object Structure in the Emerald System", *Proceedings of the Object-Oriented Programming Systems, Languages and Applications (OOPSLA '86) Conference*, Sept. 29 - Oct. 2, Special Issue SIGPLAN Notices, Vol. 21, No. 11, pp. 78-86, November 1986.

[Black *et al.* 87]

A. Black, N. Hutchinson, E. Jul, H. Levy, and L. Carter, "Distribution and Abstract Types in Emerald", *IEEE Transactions on Software Engineering*, Vol. SE-13, No. 1, pp. 65-76, January 1987.

[Blair *et al.* 86]

G.S. Blair, J.A. Mariani, and J.R. Nicol, "COSMOS - A Nucleus for a Program Support Environment", Technical Report, Department of Computing, University of Lancaster, 1986.

[Bloom and Zdonik 87]

T. Bloom and S.B. Zdonik, "Issues in the Design of Object-Oriented Database Programming Languages", *Proceedings of the Workshop of Persistent Object Systems: their design, implementation and use*, Appin, Scotland, pp. 495-517, August 1987.

[Bullis *et al.* 86]

B. Bullis, P. O'Brien, and C. Schaffert, "Adding Database Capability to Trellis/Owl", Technical Report DEC-TR-440, Object-Based Systems Group, Digital Equipment Corporation, April 1986.

[Cockshot *et al.* 84]

W.P. Cockshot, M.P. Atkinson, K.J. Chisholm, P.J. Bailey, and R. Morrison, "Persistent Object Management System", *Software-Practice and Experience*, Vol. 14, No. 1, pp. 49-71, January 1984.

[Cox 86]

B.J. Cox, *Object Oriented Programming*, Addison Wesley, 1986.

[Davidson *et al.* 84]

S.B. Davidson, H. Garcia-Molina, and D. Skeen, "Consistency in a Partitioned Network: A Survey", Technical Report TR 84-617, Department of Computer Science, Cornell University, June 1984.

[Davies 73]

C.T. Davies, "Recovery Semantics for a DB/DC System", *Proceeding of the ACM Annual Conference*, Atlanta, Georgia, pp. 136-141, August 1973.

[Detlefs *et al.* 87]

D. Detlefs, M. Herlihy, and J. Wing, "Inheritance of Synchronization and Recovery Properties in Avalon/C++", Technical Report CMU-CS-87-133, Department of Computer Science, Carnegie-Mellon University, March 1987.

[Dixon *et al.* 87]

G.N. Dixon, S.K. Shrivastava, and G.D. Parrington, "Managing Persistent Objects in Arjuna: A System for Reliable Distributed Computing", *Proceedings of the Workshop of Persistent Object Systems: their design, implementation and use*, Appin, Scotland, pp.246-265, August 1987.

[Dixon and Shrivastava 87]

G.N. Dixon and S.K. Shrivastava, "Exploiting Type Inheritance Facilities to Implement Recoverability in Object Based Systems", *Proceedings of the Sixth IEEE Symposium on Reliability in Distributed Software and Database Systems*, Williamsburg, Virginia, pp. 107-114, 17–19 March 1987.

[Eswaran *et al.* 76]

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

[Goldberg and Robson 83]

A. Goldberg and D. Robson, *Smalltalk-80: the language and its implementation*, Addison-Wesley, 1983.

[Gorlen 87]

K.E. Gorlen, "An Object-Oriented Class Library for C++ Programs", *Software Practice and Experience*, Vol. 17, No. 12, pp. 899-922, December 1987.

[Gray 78]

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

[Halbert and O'Brien 86]

D.C. Halbert and P.D. O'Brien, "Using Types and Inheritance in Object-Oriented Languages", Technical Report DEC-TR-437, Digital Equipment Corporation, 1986.

[Harland *et al.* 86]

D.M. Harland, H.I.E. Gunn, I.A. Pringle and B. Beloff, "REKURSIV - An architecture for Artificial Intelligence", *Proceedings of AI Europe*, Wiesbaden, September 1986.

[Harland and Beloff 87]

D.M. Harland and B. Beloff, "OBJEKT - A Persistent Object Store With An Integrated Garbage Collector", *ACM Sigplan Notices*, Vol. 22, No. 4, pp. 70-79, April 1987.

[Hedayati 88]

F. Hedayati, "Multicast Primitives Supporting a Large Class of Applications in Distributed Computing Systems", Ph.D. Thesis, Computing Laboratory, University of Newcastle upon Tyne, in preparation.

[Herlihy and Wing 86]

M.P. Herlihy and J.M. Wing, "Avalon: Language Support for Reliable Distributed Systems", Technical Report CMU-CS-86-147, Department of Computer Science, Carnegie-Mellon University, September 1986.

[Horning *et al.* 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, pp. 177-193, Springer-Verlag, 1974.

[Jones and Rashid 86]

M.B. Jones and R.F. Rashid, "Mach and Matchmaker: Kernel and Language Support for Object-Oriented Distributed Systems", *Proceedings of the Object-Oriented Programming Systems, Languages and Applications (OOPSLA '86) Conference*, Sept. 29 - Oct. 2, Special Issue SIGPLAN Notices, Vol. 21, No. 11, pp. 67-77, November 1986.

