Management of Concurrency in a Reliable Object-Oriented Computing System

Graham D. Parrington

Ph.D Thesis

The University of Newcastle upon Tyne

Computing Laboratory

July 1988

Abstract

Modern computing systems support concurrency as a means of increasing the performance of the system. However, the potential for increased performance is not without its problems. For example, lost updates and inconsistent retrieval are but two of the possible consequences of unconstrained concurrency. Many concurrency control techniques have been designed to combat these problems; this thesis considers the applicability of some of these techniques in the context of a reliable *object-oriented* system supporting *atomic actions*.

The object-oriented programming paradigm is one approach to handling the inherent complexity of modern computer programs. By modeling entities from the real world as *objects* which have well-defined interfaces, the interactions in the system can be carefully controlled. By structuring sequences of such interactions as atomic actions, then the consistency of the system is assured. Objects are *encapsulated* entities such that their internal representation is not externally visible. This thesis postulates that this encapsulation should also include the capability for an object to be responsible for its own concurrency control.

Given this latter assumption, this thesis explores the means by which the property of *type-inheritance* possessed by object-oriented languages can be exploited to allow programmers to explicitly control the level of concurrency an object supports. In particular, a object-oriented concurrency controller based upon the technique of two-phase locking is described and implemented using type-inheritance. The thesis also shows how this inheritance-based approach is highly flexible such that the basic concurrency control capabilities can be adopted unchanged or overridden with more type-specific concurrency control if required.

Acknowledgments

Firstly, my thanks must go to my supervisor, Professor Santosh Shrivastava, who gave me the opportunity to work in the area of object-oriented systems as a member of the Arjuna project, and in addition made many helpful comments upon the content of this thesis. I would also like to thank Professor Brian Randell and particularly Professor Pete Lee for their diligent and painstaking reading of previous drafts of this thesis. Their efforts have contributed greatly to its current presentation. Professor Harry Whitfield also deserves thanks for his patience in the early years of my research.

Thanks too must go to all of my colleagues on the Arjuna project for the many profitable discussions we have had over the years. Many other members of the Computing Laboratory to numerous to mention individually have also made my tenure here more pleasant. Thanks to you all.

Finally, the support and patience of my wife Susan, and that of my parents, throughout the long years of this research deserve a special mention.

Financial support for the work described in this thesis was provided by a grant from the UK Science and Engineering Research Council and a Serc/Alvey grant in software engineering.

Table of Contents

List of Figures

List of Tables

Chapter 1
Introduction

Over the past few decades increasing reliance has been placed upon computers to such an extent that today many organisations are totally dependent on the correct functioning of their computer systems. Enterprises such as banks and airlines simply could not function without the data contained in their computer systems being available, up to date and correct at all times.

Much of the burden of ensuring this correctness inevitably falls upon individuals since it is people that design and write the programs that execute upon the computer hardware and also design and construct the hardware itself. No matter how carefully programs are designed and tested, they are nonetheless vulnerable to external factors over which the programs have no control - in particular they may be susceptible to *interference* from other programs and possibly *failures* of the hardware and also of the software. Providing failure free hardware and software is not a sufficient solution to these problems because although a program may behave correctly when executed in isolation, this behaviour may not be repeatable when the program is executed in a multiprogramming or multiprocessing environment.

Multiprogramming is inescapable in modern computers; without it the majority of the power of the computer would be wasted. Multiprogramming allows programs to execute seemingly in parallel (or *concurrently*) with each other. If the computer has multiple independent central processing units (CPUs) then truly parallel execution can occur as each program can be executed upon a different CPU. Concurrent execution of programs can lead to problems if shared data is being manipulated by the programs in question since the execution of one program could *interfere* with the execution of another by changing the value of the data shared between the programs in a seemingly arbitrary fashion. Thus,

apparently correct programs (that is, programs that obey their specification when executed in isolation) can behave in an unexpected (and often unrepeatable) fashion. Avoiding this problem requires the use of some form of *concurrency control* technique.

In addition to the problems caused by concurrency, computer systems are also subject to many types of failure. These failures, which may affect the hardware and also the software, can either halt a program or force it to behave in an abnormal fashion (that is, the program no longer obeys its specification) at any point in its execution, leading to potential inconsistencies in the system. Once such a failure has occurred and been detected, the system must be able to *recover* from the effects of the failure so that prior to the recommencement of normal operation the state of the system is once again consistent.

What constitutes consistency is of course system and application dependent. However, it is assumed that there exists a set of *a priori* constraints upon the system which suffice to determine if any given state of the system is consistent or not. Furthermore, it is also assumed that given a consistent system state then the applications programs will maintain this state or move the system into a new, equally consistent, state. This implies that the possibility of design faults in the system is not being considered. Due to the complexity of modern computer systems, expecting them to be free of design faults may appear unrealistic, however, techniques exist to aid in coping with such design faults and since this topic is orthogonal to that considered in this thesis, the interested reader is referred to [*Lee and Anderson 85*].

When the resources being used by programs are distributed over a set of computers there can be further complications since there may be a high probability that some component in the distributed system is not functioning, or is not functioning correctly. While a distributed environment offers opportunities

that can be exploited to achieve higher reliability and parallelism, the problem remains as to how a distributed system should maintain consistency in the face of concurrency and failures.

In addition to the complexities introduced by such problems as interference and failure, the task of writing a correct application program has itself become increasingly complex. As computers have been introduced into more areas of human endeavour, the tasks that they must perform have become more sophisticated. Consequently, programs and systems consisting of hundreds of thousands of lines of code are not uncommon.

Overcoming all of these problems is extremely difficult and in order to stand any chance of success the overall task must be divided into more manageable sub-tasks. This is the basic strategy of *divide and conquer*. By breaking the entire task into a set of pieces, each of which may in turn be further broken down, it is hoped that eventually the individual pieces become small enough (and simple enough) to be comprehendable and thus implementable as part of a computer program. This process of decomposition requires discipline in both the design and coding of such systems. Many disciplines, some with formal underpinnings, are available. In this thesis the use of one of these approaches is examined - the so-called *object-oriented* paradigm [*Jones 78*]. This discipline will be described further in the following section.

Having overcome the sheer complexity of the system in design terms, the problems of concurrency and failure still remain. In order to overcome these problems a computing abstraction known as an *Atomic Action* may be utilised. Atomic actions have several useful properties that make them well suited for this purpose. Section 1.2 of this chapter will briefly describe why the combination of atomic actions with the object-oriented paradigm is useful.

1.1 Object-Oriented Programming

The fundamental construct used in object-oriented programming is the *Object*. An object is a logical or physical entity that is self-contained and which provides a well defined interface that permits orderly interaction between the object and other objects. Breaking the system down into a set of objects provides a way of managing the complexity of the programming task. Each object is an *instance* of some *type* and consists of some data structure (its *instance variables*) and a set of operations (its *methods*). The interface defines the visibility of these operations and instance variables, to other objects. An object-oriented program then consists of a set of such objects and a sequence of operations upon those objects. By structuring programs using the object-oriented paradigm various benefits ensue including modular design and the possibility of software reusability. In addition, since an object is self-contained and provides a well defined interface then the object-oriented style of programming directly supports the notions of data abstraction and information hiding, because the details of how an object is implemented is completely hidden (unless explicitly revealed).

The above is not, however, a complete definition of object-oriented programming since it could equally well be fulfilled by any language that provides *user-defined* types (sometimes called *abstract data types* or *ADT*'s), for example, Ada [*Ada 80*]. According to Stroustrup [*Stroustrup 87a*], what distinguishes object-oriented programming from programming using user-defined types, is the ability to make the commonality between various types explicit. Thus two types representing specific shapes (say a circle and a square) could be specialisations of a more *generic type* shape, and thus may have many operations in common that can be shared. Such commonality is expressed in object-oriented programs via *inheritance*.

Perhaps the most well known object-oriented language and system is Smalltalk-80™ [*Goldberg and Robson 83*]; however, there are several other systems and languages that claim to be object-oriented, for example, Clu [*Liskov et al. 79*], CommonLoops [*Bobrow et al. 86*], Flavors [*Moon 86*], C++ [*Stroustrup 86*], Objective-C [*Cox 86*], and Trellis/Owl [*Schaffert et al. 86*]. In fact the earliest such language is Simula-67 [*Birtwhistle et al. 73*] which, while being based upon Algol-60, pioneered many of the features considered essential in an object-oriented language. Its use of *classes* to define types and the notion of *virtual functions* which enable the specialisation of inherited capabilities have been carried over into C++.

Object-oriented programming has been an active area of research for many years, and there are many notable systems and languages that claim to support it. However, there does not as yet appear to be an agreed definition of precisely what object-oriented programming is. For the purposes of this thesis it is assumed that for a programming language to be called object-oriented it has at least the following properties:

- *Data Abstraction.* The available set of operations provided by a type provides the only means by which instances of the type (objects) may be manipulated. The user of the type usually does not know how the operations are implemented nor how the type is represented. Data abstraction allows the separation of the abstract behaviour of a type from its concrete implementation.

™Smalltalk is a Trademark of Xerox Corporation.

- *Sub-Typing.* New types can be composed out of existing types by *deriving* a new type from an old type. The newly created type is said to be a *sub-type* of the existing type (which is referred to as the *base* type of the new type).

- *Inheritance.* When a new type is created by derivation from an existing type it can *inherit* the attributes of the parent type. These inherited attributes may be left unchanged in the new type, or the new type may provide suitably modified versions of any of the attributes so that they are more applicable to instances of the new type. If a new type can have more than one parent type then it can inherit properties from all of them.

This definition of object-oriented programming is also in accordance with that of Wegner [*Wegner 86*] who states that:

object-oriented = data abstractions
+ abstract data types
+ type inheritance

For the purposes of this thesis these properties serve to define object-oriented programming. Another property often assumed necessary, that of *message passing*, is not considered to be required here. Thus the definition allows languages based on procedure calls rather than message passing to be object-oriented. Examples of such languages include Trellis/Owl, and C++.

1.2 Atomic Actions

Atomic actions are programmer defined sequences of operations upon objects that have three highly desirable basic properties that make them well suited as a method of structuring software to simplify the problems caused by both concurrency and failure (section 1.4 will describe exactly what faults are expected to be tolerated). These properties are:

- *Failure Atomicity.* Either all of the operations that constitute the action happen or none of them do. That is, if the action succeeds (*commits*) then all of the operations upon any objects manipulated under control of the atomic action will have been performed. If the actions fails (*aborts*), the effect is as though none of the operations had been performed.

- *Concurrency Atomicity.* Individual actions appear to execute in some serial order despite the fact that they may in reality have been executed concurrently. This property is also known as *Serialisability*. The effect of this property is to give the illusion that the constituent operations of the atomic action happened instantaneously from the point of view of other atomic actions.

- *Permanence of Effect.* Once an atomic action has successfully terminated, its results are permanent. This usually requires the implementation of *stable storage*.

Thus, by the use of atomic actions the programmer is freed from the burden of worrying about the undesirable effects of concurrency and failure upon the application, since the atomic action support system provides capabilities that automatically handle the problems.

1.3 Distributed Systems

The rapid rise in the number of distributed systems in the past decade can be attributed to two major forces. Technological improvements in the area of Very Large Scale Integration (VLSI) have made it possible to provide individuals with more computing power on their desktop than was available from an entire room full of equipment a mere decade ago. In addition, as the performance of computers has increased, the size and cost of them has decreased. As a result a modern personal workstation dedicated to a single user is capable of delivering 10

million instructions per second (Mips) - a far cry from the days of the old centralised, shared (and usually heavily overloaded) mainframe.

Coupled with this advance in computer technology has been an similar advance in communications capabilities. Currently, Local Area Networks (LANs) such as Ethernet [*Metcalfe and Boggs 76*] are capable of transmission speeds of 10 Megabits per second. This fact, coupled with the very low error rates that such networks possess, makes distributed systems a viable and cost-effective alternative to the traditional centralised system in many environments.

In addition, real world problems are themselves often distributed. For example, banks typically have many branches dispersed over very large areas. Such geographical distribution, because of the poor response times that might otherwise result, often motivates the distribution of computing facilities so that they are adjacent to their particular users. These issues will be covered in more detail in the following sections of this chapter.

What Constitutes a Distributed System

There are no hard and fast guidelines or definitions of what precisely constitutes a distributed system. According to Enslow [*Enslow 78*], distributed processing systems have five principle components:

- A multiplicity of general purpose resources, both physical and logical, that can be assigned to specific tasks dynamically. General purpose is important here so that systems that contain specialised processors to handle input/output, for example, are excluded.

- Physical distribution and interconnection. This requires communications over some link using a cooperative protocol. Systems that operate in a

Master/Slave relationship are not allowed because of the lack of autonomy such a relationship implies.

- A high-level operating system that unifies and integrates control of the distributed components. This does not imply that each component system must employ the same operating system. Rather each system is allowed to execute its own, but there is a well defined set of policies that governs the integrated operation of the distributed system as a whole.

- Transparency. The existence of the distributed system should be transparent to the user unless the user needs to know of the distribution for specific reasons (for example, to use local resources for efficiency). Services must thus be named in some generic fashion.

- Cooperative autonomy. Each component is an autonomous entity in its own right which agrees to cooperate with others to achieve some purpose. Agreement is important here; systems must be free to reject requests for service at any time regardless of previous behaviour.