[Kenley 86]

G.C. Kenley, "An Action Management System for a Decentralized Operating System", M.Sc. Thesis, Technical Report GIT-ICS-86/01, School of Information and Computer Science, Georgia Institute of Technology, January 1986.

[Kernighan and Ritchie 78]

B.W. Kernighan and D.M. Ritchie, *The C Programming Language*, Prentice-Hall, 1978.

[Lampson and Sturgis 76]

B.W. Lampson and H.E. Sturgis, "Crash Recovery in a Distributed Data Storage System", Technical Report, Xerox Palo Alto Research Center, Palo Alto, California, 1976.

[LeBlanc and Wilkes 85]

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

[Lee 83]

P.A. Lee, "Exception Handling in C Programs", *Software-Practice and Experience*, Vol. 13, No. 5, pp. 389-405, May 1983.

[Liskov 84]

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

[Liskov 88]

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

[Liskov *et al.* 81]

B. Liskov, R. Atkinson, T. Bloom, E. Moss, C. Schaffert, R. Schiefler and A. Snyder, *CLU Reference Manual*, Lecture Notes in Computer Science 114, Eds. Goos and Hartmanis, Springer-Verlag, Berlin, 1981.

[Liskov *et al.* 87]

B. Liskov, D. Curtis, P. Johnson, and R. Scheifler, "Implementation of Argus", *Proceedings of the 11th ACM Symposium on Operating Systems Principles*, Special Issue SIGOPS, Vol. 21, No. 5, pp. 111-122, November 1987.

[Liskov and Scheifler 83]

B. Liskov and R. Scheifler, "Guardians and Actions: Linguistic Support for Robust, Distributed Programs", *ACM TOPLAS*, Vol. 5, No, 3, pp. 381-404, July 1983.

[Lomet 77]

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

[Metcalfe and Boggs 76]

R.M. Metcalfe and D.R. Boggs, "Ethernet: Distributed Packet Switching for Local Computer Networks", *Communications of the ACM*, Vol. 19, No. 7, pp. 395–403, July 1976.

[Merlin and Randell 78]

P.M. Merlin and B. Randell, "Consistent State Restoration in Distributed Systems", *Digest of Papers, IEEE FTCS-8*, pp. 129-134, June 1978.

[Morrison 79]

R. Morrison, "S-Algol reference manual", Technical Report CS 79/1, Department of Computer Science, University of St. Andrews, 1979.

[Morrison *et al.* 88]

R. Morrison, M.P. Atkinson, A.L. Brown, and A. Dearle, "Bindings in Persistent Programming Languages", *ACM SIGPLAN Notices*, Vol. 23, No. 4, pp. 27-34, April 1988.

[Moss 81]

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

[Moss 87]

E. Moss, "Experience Interfacing an Object Oriented Language and an Object Server", *Proceedings of the Workshop of Persistent Object Systems: their design, implementation and use*, Appin, Scotland, August 1987.

[Mullender and Tanenbaum 86]

S.J. Mullender and A.S. Tanenbaum, "The Design of a Capability-Based Distributed Operating System", *Computer Journal*, Vol. 29, pp. 289-299, August 1986.

[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 *et al.* 85]

E. Nett, K. Großpietsch, A. Jungblut, J. Kaiser, R. Kröger, W. Lux, M. Speicher, and H. Winnebeck, "Profemo: Design and Implementation of a Fault Tolerant Distributed System Architecture", GMD-Studien, Nr. 100, June 1985.

[Nett *et al.* 86]

E. Nett, R. Kröger, and J. Kaiser, "Implementing a General Error recovery Mechanism in a Distributed Operating System", *Digest of Papers, IEEE FTCS-16*, Vienna, Austria, pp. 124-129, 1-4 July 1986.

[Nicol *et al.* 87]

J.R. Nicol, G.S. Blair, and J. Walpole, "Operating System Design: Towards a Holistic Approach?", *ACM SIGOPS*, Vol. 21, No. 1, pp. 11-19, January 1987.

[O'Brien *et al.* 86]

P. O'Brien, B. Bullis, and C. Schaffert, "Persistent and Shared Objects in Trellis/Owl", *Proceedings of the 1986 International Workshop on Object-Oriented Database Systems*, Asilomar, California, pp. 113-123, September 1986.

[Panzieri and Shrivastava 88]

F. Panzieri and S.K. Shrivastava, "Rajdoot: A Remote Procedure Call Mechanism Supporting Orphan Detection and Killing", *IEEE Transactions on Software Engineering*, Vol. SE-14, No. 1, pp. 30-37, January 1988.

[Parrington 88]

G.D. Parrington, "Management of Concurrency in a Reliable Object-Oriented Computing System", Ph.D thesis, Computing Laboratory, University of Newcastle upon Tyne, in preparation.