This definition is overly strict and means that a distributed system requires distributed hardware, distributed control and distributed data.

Sloman [*Sloman 87*] relaxes this definition slightly, particularly with respect to transparency and concludes that:

> "*A distributed processing system is one in which several autonomous processors and data stores supporting processes and/or databases interact in order to cooperate to achieve an overall goal. The processes coordinate their activities and exchange information by means of information transferred over a communications network.*"

This latter definition captures the essential qualities of a distributed system. Such a system is considered to be made up of a number of autonomous *nodes* (abstract computers) connected by, and communicating over, some communications medium an example of which is illustrated in Figure 1-1. New

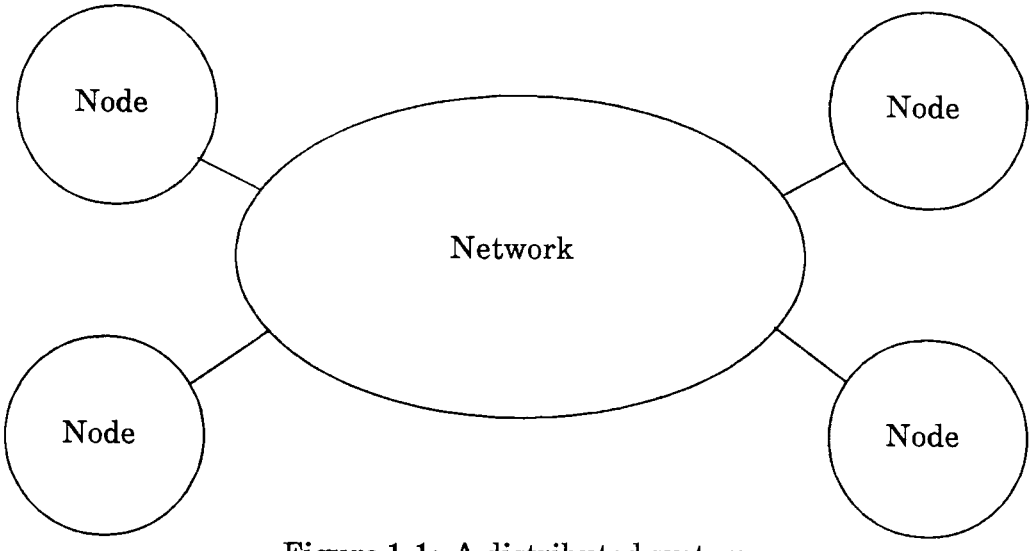

Figure 1-1: A distributed system

nodes may be added (though not removed except in special cases) to the distributed system at any time.

Each node (Figure 1-2) consists of one or more processors, together with associated storage (memory) that is either *permanent* or *volatile.* Permanent storage has the property that it can be assumed not to lose its contents when the node fails (more shall be said about node failure shortly). Thus permanent storage is *stable.* Some techniques for building stable storage are described by Lampson and Sturgis [*Lampson and Sturgis 79*] and will only be briefly described here. Their approach builds stable disk storage using pairs of conventional magnetic disks that are assumed to fail independently of one another. Each disk pair represents a single logical disk. Each real disk is *carefully* updated. Careful updating requires that each disk is updated in turn and is also read immediately afterwards to ensure that the update was successful. Such an approach ensures

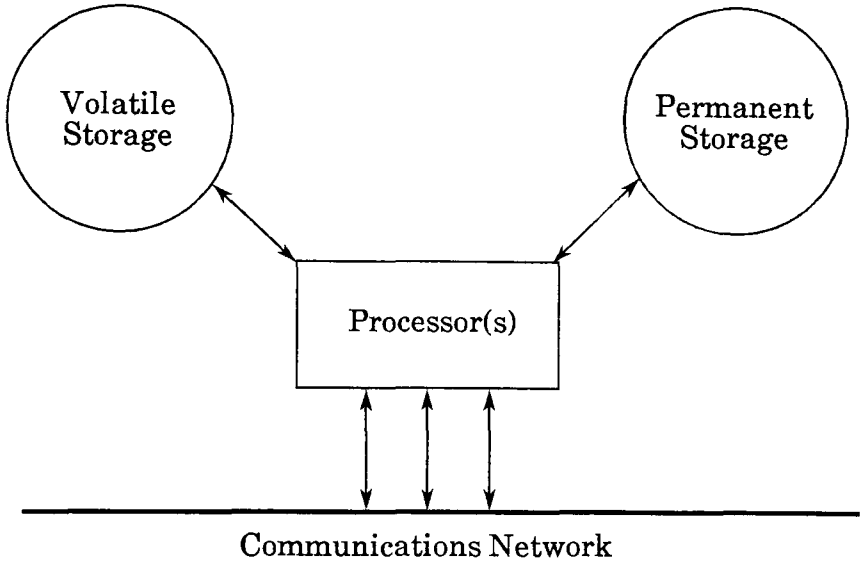

Figure 1-2: The structure of a node

that there is a high probability that at least one copy of the data is correct. Increased confidence can be gained by using more than two disks. Alternative approaches to stable storage are possible; for example, Banâtre has built such stable storage using stable memory instead of disks as part of the Enchére project [*Banâtre et al. 83*]. In contrast to permanent storage the contents of volatile storage are always assumed to be lost when the node fails; such storage is usually implemented in the main memory of the computer.

The notion of node autonomy is also very important. Any node is free to manage its own resources in any way it sees fit. All of the resources of a node are wholly under the control of that node and furthermore are only accessible and usable to others through the cooperation of the node. Nodes may not be available for a variety of causes, including failure and administrative reasons. However, when they are available they are willing to cooperate with other nodes in a fashion defined by the interfaces they present to those other nodes.

The Advantages of Distributed Systems

It is an inevitable fact of human nature that there will always be a requirement for any system to support more users, do things more quickly and more reliably, and perhaps most importantly to do things less expensively. Distributed systems are expected to meet these objectives in a way that conventional centralised systems cannot. In particular, distributed systems provide:

- *Reduced Costs.* Processing power and memory is cheap and getting cheaper each year. High quality printers and other specialised devices are not. The ability to share expensive peripherals whilst distributing processing power to where it is needed is both useful and cost effective.

- *Flexibility and Extensibility.* Should the system need extending for some reason (say to add in some new specialised device) it is usually easy to add another node into the distributed system. Such flexibility is not generally available with conventional centralised systems. In addition, by utilising standard protocols, equipment from different manufacturers can be incorporated, thus reducing the dependency on a single manufacturer.

- *Availability.* When any part of a centralised system fails then the entire system usually fails with it. Distributed systems can overcome this since individual nodes may fail without necessarily affecting the rest of the system. Furthermore if resources are replicated then the failure may not be apparent to users of the system.

- *Performance.* This encompasses both the areas of response time and throughput. Centralised systems usually have a fairly fixed performance characteristic that can only be altered in relatively static ways, for example, by upgrading to the next model of CPU. In addition this path is often

limited - there are only a finite number of faster models of CPU. In general distributed systems do not suffer from this problem since to increase performance another node can be added into the system usually with little trouble.

- *Local Control.* By allowing localised control over data and processing the system can be made more sensitive to local needs. In addition expecting a site to relinquish control over its data may be unrealistic.

The Disadvantages of Distributed Systems

Distributed system may have advantages over centralised systems, but they also have disadvantages, which include:

- *Operating Costs.* With a central site operating costs can be kept to a minimum since all of the trained staff are often located in one place. With distributed systems many more people may need to be involved to provide local support. In addition there are various problems concerned with purely operational matters, such as who is responsible for safeguarding the data by making backups periodically. In the centralised case the answer is simple; it may not be so for distributed systems. This leads to the observation that distribution often brings administrative headaches with it.

- *Development.* Developing a distributed application is a considerably more complex task than developing a non-distributed one (which is itself complex). Indeed the craft is still a topic of active research. Enforcing standards may also prove difficult, together with trying to overcome the tendency to 're-invent the wheel' at each site.

- *State of the Art.* Distributed systems are currently regarded as being *State of the Art.* As such, few people in the real world are willing to be guinea pigs!

Despite these problems, the growth of distributed systems is continuing. As more knowledge is accumulated through research and commercial experience of such systems, this growth is likely to continue for many years to come.

1.4 Programming Distributed Systems

Programming large complex applications has already been described as a difficult task. Constructing distributed applications is even more difficult due to the additional problems that distribution brings. It is one of the propositions of this thesis that the adoption of the object-oriented programming approach eases the programming of such distributed applications. Having structured the application program as a sequence of operations upon some set of objects, it should not matter to the programmer where in the distributed system the actual objects are located. Given adequate programming tools, programs that access purely local objects should look no different to those that access objects at other nodes. Thus, programming a distributed application becomes no more complex than programming a centralised application.

Distributed systems appear to have the potential for increased reliability over conventional centralised systems since they no longer possess a single point of failure. As each component in the distributed systems is assumed to be autonomous, failures in individual components should not cause failure of the entire distributed system. This is an important gain which can be exploited by providing appropriate levels of redundancy so that the distributed system can behave as if the failure had not occurred, or alternatively, the system may continue to operate but provide a degraded level of service. Even failure of the

network itself need not be catastrophic since local processing is likely still to be available.

Unfortunately, since failures do not typically affect the entire distributed system, then without care it is possible for the system to end up in an inconsistent state as work proceeds at non-faulty nodes unaware of problems elsewhere in the distributed system. In order to overcome these problems, applications should be structured as atomic actions, whose properties ensure that the applications complete successfully or appear not to have executed at all. Thus by using the object-oriented programming approach, the overall complexity of the programming task has been eased, and when this approach is used in combination with the atomic action abstraction, the problems of failure and concurrency are also eased, leaving the programmer free to concentrate on the task at hand.

The model of distributed systems adopted by this thesis regards failures of the processor or volatile memory as failures of the entire node (recall that it is assumed that permanent storage never fails). If a node fails (crashes) it is assumed to be equivalent to halting the processor; that is, the node behaves in a *fail-stop* fashion [*Schlicting and Schneider 83*]. After a crash the node is repaired within some finite period of time and restarted (the node is said to have *recovered*).

Node behaviour is thus classified as *correct* if the node is functioning; *tolerable* if it crashes and recovers, and *intolerable* otherwise. Thus, in general, permanent removal of a node from the network (except when it can be guaranteed that the node was idle and no longer needed) is classified here as intolerable behaviour. Hence, the requirement made earlier that nodes could not be removed from the distributed system and that nodes must be repaired within some finite period of time.

It is further assumed that the communication system itself can cause problems, possibly delivering messages out of order, delaying messages for arbitrary periods, corrupting messages, or even deleting messages entirely. However, it is assumed that by appropriate use of checksums (or some other similar technique) corrupt messages can be detected with high probability and rejected. Furthermore, by including the addressing information in the checksum it will be assumed that messages will only be received by their intended recipients. Thus if a message arrives, it arrives intact and at the correct node. Other problems of message duplication, etc., can be handled in well-known ways by various protocols so will not considered further here.

1.5 Aims of this Thesis

This chapter has postulated that programming distributed applications following the object-oriented paradigm is a profitable approach to adopt. It has been further suggested that the use of atomic actions can relieve the programmer from some of the burden of worrying about the possible effects of concurrency and failure. This thesis concentrates on the provision of support for one particular property of atomic actions, the property of concurrency atomicity, within a distributed object-oriented system.

The thesis shows how a concurrency controller can be designed and implemented in an object-oriented environment in a highly flexible manner that allows a wide variety of the available concurrency control techniques to be available to the programmer. In support of this claim a concurrency controller based upon a technique known as two-phase locking is designed, implemented, and its performance measured. Given this design, the thesis shows how user-defined objects may utilise it in a simple fashion such that concurrency atomicity is achieved.

The means by which this support is provided follows the object-oriented programming approach by providing a basic concurrency control type from which a user's type can be derived in the standard object-oriented fashion. The same technique has been used to provide the other properties of atomic actions but this is not described here, see [*Dixon 88*] for details. Thus, the approach adopted in this thesis is intentionally evolutionary, not revolutionary. That is, it is not the aim of this thesis to design a new programming language or operating system that supports concurrency control directly, but rather to take advantage of existing language features and systems to implement the ideas.

The thesis also describes how it is possible to override the basic system such that higher levels of concurrency can be supported based upon the programmer's knowledge of the object.

1.6 Structure of Thesis

In chapter two the problem of concurrency control is examined in greater detail, describing the problems that concurrency control sets out to solve; for example, *lost updates* and *inconsistent retrieval*. The chapter then describes many of the basic techniques by which this control has been achieved.

Chapter three is devoted to the relationship between atomic actions and concurrency control. In particular the problems that need to be solved to make concurrency control in atomic actions work are described, together with the descriptions of the implementation of atomic actions in several existing distributed systems.

Chapter four considers the object-oriented framework in greater detail and describes the characteristics such a framework has that makes it suitable for implementing reliable software. This chapter also describes the design of a lock-based concurrency controller that allows individual objects to control their own

level of concurrency. Naturally the design is itself object-oriented and this approach is contrasted with the efforts of other researchers in this area. The chapter also deals with such issues as deadlock and lock conversion.

Chapter five describes how the techniques and designs of the previous chapter have been implemented as part of *Arjuna* - a programming system for reliable distributed computing currently under development at the University of Newcastle upon Tyne. This chapter also gives some performance characteristics of this particular implementation.

Chapter six deals with how to build alternative concurrency controllers in the object-oriented environment. It describes how type-specific concurrency controllers, that exploit the programmer's knowledge of the semantics of the operations supported by an object, can be built in a simple fashion, building upon the basic concurrency controller design presented in chapter four. In addition, the requirements such concurrency controllers place upon the underlying system, in order that the increased level of concurrency can be realised are also noted.

The final chapter presents some conclusions from the work presented in this thesis and suggests where it should progress in the future.

Chapter 2
Concurrency Control Techniques

This chapter takes a closer look at the problems that concurrency control techniques should overcome and describes some of the basic techniques themselves. The topic of concurrency control has been an area of active research for many years and there is now a great depth of knowledge in the field (see [*Kohler 81, Bernstein and Goodman 81, Bernstein et al. 87*] for some comprehensive studies). In particular, a large number of different techniques have been proposed, some of which are general purpose, whereas others are only applicable in particular specialised applications. New techniques and subtle modifications to existing techniques are published regularly, particularly in database literature. The theory of concurrency control has not been neglected either, and sound mathematical proofs underlie many of the more popular methods (see for example, [*Eswaran et al. 76, Papadimitriou 79, Bernstein et al. 87*]).

Despite the wide choice of available techniques only a relatively small number have found favour so that many exist in purely theoretical form only. This chapter only concentrates upon these popular techniques; the interested reader will find many other techniques described elsewhere (see for example, [*Buckley and Silberschatz 84, Goodman and Shasha 85*]).

The majority of the studies of concurrency control have been driven by the need to access shared, centralised databases. Consequently, many of the techniques are described in the literature in database style terms. For example, programs are assumed to be manipulating data items that are basically structured as physical or logical storage entities (files, records, pages);

furthermore the access to the data is usually only classified as a *read* or a *write* access. In addition, it is usually assumed that there is only a single concurrency controller for the system. This single controller handles all requests for access to all of the data items. This centralised approach makes certain concurrency control techniques easier to implement since at any given time the concurrency controller effectively has global knowledge regarding which objects are being accessed concurrently and by which programs. The ability to gather and use such knowledge makes the detection and handling of certain problems such as deadlock far easier.

In the case of distributed systems it is still usually the case that a single concurrency controller exists per site. Furthermore, the global knowledge necessary to detect deadlock must still be acquired somehow, usually by communication between the concurrency controllers of each site, despite the fact that gathering this information is a potentially costly operation.

Consequently, in the descriptions that follow, these traditional description conventions are followed. In later chapters, however, the concurrency control techniques described in this chapter will be applied to the object-oriented environment that is really under consideration in this thesis. In particular, the notion of having only one concurrency controller per site will be abandoned in favour of having one concurrency controller per object.

2.1 The Concurrency Control Problem

Concurrency control is the act of coordinating the concurrent accesses by processes (it will be assumed in this chapter that user programs are executed by processes, which are the standard agents supplied by the operating system for this task) executing in parallel with each other to shared data such that those processes do not *interfere* with each other. Thus, concurrency control is a generalisation of the traditional problem of mutual exclusion found in operating

systems where certain data structures may only be manipulated by a single process at any moment in time.

The general problem of concurrency control is, however, somewhat more complex than simple mutual exclusion since it is often unacceptable to allow only exclusive access to a data item for performance reasons. In addition, most programs require access to multiple data items, since the value of one data item is often used to calculate the value of another. In such a situation the program would need to ensure it had exclusive access to all of the data items, otherwise the consistency of the data could be compromised.

Consistency is not the only goal of a good concurrency control technique. In addition it should also:

- permit sufficient parallelism in the system. That is, the concurrency control technique should not overly constrain the potential parallelism.
- not place too great an overhead on the system by consuming excessive amounts of resources.
- place as few constraints as possible on program structure.

2.1.1 Interference

Interference between processes can occur in many ways but two of the more common problems are known as *Lost Updates*, and *Inconsistent Retrievals*. A simple example serves to illustrate these problems further:

In this example (Figure 2-1) the deposit procedure places money into some account and is sufficiently trivial so as to appear to be correct. However, should two people attempt to execute this procedure in parallel it is possible for the account to become inconsistent.

```
procedure Deposit (Account, Amount)
begin
      current := Read(Account);
      current := current + Amount;
      Write (Account, current);
end
```

Figure 2-1: Deposit procedure

Consider the following sequence of events: Customer 1 attempts to deposit £20 into the account, while customer 2 simultaneously attempts to deposit £100. If the account currently holds £100 then the expected result is that after the two deposits the account should hold £220. However, the following shows one possible interleaving of the execution of the two transfers that renders this required state impossible.

C_1 read the account balance and gets £100

C_2 reads the account balance and gets £100

C_1 adds the amount £20 and writes £120 back into the account

C_2 adds the amount £100 and writes £200 back into the account

The end result is that the account contains £200, not £220 as it should. The problem is that C_2 read the account prior to C_1 completing its update. This phenomenon, known as the *Lost Update* problem, occurs when two processes both read an old value of some object and then both attempt to write a new value for the object.

A related problem can occur if another process is simply retrieving the value of an object. Consider the concurrent execution of the transfer and print programs shown as Figures 2-2 and 2-3.

```
procedure Transfer(Account1, Account2, Amount)
begin
      temp := Read(Account1);
      Write (Account1, temp - Amount);
      temp := Read(Account2);
      Write (Account2, temp + Amount);
end
```

Figure 2-2: Transfer procedure

```
procedure Printsum(Account1, Account2)
begin
      temp1:= Read (Account1);
      temp2 := Read (Account2);
      sum := temp1 + temp2;
      output (sum);
end
```

Figure 2-3: Print procedure

If C_1 attempts to transfer £50 from account 5 to account 9, while C_2 attempts to print the balance of the same two accounts, then the following interleaving of the execution of these two programs is possible (assume accounts 5 and 9 both initially contain £400):

C_1 reads account 5 and gets £400

C_1 subtracts £50 from the value it read and writes £350 to the account

C_2 reads account 5 and gets £350

C_2 reads account 9 and gets £400

C_1 reads account 9 and gets £400

C_2 prints the sum as £750

C_1 writes £450 into account 9.

The problem here is one of *Inconsistent Retrieval*. Because C_2 was able to retrieve the balance from account 5 after it had been updated, but retrieved the balance from account 9 prior to the corresponding update to it, then there appeared to be a loss of money. In actual fact no money had been lost and the two accounts are in fact correct.

2.2 Serialisability

The examples in the previous section, albeit simple, nonetheless showed how concurrent execution can make programs that would normally function correctly if executed in isolation, behave in an inconsistent fashion. Note that the problems only arose because of the particular order in which the operations where executed at run-time. If executed in isolation and to completion the programs would have produced the expected results.

Such problems can obviously be avoided by executing the programs strictly sequentially. However, the degradation of performance that would occur by doing so makes such an option untenable. What is required is some way of making the programs behave *as if* they had been executed sequentially. This is known as *serialisability*. More precisely, any given concurrent execution of a set of programs is *serialisable* if it is equivalent to *some* serial execution of the same programs. Attaining serialisability is the goal of many concurrency control techniques.

Serialisable executions avoid the problems outlined in the previous section as follows. Lost updates can only occur if two processes read an old value of some object prior to updating it. With a serial execution one update must read the result of the preceding update regardless of the order the updates execute in. Since a serialisable execution is equivalent to some serial execution it cannot cause lost updates.

Similar arguments can be applied to the problem of inconsistent results. Since in a serial execution the retrieval process executes either before or after the update, in a serialisable execution the inconsistency cannot arise.

There are many possible serialisable executions just as there are many possible serial executions - all of which are equally correct (assuming that the programs themselves are correct). However, there is no way to ensure that any *particular* serial order is followed without user intervention.

2.2.1 Limitations of Serialisability

Serialisability is not without its problems. In particular, it limits concurrency. Kung and Papadimitriou [*Kung and Papadimitriou 79*] show that it uses only syntactic information about programs and that higher levels of concurrency are possible if semantic knowledge is also used. In addition, serialisability introduces synchronisation problems of its own. For example, lock-based approaches can encounter deadlock and also enforcing serialisability restricts the ability of programs to directly exchange messages since such an exchange would be unserialisable.

Given these problems several researchers have considered non-serialisable approaches which are nonetheless consistency preserving. Some of these approaches are briefly examined in section 2.8 of this chapter.

2.3 Concurrency Control Techniques

Concurrency control techniques can be broadly classified into two distinct types: *Pessimistic* and *Optimistic*. Pessimistic concurrency controllers prevent potentially conflicting operations from occurring. In doing so they must always assume the worst possible case in that if two operations might conflict, the concurrency controller assumes that the conflict will happen. Optimistic concurrency controllers, on the other hand, allow free access to the data items and then attempt to determine if any conflict had occurred at some later point in time (usually when a program terminates). Thus, they assume (optimistically) that conflict will not occur, and only take action if it actually does.

In general when the concurrency controller is presented with an access request for a data item it has three possible options open to it:

- *Accept.* The access to the data item is granted immediately with the concurrency controller recording details of the request to support any later decisions it may be required to make.

- *Reject.* The access to the data item is denied. When this occurs the process attempting the access is usually aborted. Rejection of a request implies that serialisability would be compromised if the request was granted.

- *Delay.* The request cannot be granted immediately so the concurrency controller queues the request for later processing. This allows the concurrency controller some leeway with regard to later decisions but restricts concurrency.

In addition to being pessimistic or optimistic, all of the concurrency control techniques can be broadly classified as *Aggressive* or *Conservative.* An aggressive concurrency controller avoids delays and always grants requests if possible. By doing so it may reach a situation whereby it ends up rejecting other requests (since they would violate serialisability) and thus must abort the process making those requests.

Conservative concurrency controllers tend to delay requests. This makes it possible to re-order the request queue in the hope of permitting more operations to complete. This has an obvious effect on the potential level of concurrency.

Aggressive concurrency controllers work well in environments where conflicts are rare, and hence conflicts that require rejection are likely to be rarer still. On the other hand if the rate of conflict is high a conservative approach may

be better since the concurrency controller could re-order the requests to cause the least number of rejections.

2.4 Pessimistic Concurrency Control

Pessimistic approaches prevent potentially conflicting operations from occurring concurrently. Such techniques are pessimistic because they always assume the worst possible case. Simply because there is a potential conflict does not always mean that the conflict will actually occur. Consequently pessimistic approaches tend to restrict concurrency somewhat more than is necessary. This section describes several pessimistic concurrency control techniques, the first of which, has almost become the standard method of implementing concurrency control.

2.4.1 Locking

Locking is the most widely used form of concurrency control mechanism for controlling access to shared resources. The basic mechanism is extremely simple and easy to implement and has been the method of choice in the majority of existing systems.

In the simplest case there is a lock that is associated with each object which has to be acquired before the object can be accessed. If the lock is busy the requesting process generally must wait until it becomes free or be aborted.

As stated, this is no different to the traditional mutual exclusion problem, and given that there was only a single lock associated with each data item, could be handled in the same way. However, to increase concurrency it is useful to distinguish between several different types of lock depending upon how the data item is to be accessed. At the simplest level this distinction is simply between *Read* access and *Write* access. When attempting to set a lock of a given type the concurrency controller must examine each of the locks currently set to determine

if setting the requested lock would cause a conflict. If the locks do not conflict then concurrent access to the data item is permitted, resulting in an increased level of concurrency. If conflict would occur then the request must be queued until all the existing locks that conflict with the request have been released.

The notion of what constitutes conflict is fairly obvious in this case; the traditional policy that Reads conflict with Writes and Writes also conflict with other Writes is adopted. However, lock requests from the same process never conflict with each other regardless of the actual lock type. The reasons behind this are not immediately obvious. Consider some data item x; a process may read this object (and thus require and set a read lock) and may decide at some later stage to update the object (and thus require a write lock). Since reads and writes normally conflict the write lock could not be set until the read lock was released. To overcome this problem programs are allowed to *convert* their locks from one type to a stronger type (for example, a read lock can be converted to a write lock, but not vice-versa).

Obviously, in the same way that it was possible to increase concurrency by defining locks to be of read or write types, it is likely that by introducing different types of lock (and by specifying precisely how such locks conflict with each other) a further increase the level of concurrency might be possible. This idea leads to the notion of *Type-Specific* locking. This topic will be returned to in chapters four and six; for now the discussion is restricted to the basic read and write types of lock.

Processes that make use of locking must be *well-formed*; this requires that they:

- lock an object prior to accessing it.
- do not lock an object for which a conflicting lock already exists.

- eventually unlock all the objects they have locked.

2.4.2 Two-Phase Locking

The basic locking approach outlined above reveals little about when locks should be released. The most obvious approach - release the lock when manipulation of the object is complete - has the unfortunate side-effect of producing non-serialisable executions. To illustrate this consider the possible interaction of the processes P_1 and P_2.

P_1: read[x]; write[y];

P_2: write[x]; write[y];

If each object (x and y) was unlocked immediately after use the following execution history could occur:

P_1 read locks x, reads its value and unlocks it

P_2 write locks x, writes it, and unlocks it

P_2 write locks y, writes it and unlocks it

P_1 write locks y, writes it and unlocks it

Such an interaction is clearly not serialisable since it appears that the execution of P_2 follows P_1 as far as x is concerned, but precedes it as far as y is concerned.