[Parrington and Shrivastava 88]

G.D. Parrington and S.K. Shrivastava, "Implementing Concurrency Control for Robust Object-Oriented Systems", *Proceedings of the Second European Conference on Object-Oriented Programming, ECOOP88*, August 1988.

[Randell 75]

B. Randell, "System Structure for Software Fault Tolerance", *IEEE Transactions on Software Engineering*, Vol. SE-1, No. 2, pp. 220-232, June 1975.

[Reed 78]

D.P. Reed, "Naming and Synchronization in a Decentralized Computer System", Ph.D. Thesis, Technical Report MIT/LCS/TR-205, Laboratory for Computer Science, Massachusetts Institute of Technology, September 1978.

[Richardson *et al.* 87]

J.E. Richardson, M.J. Carey, D.J. DeWitt, and D.T.Schuh, "Persistence in EXODUS", *Proceedings of the Workshop of Persistent Object Systems: their design, implementation and use*, Appin, Scotland, pp. 96-113, August 1987.

[Ritchie and Thompson 78]

D.M. Ritchie and K. Thompson, "The UNIX time-sharing system", *Communications of the ACM*, Vol. 17, No. 7, pp. 365-375, July 1974.

[Schaffert *et al.* 86]

C. Schaffert, T. Cooper, B. Bullis, M. Kilian and C. Wilpolt, "An Introduction to Trellis/Owl", *Proceedings of the Object-Oriented Programming Systems, Languages and Applications (OOPSLA '86) Conference*, Sept. 29 - Oct. 2, Special Issue SIGPLAN Notices, Vol. 21, No. 11, pp. 9-16, November 1986.

[Schwarz 84]

P.M. Schwarz, "Transactions on Typed Objects", Ph.D Thesis, Technical Report CMU-CS-84-166, Department of Computer Science, Carnegie-Mellon University, December 1984.

[Shrivastava *et al.* 87]

S.K. Shrivastava, G.N. Dixon, and G.D. Parrington, "Objects and actions in reliable distributed systems", *IEE Software Engineering Journal*, Vol. 2, No. 5, pp. 160-168, September 1987.

[Shrivastava *et al.* 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", *Proceeding of UK IT 88 Conference*, July 1988.

[Shrivastava and Banâtre 78]

S.K. Shrivastava and J.-P. Banâtre, "Reliable Resource Allocation Between Unreliable Processes", *IEEE Transactions on Software Engineering*, Vol. SE-4, No. 3, pp. 230-241, May 1978.

[Skarra *et al.* 86]

A. Skarra, S.B. Zdonik, and S.P. Reiss, "An Object Server for an Object-Oriented Database System", *Proceedings of the 1986 International Workshop on Object-Oriented Database Systems*, Asilomar, California, pp. 196-204, September 1986.

[Spafford 86]

E.H. Spafford, "Kernel Structures for a Distributed Operating System", Ph.D. Thesis, Technical Report GIT-ICS-86/16, School of Information and Computer Science, Georgia Institute of Technology, May 1986.

[Spector *et al.* 85]

A.Z. Spector, J. Butcher, D.S. Daniels, D.J. Duchamp, J.L. Eppinger, C.E. Fineman, A. Heddaya, and P.M. Schwarz, "Support for Distributed Transactions in the TABS Prototype", *IEEE Transactions on Software Engineering*, Vol. SE-11, No. 6, pp. 520-530, June 1985.

[Spector *et al.* 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-87-129, Department of Computer Science, Carnegie-Mellon University, June 1987.

[Spector 88]

A.Z. Spector, "Camelot Release 0.98(52) [Alpha] Release Notes", Department of Computer Science, Carnegie-Mellon University, May 1988.

[Stroustrup 86]

B. Stroustrup, *The C++ Programming Language*, Addison-Wesley, 1986.

[Verhofstad 78]

J.S.M. Verhofstad, "Recovery Techniques For Database Systems", *ACM Computing Surveys*, Vol. 10, No. 2, pp. 167-195, June 1978.

[Weihl 84]

W. Weihl, "Specification and Implementation of Atomic Data Types", Ph.D. Thesis, Technical Report MIT/LCS/TR-314, Laboratory for Computer Science, Massachusetts Institute of Technology, March 1984.

[Weihl and Liskov 85]

W. Weihl and B. Liskov, "Implementation of Resilient, Atomic Data Types", *ACM Transactions on Programming Languages and Systems*, Vol. 7, No. 2, pp. 244-269, April 1985.

[Wheater 88a]

S. Wheater, "A Stub Generator for Distributed Programming using C++", Internal Report, Computing Laboratory, University of Newcastle upon Tyne, 1988.

[Wheater 88b]

S. Wheater, Ph.D. Thesis, Computing Laboratory, University of Newcastle upon Tyne, in preparation.

[Wilkes and LeBlanc 86]

C.T. Wilkes and R.J. LeBlanc, "Rationale for the Design of Aeolus: A Systems Programming Language for an Action/Object System", Technical Report GIT-ICS-86/12, School of Information and Computer Science, Georgia Institute of Technology, December 1986.