Two-phase locking is designed to overcome this problem. The idea is to divide the acquisition and release of locks into two distinct phases as is shown in Figure 2-4. During the first phase (termed the *growing* phase) locks can only be acquired and not released. In the second phase (the *shrinking* phase) locks may only be released and no new ones acquired. In a classic paper, Eswaran *et al.* [*Eswaran et al. 76*] proved that by following this approach then serialisability was guaranteed.

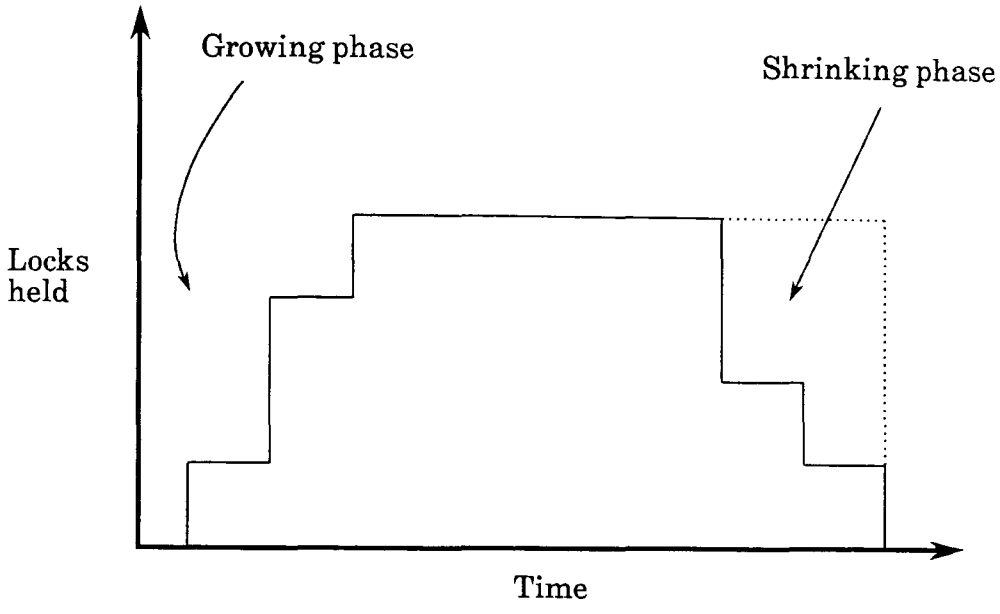

Figure 2-4: Two-phase locking

The fact that the shrinking phase may occur instantaneously (as indicated by the dotted lines) arises in an attempt to avoid the problem of *cascading aborts*. This will be considered in the next chapter. This latter approach is known as *strict two-phase locking*.

2.4.3 Conservative Two-Phase Locking

One of the major problems with two-phase locking is that the incremental acquisition of locks can lead to a situation known as deadlock (of which more will be said in section 2.4.6). Basically, deadlock occurs when two processes wait for each other to release the resources the other holds. For example, P_1 may have locked x and be wanting to lock y, while P_2 has locked y and wants to lock x. Obviously in such a case neither process is unlikely to make any progress.

This problem can be overcome be pre-declaring all the necessary locks and acquiring them in one single operation. This is the approach adopted by conservative two-phase locking. Using this technique either all the requested

locks are granted or none of them are, thus the deadlock described above is impossible. Unfortunately, there is the possibility with this approach that a particular process will never proceed because all the locks it requires are never all available at the same time.

Further complications arise with this strategy if the determination of which locks are required is decided dynamically. For example, in the program fragment of Figure 2-5 depending upon the value of the data item *a* then the program

```
    ...
read (a);
if (a < 0) then
    read (b)
else
    read (c);
    ...
```

Figure 2-5: Dynamic lock acquisition

accesses either *b* or *c*. With pre-declaration, locks on both *b* and *c* must be acquired regardless of the actual pattern of access. Finally pre-declaration really requires compiler support to determine all the objects manipulated, since leaving the choice up to the programmer is probably far to prone to error to be acceptable.

2.4.4 Multi-Granularity Locking

The locking protocols of the previous sections assume that there is no relationship between the data items being locked. However, in reality a data item could be a file, a record, or even an entire database. This leads to the notion of *granularity*; the relative size of an object. Here, a database has a *coarser* granularity than a file or a record.

Granularity affects performance. Locking at a coarse level of granularity reduces overhead due to fewer locks being requested, but it also reduces concurrency since processes are more likely to conflict. For example, two processes could not concurrently modify different records in a file since both

would require write locks on the file and thus appear to conflict. The apparently obvious solution of always locking at the finest level of granularity is not a panacea either since the overhead of doing so is likely to be significant.

A solution to this problem is to use *multi-granularity locking*. Using this approach processes lock data items at an appropriate level of granularity for their purpose. This approach was suggested by Gray *et al.* [*Gray et al. 75*]. Essentially locks are considered hierarchical, such that setting an *explicit* lock at a coarse level *implicitly* locks all of the contained objects at finer levels. Thus a read lock at the file level automatically read locks all of the records in the file also.

This is not the complete scheme, however, since there is also the need to reflect locks set at fine levels back at coarser levels. The reasons for this are as follows. Assume some process has write-locked several records in a file; in order to prevent another process read-locking the entire file (that is, setting a read lock at the next higher level) the process must indicate that locking is occurring lower down the hierarchy. One possible approach is to require that setting coarse locks causes all finer level locks to be checked for possible conflict. This would achieve the desired result but it imposes enormous overhead. An alternative approach introduces *intention* locks into the systems [*Gray et al. 75*]. Prior to setting a lock at any given level, intention locks must have also been set on all coarser levels. Thus in order to write records in a file, a process must acquire intention locks on the database and the file (in that order).

Locks are thus acquired starting at the coarsest level and working towards finer levels. They are released in the reverse order to ensure that there never arises a situation in which fine level locks are held but not coarse level ones.

The conflict rules for multi-granularity locking are more complicated than those for simple read/write locking and are given below in Figure 2-6 (from [*Bernstein et al. 87*]). The notation *ir* and *iw* represents intention-read and

Requested Mode \ Held Lock Mode	r	w	ir	iw	riw
r	y	n	y	n	n
w	n	n	n	n	n
ir	y	n	y	y	y
iw	n	n	y	y	n
riw	n	n	y	n	n

Figure 2-6: Multi-granularity locking compatibility matrix

intention-write locks respectively. The *riw* mode is a useful shorthand that is the same as owning both a read lock and an intention-write lock on the object. Its presence arises from the observation that programs frequently need both a read lock on a file (to be able to read the records within), and an intention-write lock so that it can write lock certain records to update them.

Deciding at what level of granularity locks should be applied can be complicated. If locks are always set at the finest level there are no problems. The question arises though of when to set coarse level locks. One approach requires the concurrency controller to analyse requests to determine which level of lock is appropriate. For example, if a process requests many fine level locks, the concurrency controller can *escalate* the level to a coarser one (for example, from record level locks to file level locks). Unfortunately such escalation can lead to deadlock if two processes attempt to escalate write locks on records to write locks on files.

2.4.5 Multi-Version Locking

One problem with any locking protocol is that write access precludes the possibility of read access since the two modes of access conflict with each other. One approach that overcomes this drawback is to maintain multiple versions of each object [*Stearns and Rosenkrantz 81*].

In the simplest case at most two versions of the object are maintained: a *certified* version and a *temporary* version. If a process wishes to write an object it creates a new version for its own use. Concurrent reads are permitted to read the old certified version. Since the old version is precisely the version that the failure recovery mechanisms need to maintain for their own use this approach can be quite attractive.

Implementing the two-version scheme requires the use of *certify* locks, a compatibility matrix for which is shown below in Figure 2-7.

		Held Lock Mode		
		r	w	c
Requested Mode	r	y	y	n
	w	y	n	n
	c	n	n	n

Figure 2-7: Two-version locking compatibility matrix

When a process terminates all of its write locks are automatically converted by the concurrency controller to certify locks. Since only a single write lock is allowed at any time, this ensures that a maximum of two versions of the object can exist. Furthermore after the update only a single certified version remains. Since read locks and certify locks conflict the attempt to convert a write lock to a certify lock is delayed until all read locks are released.

The scheme can be extended to allow multiple uncertified versions, however, in general only a single certified version exists.

2.4.6 Problems with Locking Protocols

The major problems associated with lock-based protocols are due to the fact that processes can be made to wait forever. Rosenkrantz *et al.* [*Rosenkrantz et al. 78*] point out that processes may wait indefinitely for four reasons:

- Deadlock.
- Infinite chain. This occurs if a process waits for a second, which in turn waits for a third, and so on as new processes enter the system.
- Waiting for a non-terminating process.
- Waiting for an infinite number of new processes that complete or abort. This can occur in the following fashion. Say P_1 waits for P_2. A new process P_3 starts and P_1 is made to wait for it also. P_2 terminates but P_1 is still blocked waiting for P_3, and so on.

By far the major problem associated with locking protocols is the fact that they are prone to *deadlock*. Deadlock occurs when two or more processes wait for resources that will never become available. In this case the processes become blocked forever and, unless external action is taken, will stay that way.

One simple approach to overcoming deadlock is to detect that a given process has been waiting for a lock for a long time and assume that it must be deadlocked with some other process. Of course what constitutes 'a long time' is system dependent. In fact the process may not be deadlocked at all, but merely held up by some other process that is taking a long time to complete.

Such problems can be overcome using a long timeout period. However, the longer the period the longer a deadlocked process must wait. Thus careful tuning is required to ensure that deadlocks are detected quickly enough but without falsely detecting deadlocks that are not really there.

Usually, a better approach is to detect deadlocks precisely. Doing this requires the construction of a *Wait-For Graph*. A wait-for graph is a directed graph with each node representing a process. There is an arc in the graph from P_i to P_j if P_i is waiting for P_j to release some lock. Deadlock detection then amounts to detecting cycles in this graph. Once a cycle is detected the concurrency controller must break it by aborting one of the processes (any will do, although for example, some consideration about the amount of work already performed by each process can be taken into account).

Building and checking a wait-for graph is a potentially expensive operation - even more so in a distributed system. Thus it is important to optimise when the deadlock check is initiated (that is, when the graph is built and checked).

Rosenkrantz *et al.* [*Rosenkrantz et al. 78*] overcome these problems by using two protocols which they term *Wait-Die* and *Wound-Wait*. These protocols combine the notions of *timestamps* (described more fully later in this chapter) with two-phase locking. Rather than make the process simply wait when a conflict is detected, the concurrency controller adopts one of two basic policies:

- *Wait-Die.* If the requester is older than the process with which it conflicts it waits, otherwise it commits suicide and aborts.

- *Wound-Wait.* If the requester is older then it attempts to *wound* the conflicting process, otherwise it waits. Wounding is an attempt to force the conflicting process to abort. This attempt may not be successful if the process was already terminating, but in either case the conflict is resolved.

Both of these protocols give priority to older processes (the notions of process age being based upon the ordering of their timestamps) since in the *Wait-Die* approach the younger process aborts itself, while in the *Wound-Wait* approach the older process tries to force a younger process to abort

2.4.7 Other Locking Protocols

There are certain structures commonly used in databases that require specialised protocols to ensure that maximum performance is achieved. In addition certain data items are often accessed more frequently than others leading to so-called *hot-spots.*

Specialised protocols have been developed for these situations including *Tree-Locking* protocols [*Bayer and Schkolnick 77, Kedem and Silberschatz 81*] amongst many others. Such protocols are considered no further in this thesis due to their specialised application environment.

2.4.8 Timestamping

A *timestamp* is simply a unique number that is drawn from a monotonically increasing sequence, and is assigned to a process. Often timestamps are derived directly from the value of the local system clock. The total ordering of timestamps ensures that if TS_1 and TS_2 are two timestamps then either $TS_1 < TS_2$ or $TS_2 < TS_1$. Timestamps are examples of what Rosenkrantz *et al.*

[*Rosenkrantz et al. 78*] call a *valid numbering scheme.* The serialisation order imposed by timestamp-based methods is that defined by the order of the timestamps themselves.

Generating timestamps in a distributed environment can be handled simply by assigning each site a unique identifier that is concatenated with the value of the local site system clock to produce the timestamp. Given such an approach then all timestamps generated at one site appear to precede or follow all timestamps generated at another site, and thus form part of a total ordering

In addition to their use in concurrency control, timestamps can also be used in deadlock detection to determine which process should be aborted to break the deadlock once it has been detected.

2.4.9 Basic Timestamping

The rules of timestamp-based concurrency control state that operations must be carried out in timestamp order, thus if any request arrives out of order it must be rejected. Basic timestamping concurrency controllers are thus aggressive in nature since operations are performed strictly first in, first out. For example, if two processes, one with timestamp 1, and the other with timestamp 5, had already manipulated some object x and a process with timestamp 2 attempted to manipulate the same object x it must be aborted otherwise the timestamp order would be violated.

2.4.10 Conservative Timestamping

Basic timestamping could abort a large number of requests if the order in which requests are processed by the concurrency controller differs badly from the timestamp order. Recall that a conservative concurrency controller attempts to queue requests to avoid this situation. Hence a conservative timestamp

controller queues requests for a while to see if any requests with earlier timestamps will arrive [*Bernstein et al. 78*].

Obviously the longer the delay imposed by the concurrency controller the less number of rejections should be generated. Unfortunately this slows the processing rate, implying that a compromise must be reached. In its ultimate form conservative timestamping produces a purely serial execution.

2.4.11 Multi-Version Timestamping

Multi-version timestamping was introduced by Reed [*Reed 78, Reed 83*]. As with multi-version locking the idea is to maintain multiple versions of each object. In Reed's scheme, object versions have a lifetime defined by two timestamps (*pseudotimes* in Reed's terminology). For example, an object might have the following history:

$$<v0[t0,t1]>, <v1[t2,t3]>, <v2[t4,t5]>$$

which implies that the object had value *v0* between pseudotimes *t0* and *t1*, *v1* between *t2* and *t3*, and *v2* between *t4* and *t5*. It is permissible for there to be gaps in the history for which no version is valid - for example, in the above history, *t1* and *t2* need not be the same pseudotime, however, *t1* must precede *t2* in pseudotime order.

Processes draw timestamps from a *pseudo-temporal environment* and it is these timestamps that determine which version of an object is visible to the process.

New versions of an object are first created as *tokens* and are only converted into proper versions when the process terminates successfully. In addition the *pseudotime* interval during which any single version is valid can be extended by reading the version. Thus in the above example, if a process with a timestamp

greater than *t1* but less than *t2* (say *t1.5*) read the object, then the validity of version *v0* would be extended from [*t0,t1*] to [*t0,t1.5*].

2.4.12 Mixed Approaches

The techniques described so far in this chapter can be considered *complete* and *pure*. They are complete since they solve conflicts between reads and writes (*r-w*) and writes and writes (*w-w*) and are considered pure because they use the same technique to solve both types of conflict.

It is, however, possible to design a concurrency controller that uses different approaches to tackle each of these two types of conflict, providing that the resulting *integrated* concurrency controller behaves in a consistent fashion and produces correctly serialisable executions. For example, in Bernstein and Goodman [*Bernstein and Goodman 81*] such an integrated controller is developed which uses two-phase locking for r-w conflicts and a derivation of timestamping called the Thomas Write Rule [*Thomas 79*] for w-w conflicts. Such mixed concurrency controllers will not be considered further.

2.5 Optimistic Concurrency Control

Optimistic concurrency control is based upon the premise that it is easier to apologise after the event than to ask permission before it. That is, whereas pessimistic approaches always obtain permission to use an object before they actually do so, optimistic approaches use the object and then determine at a later stage whether this has caused problems. The methods are optimistic because they assume that conflicts between processes are likely to be very rare such that checking for conflict later is likely to be much cheaper than preventing conflicts from occurring in the first place.

Optimistic approaches divide process execution into three stages:

- *Read Phase.* During this phase processes read objects but only write to local copies that are not visible to others.
- *Validation Phase.* Prior to making objects they have written visible to others, processes must be validated to ensure that no conflicts have occurred.
- *Write Phase.* Assuming that validation was successful the local copies of the object replace the originals and become globally visible.

In the following sections some optimistic concurrency control techniques are described. As yet none have been adopted in any system since the benefits (if they exist) have not been established.

2.5.1 Serial Validation

This concurrency control approach, described by Kung and Robinson [*Kung and Robinson 81*], assumes that there is a single concurrency controller capable of collecting all of the information it requires in order to determine if conflicts have occurred during the concurrent execution of the processes in the system.

During process execution this concurrency controller accumulates information about the *read-sets* (the objects read) and *write-sets* (the objects written) of each process. These sets are used during the validation phase to validate the process when it terminates. In addition, a monotonically increasing counter is used for timestamp-like purposes.

The protocol proceeds as follows. When a process is started the value of the counter is read and the process is assigned this value as a timestamp. At the start of the validation phase, the counter is read again and is used to determine all those processes which have terminated since the process attempting validation

started. These processes are then the ones which could have invalidated any of the objects (by creating new versions of those objects) that the validating process has read. Thus the concurrency controller examines the write-set of each of the terminated processes to see if this set intersects the read-set of the validating process. If there is an intersection then an already terminated process has written a value after the currently validating process read it. In this case validation fails. Otherwise validation succeeds and the process enters its write phase at which point the counter is increased.

Increasing the counter only after successful validation ensures that at validation time the concurrency controller can easily detect those processes that have terminated successfully since the validating process began. The validation phase and the write phase must be carried out inside a critical section to ensure consistent results; hence the method is termed *serial validation.*

2.5.2 Other Optimisitic Methods

Lausen [*Lausen 82*] has proposed a scheme integrating two-phase locking and the optimistic concurrency control of Kung and Robinson which allows processes to use either technique. His scheme requires that processes using two-phase locking have the same three phases as the optimistic approach.

Carey [*Carey 87*] has improved the performance of the standard serial validation algorithm by using timestamps instead of a counter. Also, by introducing multiple versions of objects, he has produced a protocol called *multi-version serial validation.*

2.6 Effects of Distribution on Concurrency Control

All of the techniques described so far have been designed with a centralised system in mind. However, most will adapt to a distributed environment. Strict two-phase locking adapts the easiest since in order to grant a lock the local

concurrency controller only needs to know what other locks are currently held on an object. Since objects typically only live at one site (ignoring here the possibility that an object may reside at several sites, either in part or in total) all of this information is available. However, distribution compounds the problems of deadlock detection since it becomes considerably more expensive to produce a global wait-for graph from all of the local graphs held at each site. The cost of producing this graph implies that the initiation of the deadlock detection procedure should be undertaken less frequently. Furthermore, there is the possibility of *phantom* deadlocks. These are deadlocks that appear in a global wait-for graph due to the delay in building it. For example, after a site has transmitted its local graph to the deadlock detector several processes might terminate, thus releasing the locks that they hold. However, when the global wait-for graph is built these processes still appear in it and might appear to cause deadlock despite the fact that they have since terminated.

Timestamp-based concurrency controllers are also easy to apply to distributed systems providing that the timestamps from all sites are totally ordered. A simple technique to ensure this has already been described in section 2.4.8.

An optimistic concurrency control technique has been adapted to a distributed environment by Ceri and Owicki [*Ceri and Owicki 82*] who have extended Kung and Robinson's serial validation scheme to a distributed environment.

2.7 Adaptive Concurrency Control

In his thesis, Robinson [*Robinson 82*] notes that given the proliferation of concurrency control techniques, choosing an appropriate one is difficult. Furthermore, the appropriate method could well change with system use. What

is required is some form of *adaptive* strategy that avoids commitment to any one technique.

Robinson's concurrency control scheme requires processes to generate requests to the concurrency controller for *read*, *write*, *read/write* and *validate* access to objects. Using these requests the concurrency controller maintains sufficient information about the types of access to objects and the set of objects accessed to enable it to decide how to reply to any individual request. Robinson's scheme is adaptive in that by selecting appropriate replies to requests it can behave in either an optimistic or a pessimistic manner.

For each request the concurrency controller has four possible options::

- *Wait.* The requesting process is made to wait for all conflicting processes.
- *Kill.* All conflicting processes are aborted.
- *Die.* The requesting process is aborted.
- *Grant.* The access is granted. This option is illegal in response to a validate request.

The concurrency controller selects one of these options based upon whether it detects conflict and based upon the policy that is being followed. Thus two-phase locking is equivalent to selecting *wait* for all responses if a conflict exists. Similarly the optimistic approach is obtained by selecting *grant* for read, write and read/write requests, and *kill* for validate requests.

Unfortunately this scheme requires global knowledge of the system and Robinson deliberately aims it at an environment where there is some form of global object store where the concurrency controller and all shared objects reside.

This store is maintained by a *global memory manager* (GMM) that interacts with the concurrency controller.

Another interesting aspect of this scheme is that the GMM maintains object versions so that processes that declare themselves to be read-only never need to interact with the concurrency controller at all.

2.8 Non-Serialisable Approaches

Many researchers have pointed out that serialisability is often a far stronger constraint than is really necessary. Hence there have been investigations into non-serialisable approaches. A concurrency controller that produces non-serialisable schedules must still, however, produce results that are consistent and correct. This section very briefly notes some of these efforts.

Garcia-Molina [*Garcia-Molina 83*] has proposed the idea of *semantically consistent* schedules. Processes are divided into two types: one type requires a consistent view, the other type does not. Schedules are semantically consistent if those processes that need a consistent view see such a view.

In the context of abstract data types, Allchin and McKendry [*Allchin and McKendry 83*] develop the notion of *end of action serialisability* which although serialisable at the abstract level is not so at the concrete level. They further allow non-serialisable behaviour at the abstract level by adding extra procedures to an object allowing information about object use to be gathered. That is, the object is informed when atomic actions that have used them, commit or abort. In the same area, Schwarz and Spector [*Schwarz and Spector 82*] uses semantic information to track dependencies between programs.

Perhaps the most interesting (and complicated) approach is that of Sha [*Sha et al. 83, Sha 85*] who has developed a model of consistency that is termed the *relational* model. As pointed out in [*Sha et al. 88*], when using non-serialisable

schedules it is no longer correct to assume that the execution of an action will always be consistent and correct even if the action itself is consistent and correct when executed in isolation. That is, when executing under a non-serialisable schedule the results of any execution could be different from any serial execution and so could prove to be neither consistent nor correct. In order to overcome this *modular* concurrency techniques are developed that are both consistent and correct.

One such technique uses *setwise serialisability* to allow *elementary* transactions accessing different *atomic data sets* (partitions of the data such that the consistency of each set can be maintained independently of other sets) to be combined into *compound* transactions, the execution of which is generally not serialisable.

Other work in this area includes that of Birman and Joseph [*Birman and Joseph 87*] who have proposed the notion of *virtual synchrony* which is a weaker consistency constraint than serialisability but which they argue is more applicable to distributed systems. With this scheme one event seems to happen at a time, system wide, although the actual execution is concurrent. Furthermore, event ordering is preserved in that if one event precedes another then everyone sees a consistent event ordering. Their work, and a similar notion, *virtual time* [*Jefferson 85*], provides an interesting departure from classic notions on serialisability.

2.9 Summary

This chapter has surveyed some of the many techniques available to ensure that concurrent access to an object does not result in inconsistencies. Many of the basic techniques have been known for many years, and still more are being invented, usually to solve some specialised need. Despite this, most systems still use locking as their concurrency control technique, hence this chapter's

concentration on this approach. Even new distributed systems research projects such as *Argus* [*Liskov 84, Liskov 88*] (which will be described in chapter four) have persisted in using this approach.

In reality the choice of which technique to use is complex. Some studies have been carried out (for example, [*Franaszeck and Robinson 85, Tay et al. 85, Agrawal and DeWitt 85*]) to determine the performance of the various approaches under different assumptions, but no conclusive evidence appears forthcoming.

A general consensus of opinion is that in situations of high contention, lock-based approaches are the most suitable despite their inherent problems. Whether optimistic approaches are truly viable has yet to be established. Robinson's adaptive approach appears highly flexible given an appropriate environment, however, as yet it has not been tested in anything other than a simple prototype research system. Certainly the most active area of research at the current time is in the area of non-serialisable techniques.

Chapter 3
Atomic Actions and Concurrency Control

The previous chapter described several of the available concurrency control techniques. In doing so emphasis was placed solely on the interactions of processes executing programs that referenced shared objects. It assumed a perfect environment in which failure never occurred and processes terminated correctly at all times. This was deliberate since it is part of the view of this thesis that the topic of concurrency control can be considered separately to that of handling failure. Such an assumption is of course clearly unrealistic in practice, so this chapter examines the concept of the *atomic action* and investigates the relationship that concurrency control has with it.

First, however, the concept of the atomic action is examined in more detail describing why the concept is a suitable one to use in programming reliable distributed systems. Having done so the chapter then describes how many of the concurrency control techniques of chapter two can be utilised to provide one of the key properties of the atomic action abstraction - that of *concurrency atomicity*. This is followed by an appraisal of what it means for atomic actions to be nested and the requirements that this nesting places upon concurrency control. Finally, the implementation of atomic actions in several existing systems is described to illustrate the point that different concurrency control techniques can be (and are being) used in practice.

3.1 Atomic Actions

The general concept of the atomic action has been around for many years. Probably the first reference to it was by Davies [*Davies 73*] under the name *Spheres of Control*. The name atomic action was coined by Lomet [*Lomet 77*] who described atomic actions as a means of process structuring, synchronisation, and recovery.

Davies' concept was adopted by the database community where it was rechristened the *transaction*, a term which is now considered synonymous with atomic action in most circles. The popularity of the abstraction of the atomic action can be attributed to its three fundamental properties:

- Failure Atomicity.
- Concurrency Atomicity.
- Permanence of Effect.

Failure atomicity ensures that an atomic action can only terminate in two ways: either normally, *committing* its results; or abnormally, *aborting* and producing no results at all. The net effect of the execution of an atomic action is to move the system from one consistent state to another if the atomic action commits, or to leave the system in the same consistent state that it was in before the atomic action started should the atomic action abort. The provision of this property is usually by means of *backward error recovery*, which is invoked whenever an error is detected in the system. Backward error recovery requires that the states of any objects manipulated under the control of the atomic action are restored to the corresponding states each object was in prior to the start of the atomic action. Various techniques can be used to achieve this state restoration. The simplest records the prior state of each object as a *checkpoint*, and restores this state if the atomic action aborts. Alternatively, the sequence of operations

performed upon each object can be recorded as an *audit trail*, allowing the operations to be undone if required. Such state saving and recovery can often be made automatic thus freeing the programmer from this burden.

One problem with the backward error recovery approach involves atomic actions that interact with the real world. In such situations it may be impossible to effectively restore the prior state of the system. For example, if an automatic cash dispenser gives out money as part of the execution of an atomic action in response to a client's request, then in this case simple state restoration is impossible should the action be aborted since the money has already been dispensed!

One possible solution to this particular problem is to delay performing such unrecoverable operations until the action is certain to commit and only perform such operations at that time. Essentially the operations need to be recorded as *intentions* and when the action commits all such intentions are performed only then.

Instead of attempting to perform backward error recovery another possible approach is that of *forward error recovery* [*Melliar-Smith and Randell 77*]. The idea here is not to restore the state to one which existed at some time in the past, but to attempt to modify the existing erroneous state such that it becomes consistent again after an error has occurred that caused the atomic action to abort. One form of forward error recovery is based upon the use of a *compensating* action which can be started and which attempts to undo the effects of the failed action. Obviously the success of this compensation effort depends critically upon the operation performed which can make writing such compensating actions difficult.

Forward and backward error recovery can be used in conjunction with each other. When used in this way, forward error recovery allows efficient handling of expected errors, with backward error recovery handling the more general errors that were not anticipated, or were deliberately ignored. Forward error recovery is a far more complex task that cannot usually be performed automatically, thus failure atomicity is implemented by utilising backward error recovery in the majority of existing systems. Forward error recovery does have it place; particularly in asynchronous systems, however, such systems are beyond the scope of those under consideration in this thesis, and so the interested reader is referred to [*Campbell and Randell 86*] and [*Shrivastava 85*] for further enlightenment.

Given that an atomic action has completed successfully, external consistency (defined shortly) requires that the effects of the atomic action are permanent and will not be lost due to a subsequent failure of the system. This permanence of effect property requires the provision of stable storage; storage that will survive failures of the system with a very high probability of success. Such storage is the most reliable (and so also the most expensive to use) storage available and can be considered to be at the top of a storage hierarchy that has at its bottom normal, volatile computer memory. There are, of course, intermediate levels of storage that provide various degrees of susceptibility to failure, however, for the purpose of this thesis, the basic model outlined in chapter one will be followed and it will be assumed that storage is only either volatile or stable. Lampson and Sturgis [*Lampson and Sturgis 79*] detail the design of such stable storage using pairs of conventional magnetic disks and an implementation of their technique in a UNIX† environment is described in [*Anyanwu 84*]. An

†UNIX is a registered trademark of AT&T in the USA and other countries.

alternative technique for implementing stable storage using stable memory has been used by Banâtre [*Banâtre et al. 83*] as part of the Enchére system. In addition, work was carried out at MIT as part of the SWALLOW project [*Svobodova 80, Svobodova 81*] on using write-once optical discs for stable storage.

The second property of atomic actions, concurrency atomicity, ensures that computations structured as atomic actions do not interfere with each other. It is the provision of this property that is the primary concern of this thesis. In addition, atomic actions have the property that they are consistency preserving. If a system is consistent prior to the start of an atomic action then the system will also be consistent after the action has terminated (even if the action aborted). Of course during the execution of the action such consistency will probably be violated temporarily. For example, if the transfer procedure of the previous chapter was executed under the control of an atomic action the constraint that the sum of the two accounts was constant would be maintained before and after the action executed. However, during the actual execution this constraint is temporarily violated as money is moved between the two accounts. Such constraints are user-defined and are more precisely termed *internal* consistency constraints, since they define the correctness of the actual internal state of the system.

In addition to internal consistency, atomic actions should also preserve *external* consistency. That is, the user's perception of the state of the system conforms with the actual state of the system. This implies that once a user has been informed that some action has been performed it must not be undone otherwise the user's perception of the system would be inconsistent with the true state of the system.

Such perception is, of course, influenced by the nature of the communication between the system and the user. If all communication can be delayed until an action commits then there is no problem. However, some actions will invariably solicit input from the user during their execution. Thus the user perceives the progress of the action through the system by the output it produces and the input it demands. In such cases feedback from the system is vital to inform the user of the final outcome of the action. One interesting approach to this problem was adopted by the TABS project at Carnegie-Mellon [*Spector et al. 85b*]. Within this system output could be displayed in three different fashions. While the action was executing the output was displayed with a grey background indicating its tentative nature. If the action successfully committed the output was redrawn in black, otherwise if the action aborted, lines were drawn through the output to indicate that it had been canceled. This latter approach was felt to be a more communicative way of informing the user that an action had aborted rather than simply erase the screen making the output disappear which could have been very disconcerting to the user!

3.2 Atomic Action Operations

The description of atomic actions given in the previous section leads to a natural requirement that at least the following operations must be implemented in order to support them:

- Begin Action
- Commit Action
- Abort Action

the following sub-sections outline the support from the system that each operation requires.

3.2.1 Begin Action

Atomic actions are started by a process executing the *Begin Action* operation. Failure atomicity requires that any object manipulated by the process after executing this operation must record sufficient information so that the initial state of the object may be restored later if needed (assuming a backward error recovery approach has been adopted). Often this is handled by taking a checkpoint of the initial state of the object the first time it is modified, which can be restored later if the action aborts.

When an atomic action is begun it is allocated a *atomic action identifier* that identifies the action to the system. This identifier is supplied as an implicit parameter to all of the operations that the action executes from now on, thus ensuring that all of the effects of the action can be identified should the action need to be aborted. This identifier may also used by the concurrency controller to enable it to make any decisions about the permissible level of concurrency. Such an identifier needs to be globally unique across the entire distributed system so that no two actions have the same identifier. Traditionally such identifiers are generated by concatenating together an identifier that uniquely identifies the creating site, and the current (unique) value of the local system clock.

Depending upon the system, processes may not have to explicitly start an atomic action themselves. Rather it may be implicit with the start of the process itself. Handling action commencement in such an implicit fashion guarantees that all processes in the system run as atomic actions and is thus less prone to programmer error.

3.2.2 Commit Action

Commit Action indicates the successful termination of the atomic action. This normally requires that the persistent objects (that is, those objects whose lifetime is not restricted to the lifetime of the action) affected by the atomic action are made permanent and any concurrency control information that has been collected may be usually be discarded. It may not be appropriate to discard the concurrency control information if the atomic action was nested within another atomic action. This latter point will be covered further in section 3.6 of this chapter.

3.2.3 Abort Action

Abort Action indicates that the computation executing under the control of an atomic action has failed for some reason and any changes the computation has made to the system state must be undone. Thus the state of each object modified within the scope of the action must be recovered in an appropriate manner. In the case of backward error recovery this amounts to restoring the prior state of each object.

Depending upon the system this recovery may require a lot of work, or virtually none. For example, if the current state of an object was maintained in volatile memory this state can often simply be destroyed since the proper version to restore to usually still exists on stable storage. On the other hand, if the current state has already been propagated to stable storage, either partially or fully, the prior state must be reinstalled on stable storage as the current version.

3.3 Distribution and Two-Phase Commit

Whether an action commits or aborts it is essential that the states of all of the objects that the action modified are also either committed or recovered. Ensuring this uniformity requires the use of a special *commit* protocol. The most

common protocol is the *two-phase* commit protocol [*Gray 78*]. As its name implies this protocol is split into two distinct phases. During the first phase the initiator of the commit protocol (the *co-ordinator*) broadcasts *prepare* messages to each of the objects (the *components* or *participants*) and waits for each to reply. When a participant receives the prepare message, if it is willing to commit, the participant saves sufficient information on stable storage to allow it to commit or abort under instruction from the co-ordinator and replies with an *ok* vote. Once in this state the participant has lost the right to act unilaterally and cannot proceed further until directed by the co-ordinator. If the participant is unwilling to commit it replies *no*.

The co-ordinator gathers all of the replies from the participants and then starts the second phase of the protocol. If all of the votes were *ok*, the co-ordinator records a commit flag on stable storage and broadcasts *commit* to its participants. If any vote was *no*, or no reply was received from any of the participants, then the co-ordinator records *no* on stable storage and broadcasts *abort* only to those participants that had replied *ok*. In either case the co-ordinator waits for acknowledgments from the participants it had sent messages to in the second phase, before it then terminates. Participants await the decision of the co-ordinator and act accordingly before acknowledging. This protocol is shown by the state diagrams of Figures 3-1 and 3-2 (which omit details of failure processing). In these diagrams the state transitions are labeled with the input messages that cause the transition, and the output messages that are sent as a result of the transition taking place. Messages labeled with an asterisk indicate messages sent to, or received from, all participants, while messages labeled with dashes (--) are null messages.

The above discussion has assumed that once the protocol has been started no failures will occur in the system. In actual fact, failures can occur, and the protocol will still ensure that all the participants take the same action. For

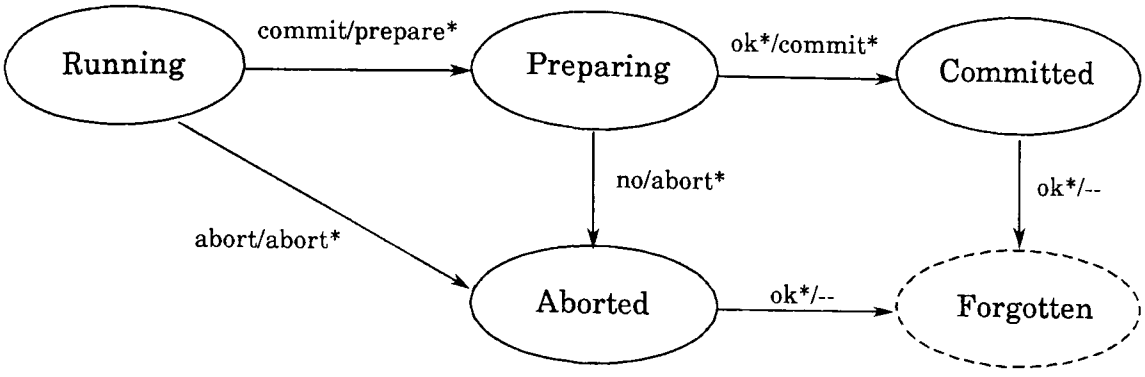

Figure 3-1: Co-ordinator state diagram

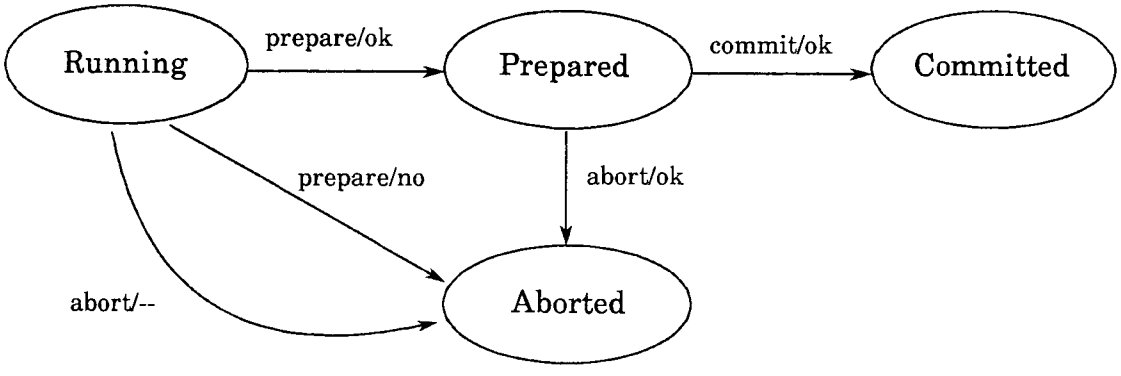

Figure 3-2: Participant state diagram

example, if the co-ordinator crashes during its first phase then upon recovery the action is considered aborted, a fact that can be discovered by the participants if they query the co-ordinator to determine the outcome of the action (something they might do if they have not received the decision from the co-ordinator after some period of time). If the co-ordinator fails during phase two, then upon recovery of the co-ordinator, the status of the action can be determined by the commit information recorded on stable storage at the end of phase one, and the

protocol can proceed. Similar arguments can be applied to failures of the participants, but are not discussed further here.

While two-phase commit is robust, it has an unfortunate problem in that it can become *blocked* if a participant, having responded *ok* to the co-ordinator, does not receive the decision of the co-ordinator for some reason (this can occur if the co-ordinator has crashed, or the network has lost or delayed the message containing the decision, etc.) since it has lost the ability to act unilaterally. Various modifications have been attempted to overcome this deficiency including the development of so-called *non-blocking* commit protocols such as the *three-phase commit* protocol [*Skeen 81*]. Other modifications have also been made in an attempt to make the protocol more efficient [*Mohan et al. 83*] but will not be discussed further.

3.4 Atomic Action Nesting

The ability to compose new programs out of existing ones is a useful technique. This reusability cuts down costs and reduces errors since existing (hopefully working) programs are used to construct new ones. The ability to compose existing atomic actions into new ones is also equally useful. Without it there would be no way to take two existing actions and combine them into a new third action, short of copying the code from each into the new action - a potentially costly operation both in terms of time and the possible errors that might result.

Another problem is that such enlarged actions might take a long time to execute; so long in fact that the 'all or nothing' property of atomic actions becomes a liability. For example,if the action requires longer to complete than the time that the system executes without a failure occurring somewhere, then the long running action can never complete.

Such problems can be overcome by allowing atomic actions to be nested as illustrated in Figure 3-3, which shows purely sequential nested actions, and Figure 3-4 which shows concurrent nested actions. Note that, by the definition of

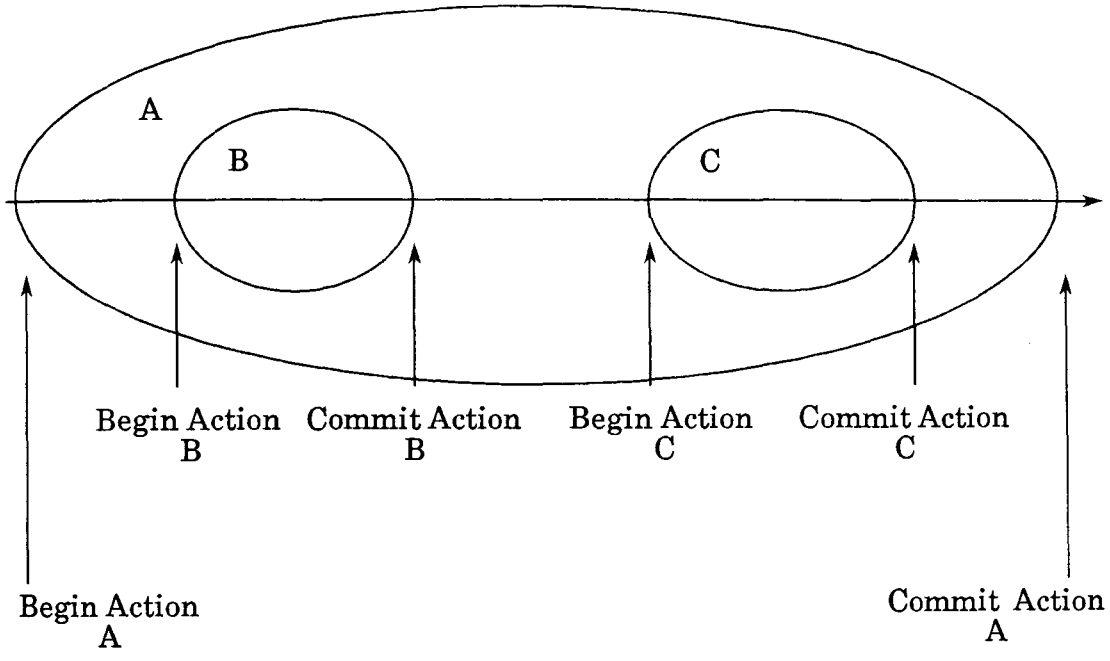

Figure 3-3: Sequential nested atomic actions

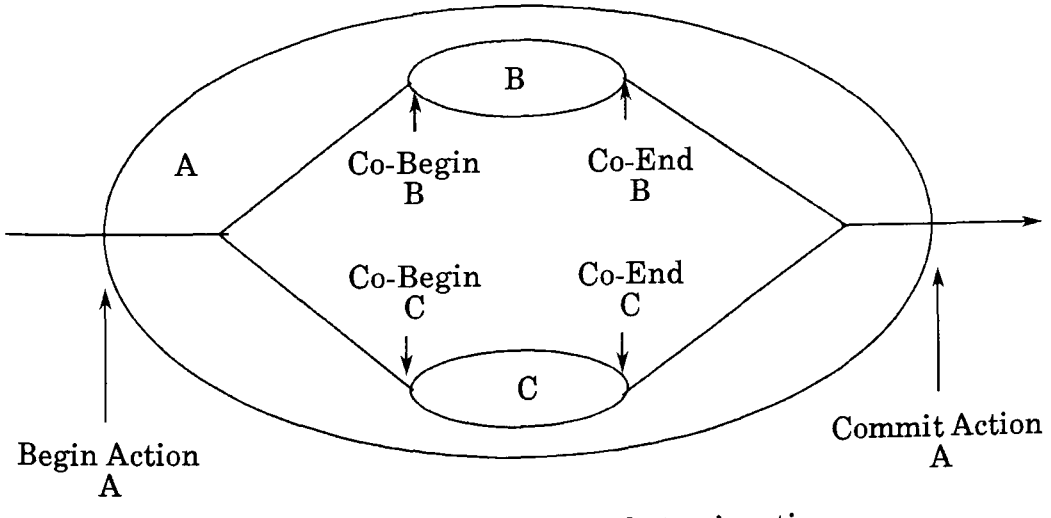

Figure 3-4: Concurrent nested atomic actions

an atomic action, any such nesting is proper in the sense that the executions of the nested actions B and C are always totally encompassed by the execution of the enclosing action A.

Such nested actions (or sub-actions as they are sometimes called) behave precisely like top-level actions. That is, they may fail independently of one another and are synchronised in the same way top-level actions would be (with minor provisos that will be explained shortly). However, stable storage is usually only affected when the top-level action commits. The reason for this lies in the fact that even if B and C commit there is always the possibility that A might abort, requiring that the effect on the system is as if B and C had never executed at all. If the commitment of the nested actions updated stable storage, these updates would then have to be undone. Typically, since updating stable storage is an expensive operation, the effect of the commitment of a nested action is only visible in the volatile version of an object. The stable version is only updated when the top-level action commits.

Nesting actions in this fashion has several advantages. Firstly, the use of concurrent actions can exploit the potential parallelism available in the system; thus top-level actions may execute faster than if they had been structured using the sequential approach. However, the effects of the parallelism may not be as great as might be expected depending upon the objects manipulated by the sub-actions. If all of the sub-actions manipulate the same objects then the concurrency controller may force a strict serial execution of the sub-actions since sub-actions are serialisable in precisely the same way as other actions. The nett effect is that the overall execution time is greater than if a sequential approach had been used due to the concurrency control overhead.

Secondly, and perhaps more importantly, nesting provides a means of isolation. That is, simply because one of the sub-actions aborts it does not mean that the top-level action must also abort. Rather, the failure is isolated to the (sub-action) tree rooted at the failed action. Thus if sub-action *B* (in Figure 3-4) aborts for some reason, action *A* is free to start another sub-action to do the work in place of the failed sub-action *B*.

This isolation property provides a kind of *firewall* to protect the top-level action from failures that would otherwise require that the entire action be aborted. In addition, using nested actions in this fashion allows the implementation of fault-tolerance based upon *recovery blocks* [*Horning et al. 74*] or N-Version programming [*Aviziennis and Chen 77*].

A third advantage of nesting is that new actions can now be composed out of existing, formerly top-level actions, since the old top-level actions simply become sub-actions of the new, more pervasive top-level action.

Since the structure that results from the use of nested actions conforms to that of a hierarchy, standard tree terminology combined with family relationships will be used to describe atomic action relationships. Thus a top-level action is the *root* of the action hierarchy. Similarly, actions having sub-actions are referred to as *parents*, while the sub-actions themselves are called *children*. Additional relationships such as *ancestor* and *descendant* have an equally obvious meaning.

3.5 Concurrency and Atomic Actions

The previous chapter treated concurrency control as a topic in its own right. This section shows how those techniques can be used to provide the important concurrency atomicity property of atomic actions. In general, this integration is a straight forward operation.

Recall that when an atomic action is started it is assigned some form of unique identifier that is usually termed the *atomic-action-id* or the *transaction-id.* The system uses this identifier throughout the lifetime of the action to track the effects of the action. If this unique identifier is such that it can be used as a timestamp (say the identifier is generated directly from the system clock in a fashion similar to that outlined in chapter two) then any of the timestamp-based approaches to concurrency control are available to provide the property of concurrency atomicity in an obvious manner - all that is required is that prior to attempting some operation upon an object, the concurrency controller is called to make sure timestamp ordering is being maintained.

Lock-based approaches to concurrency control are also possible since it is easy to arrange that prior to executing an operation the action attempts to set an appropriate lock. In fact, locking is used by the majority of actual implementations of atomic actions to provide the concurrency atomicity property with by far the most dominant method being *strict two-phase locking.* Strict two-phase locking modifies the basic two-phase requirement so that the shrinking phase is seemingly instantaneous at the end of the program. When used with atomic actions the acquisition of locks is incremental as operations are performed, while the release of locks occurs only when the action commits or aborts. The release of locks is instantaneous in order to avoid potential *cascade aborts.* Cascade aborts can occur in the following manner. Assume some action is using ordinary two-phase locking and is gradually releasing its locks during its shrinking phase. These locks can then be acquired by other actions and the objects they protect manipulated. However, if the original action now aborts it needs to restore the states of the objects it manipulated, but which may now be being used by other actions. Thus these other actions will have to be aborted also, and so on. This so-called *domino-effect* [*Randell 75*] is usually considered

undesirable and can be avoided by only releasing locks when the top-level action terminates.

Having noticed that early release is usually undesirable, the Profemo system [*Nett et al. 85*], does allow just such an approach, but uses specialised hardware to track the resulting dependencies between actions. In addition, Shrivastava [*Shrivastava 82*] has investigated a system model that tracks dependencies between actions and associates levels of confidence to results consumed as a consequence of early release of locks.

3.6 Effects of Nesting

While being highly desirable, the nesting of actions has some implications for concurrency atomicity. These sub-sections describe the required modifications to some standard concurrency control techniques to handle action nesting. The equivalent modifications required for failure atomicity are beyond the scope of this thesis.

3.6.1 Locking

When a non-nested action was committed or aborted the concurrency controller could discard any locks that it was holding on behalf of that action. The possibility that the action might be a nested one means that this is no longer true for the following reason. In two-phase locking locks cannot be released when a child action commits because the concurrency controller might then allow some other action to acquire the locks, thus breaking the concurrency atomicity property for the parent action. What is required is a means by which the parent action can *inherit* the locks acquired by its children so that it maintains control over all objects manipulated under control of itself and all of its children.

This extension to two-phase locking was made by Moss [*Moss 81*] and has become the standard way of implementing two-phase locking in a nested action environment. The scheme is as follows. A distinction is made between *holding* a lock and merely *retaining* it. When a lock is held the (sub-)action can manipulate the object in the normal way. When a child action commits, its parent action inherits and *retains* all of the locks held or retained by its child. Lock retention ensures that other actions outside of the scope of the top-level action cannot acquire the lock, but inferior child actions can. Should a child action be aborted all of its locks whether held or retained are released. Moss's locking rules are thus:

- An action may *hold* a lock in *write* mode if no other action *holds* the lock (in any mode) and all *retainers* of the lock are ancestors of the requesting action.

- An action may *hold* a lock in *read* mode if no other action *holds* the lock in *write* mode, and all *retainers* of *write* locks are ancestors of the requesting action.

- When an action aborts, all of its locks (*held* and *retained*, of all modes) are simply discarded. If any of its ancestors hold or retain the same lock, they continue to do so, in the same mode as before the abort.

- When an action commits, all of its locks (*held* and *retained*, of all modes) are inherited by its parent (if any). This means that the parent *retains* each of the locks (in the same mode as the child *held* or *retained* them).

Furthermore, lock modes are ordered, since some merging may be necessary if a parent inherited a lock from one of its children in a different mode to that which it was already retaining it in. For example, an action may have been *retaining* a lock in read mode. This would allow one of its children to acquire and *hold* the lock in write mode, so that when the child committed the parent would

inherit this lock. The parent must then retain the lock in the stronger of the two modes (in this case write).

3.6.2 Timestamping

Timestamp based approaches are more difficult to adapt to an environment supporting nested atomic actions. The major problem that arises is ensuring that the timestamp order is maintained even for the nested atomic actions. One approach to this is to allocate non-overlapping timestamp *ranges* to atomic actions and ensure that all nested actions draw their timestamp ranges from the timestamp range allocated to their parent. This technique ensures that all atomic actions are correctly serialised. One design that uses this approach was undertaken by Reed [*Reed 78, Reed 83*] and is based upon multi-version timestamping. The scheme is novel in that it took an integrated approach to the problems of action naming and synchronisation. Reed's scheme was later used in a simplified form as a basis for the SWALLOW project at MIT (see section 3.7.4 of this chapter for further details).

3.7 Examples of Systems Supporting Atomic Actions

This section describes some systems that support atomic actions in one form or another. While by no means exhaustive, the systems have been chosen to illustrate the fact that the various concurrency control techniques of the previous chapter have been used to implement concurrency atomicity in practice. The examples are drawn from distributed databases and distributed operating systems only; the description of some object-based systems is postponed until the next chapter.

3.7.1 R*

R* [*Lindsay et al. 84, Mohan et al. 86*] is an experimental distributed database system developed at the IBM research laboratory at Almaden. It supports only single level transactions and uses strict two-phase locking as its concurrency control technique. Each site in the distributed system runs the R* database manager and clients only ever communicate with their local manager. Requests for remote service are handled entirely between the R* managers themselves. Thus a request from a client is first presented to the local manager, who will forward it to some remote manager if required. Results destined for a client are likewise transmitted via the local manager. Such communication is made over a virtual circuit established on behalf of the client transaction when the first remote service request for a site is processed. Transaction identifiers are globally unique and are transmitted only when a connection between sites is first initiated (and the virtual circuit is established).

Since R* does not support nested transactions it uses an alternative, simpler, approach based upon the establishment of *save points* that act as recovery points. Should recovery become necessary a transaction is only recovered back to the last established save point, not all the way back to its start. The root process acts as a co-ordinator should this recovery be necessary by instructing all of the participants to recover to their save points.

Unlike most other database systems R* does not contain a separate lock manager process. Instead all lock related information is maintained in shared storage and is accessible to all of the processes at a site. The lock access code is executed directly by the processes accessing the database. Although generally obeying two-phase locking some locks are actually released before all other locks have been acquired for performance reasons. Allowing this requires that should the transaction abort these locks must be re-acquired. To avoid potential

deadlock problems only a single transaction is allowed to attempt lock re-acquisition at once. Deadlock detection is based on constructing a global wait-for graph. Each site maintains its own local wait-for graph and may initiate deadlock detection at any time.

3.7.2 Locus

Locus is a distributed operating system developed at UCLA [*Walker et al. 83, Walker 85*]. It has provided a testbed for several different implementations of atomic actions, some nested, others not. This section first describes the basic capabilities of Locus before considering how atomic actions have been implemented upon this base.

Locus is a UNIX compatible, transparent distributed system. It appears to the user as a single UNIX system despite the fact that it is executing on several nodes. The file system appears as a single tree structured hierarchy that spans all the nodes. Filenames in Locus are location independent, thus it is usually not possible to determine the location of a file from its name. Files may also be replicated to varying degrees for availability purposes. The file system itself is also somewhat more robust than traditional UNIX systems and uses a shadow paging technique [*Lorie 77*] coupled with commit and abort primitives to ensure that changes to files are handled atomically. In Locus only the operations performed upon files are recoverable.

Each Locus site is a full-function node executing the Locus kernel, though file system activity can involve more than one site. Locus systems define three logical sites:

- *Using Site.* This site issues the request to open a file and is the source of all file manipulation requests.

- *Storage Site.* This is the site at which a copy of the file resides. If the file is replicated it will have several storage sites, only one of which will be selected to supply pages of the file to the using site.

- *Current Synchronisation Site.* This is the site that enforces global access synchronisation to a particular file. The CSS stores information on which sites a given file is stored at, together with an indication as to which is the current version of the file. It may or may not actually store the file itself.

This partitioning of sites is purely logical and any site can be any or all of the above. The system is, however, its most efficient when all three functions are performed at a single site since there is no network communications overhead.

When a file is opened the current synchronisation site is interrogated to determine synchronisation policy and to determine a storage site for the file. The CSS is also responsible for maintaining a structure known as a *version vector* for replicated files. This structure enables the CSS to determine which Locus site currently stores the most up to date version of a replicated file.

Once a storage site has been selected communication is only between it and the using site while the file is manipulated. The CSS becomes involved once more when the file is finally closed.

Atomic Actions in Locus

The first full implementation of atomic actions in Locus included support for nested actions and was undertaken by Mueller [*Mueller 83, Mueller et al. 83*], based on an earlier simpler implementation by Moore [*Moore 82*]. The atomic action interface was simple: only a single system call was provided that started a new process executing as an action. The caller was blocked until the action thus created terminated. The created process was allowed to create other *member*

processes that where linked to the same action, any of which could start a sub-action using the same interface.

Concurrency control was via strict two-phase locking obeying Moss's nested locking rules. The Locus rules regarding the selection of a CSS and SS were modified such that actions interacted with a *transaction synchronisation site* (TSS), which played both roles. Similarly all I/O operations on the files were tagged with the transaction identifier of the action making the call.

The TSS maintained information using a *tlock* structure for a file. This structure contained information on both lock holders and retainers together with recovery information in the form of a file *version stack*. Versions of files where kept incrementally so that only those pages that had been modified by an action where noted.

Experience with the implementation described above caused a re-implementation to occur for several reasons. Firstly, the process structure was deemed to be too heavyweight. Secondly, maintaining version stacks and inter-transaction synchronisation proved to be too expensive, and thirdly, synchronisation was done only at the file level.

This second implementation [*Weinstein et al. 85*] attempted to overcome these difficulties at several levels. As a means of increasing concurrency, record level locking was introduced, allowing users to lock particular parts of a file rather than the entire file. Lock requests could be either made explicitly via a system call or implicitly when parts of the file were actually accessed.

File modification was not restricted to being performed as part of an atomic action, arbitrary processes could also do so. To cope with this extensions were made to the commit and abort mechanisms of the basic system to ensure that if a

file was modified both as part of an atomic action and by an ordinary process then inconsistencies did not arise.

The system still followed two-phase locking for actions, but not for ordinary processes whose locks could be released at any time. In addition, two methods by which serialisability could be avoided were provided. The first method was by the provision of special locks which did not have to obey the two-phase rule. The second method relied on the fact that locks acquired before an action was started were not converted to action type locks when the action did start. These locks behaved as if owned by ordinary processes.

Finally, actions were started and ended by explicit system calls and applied to the calling process. Furthermore, such actions could not be nested as they had been in the previous implementation.

Although admirable attempts, neither of these two implementations of atomic actions described in the preceding paragraphs can really be called a success. The first, which provided a full implementation of the nested action model, proved to be too expensive for general use. In addition, the user interface to it was unnatural and did not fit well with traditional UNIX interface. The second implementation remedied some of these problems but lost the flexibility that the nested model provided by reverting to a simple single level approach.

Probably the major flaw with both approaches arose from the underlying system itself. Locus was designed to be BSD UNIX compatible, with all that that entailed. In particular, the semantics of file system operation and the nature of processes did not harmonise well with the atomic action philosophy.

The arguments behind the original design effort was that by placing atomic action support in the kernel, it need be implemented only once, and could thus be made more efficient and relieve applications of the necessity of implementing

such support themselves. Unfortunately the resulting generality that was required to support various applications just did not integrate well with the UNIX model upon which Locus was based.

3.7.3 Amoeba

Another distributed operating system, Amoeba [*Tanenbaum and Mullender 81, Tanenbaum and Renesse 85*] is novel in that one of the distributed file services that are available uses an optimistic concurrency control technique combined with standard two-phase locking in a multi-version environment [*Mullender and Tanenbaum 85*].

The choice of which technique to use is based upon the amount of data to be accessed and hence the likelihood of conflict. Small updates (one file) use the optimistic approach; larger updates (several files) use locking.

The Amoeba file service makes use of immutable versions of files. When opened for writing, a new version of the file is created which initially behaves as a copy of the original. The new version becomes available when a commit operation is performed on the file. Files are structured as a tree of pages (although the page size is not fixed and is only limited to a maximum of 32k bytes) which may be shared by several versions. Thus each version is in some sense like a difference file [*Severance and Lohman 76*].

The optimistic concurrency control used is based upon the serial validation approach of Kung and Robinson [*Kung and Robinson 81*]. When an attempt is made to commit a version of a file a check is made to see if the version of the file this new version is based upon is the current version. If it is then the commit succeeds and the new version is installed as the current version of the file.

If the new version is not based upon the current version but on some older version this implies that at least one newer version of the file exists and a check must be made to see whether the changes made to the file to create the version that is being validated can be reconciled with those of the other newer versions of the file. Thus the page trees of each version are descended in parallel to determine if the two versions are serialisable. This check proceeds to check all newer versions of the file until it ascertains whether the validating version can be made the current version or not.

While this is an elegant scheme it is a sobering thought to note that this particular file service does not receive much use in the current implementation of the system. Basically, the optimistic file service is considered far too slow in comparison to some of the other file services that are also available in Amoeba.

3.7.4 Swallow

Swallow [*Svobodova 80, Svobodova 81, Arens 81*] was an attempt to use Reed's ideas on multi-version timestamping to design a reliable object repository. It simplified his model by not allowing gaps in the version history of an object. Thus when a new version was created, the validity interval of the preceding version was extended to immediately prior to the start time of the new version.

Swallow was simply a data storage system originally intended to be implemented upon write-once optical discs (theoretically an ideal medium since each version of an object was immutable). The management of this storage proved to be particularly complex since it could potentially grow forever, necessitating a distinction between *Online Version Storage* (that part currently available) and *Offline Version Storage.* Most of the problems arose due to the need the ensure that the latest version of any object was always online; thus

objects that had been inactive for a length of time frequently had to be copied to ensure their availability.

Actions are named and synchronised in SWALLOW using a concept called *pseudotime*. Pseudotime is a global, temporal coordinate system imposed upon a distributed computation such that all pseudotimes form part of a totally ordered set. Pseudotimes act as timestamps but are only loosely connected with real time. Associated with each atomic action is a pseudotime generator called a *pseudo-temporal environment*. Each such pseudo-temporal environment is effectively a non-overlapping subrange of all of the possible pseudotimes, thus all of the steps of one atomic action will either precede or follow all of the steps of another atomic action in psuedotime order. In order to handle atomic action nesting, each sub-action receives a non-overlapping subrange of its parent's pseudo-temporal environment as its own. Attempts to read or write objects require a pseudotime generated from the environment of the action. This pseudotime selected which particular version of the object could be manipulated by the action (since, in multi-version timestamp ordering, objects have versions that are valid over particular ranges of pseudotime).

When a new object is created (objects are immutable by virtue of the version scheme), it is only a tentative version known as a *token*. The set of all tokens created by an action forms a *possibility*. When an action is started a new possibility is created to which tokens are added as the action executes. Committing the action also commits the possibility and thus installs all of the tokens as proper versions of the objects in question.

Tokens are only visible to the action that created them, but not visible outside that action until it commits. In order that child actions might see each others tokens as well as those of their parents the notion of a *dependent possibility* is available.

3.7.5 Felix

Another file server, Felix [*Fridrich and Older 81*], is novel because it supported multi-file commit as one of its basic operations. Felix also allowed pre-declaration so that actions would never be aborted (a conservative two-phase locking approach). In addition, Felix maintained two versions of files (like two-version two-phase locking) using notions of *copy*, *original*, and *exclusive* types of access (in both read and write modes). Copy access provided an means of accessing the most recent version of a file but any changes made to the file were uncommitable. Original type access was the normal mode but also allowed copy access. Exclusive access provided the traditional exclusivity.

3.8 Summary

In this chapter the relationship between concurrency control techniques and the concurrency control requirements of atomic actions has been examined. As has been shown many of the concurrency control techniques described in chapter two have been attempted in one or more actual systems. However, two-phase locking has proved to be the dominant choice, particularly in commercially available systems, while other techniques have generally only appeared in research projects - often simply to show that such techniques could indeed be used and would work as envisaged.

The dominance of two-phase locking can probably be attributed to several factors. Firstly, its relative simplicity and intuitive correctness. Secondly, its wide applicability and good performance under many different situations. Thirdly, inertia is also at work; until some of the other concurrency control techniques have demonstrated any advantages they might possess, why change? Lack of commercial pressure is also a factor. Databases predominate in the commercial world and are still typically only providing single level transactions,

despite the apparent flexibility, power and elegance of the nested transaction approach.

Similarly, attempts to provide support for atomic actions within operating systems have proved to be of dubious value, often because of overkill - operating systems do not generally need the full generality of atomic actions as a rule, and supporting them becomes too restrictive and/or detrimental to performance.

Despite arguments that system level support for atomic actions is better than each application providing the support, the generality that such a system-based implementation must provide often makes using atomic actions unnatural to applications (for example, consider the Locus implementations of atomic actions) as they try to support all possible applications with the nett result that none is really supported adequately. In particular, in order to gain high performance, it is highly likely that the basic concurrency control and/or recovery mechanisms may need to be overridden by clients in order to specialise the system's level of support to one more appropriate to the needs of the application. In essence, these findings agree with those of Stonebraker *et al.* [*Stonebraker et al. 85*] who attempted to make the INGRES database system use basic transaction facilities available in the operating system of a PR1ME computer but discovered that substantial changes to both INGRES and the operating system were required before an acceptable level of performance would ensue.

Chapter 4
Object-Oriented Systems and Concurrency Control

Previous chapters have considered the provision of concurrency control and the support for atomic actions to be in some sense attributes of the system as a whole. That is, it has been assumed that the concurrency controller and the atomic action support system are system based. This chapter will modify this view significantly.

In chapter one it was postulated that building programs using the object-oriented paradigm was a profitable approach to adopt. Therefore, this chapter follows those rules and adopts the approach henceforth. The adoption of this approach leads to the interesting notion that since objects are considered to be encapsulated, then individual objects should to be responsible for their own concurrency control and recovery.

This latter proposition is the approach adopted in this chapter. In it, the notions of what constitutes object-oriented programming are first refined, followed by a concentration on the particular property of object-oriented programming languages that is useful for providing concurrency control to individual objects; that of *type-inheritance*. The chapter then goes on to show how a user-defined object can be subject to concurrency control in a simple manner by designing a basic concurrency control type that user-defined types can inherit and make use of (in particular, the design is of a concurrency control type which manages locks and which follows the two-phase locking technique). This ability for a type to inherit concurrency control capabilities is complemented by the ability for a type to also inherit recovery capabilities, however, this latter part is

beyond the scope of this thesis. For a complete description of the design of the support for these recovery capabilities, see [*Dixon 88*].

The use of inheritance as a means of providing concurrency control has a number of advantages. Firstly, it allows experimentation with different concurrency control techniques to be undertaken in a relatively simple and straightforward way (for example, chapter six examines the possible implementation of some other types of concurrency controller within the same basic object-oriented framework), since the capabilities are not tied into any particular system. Secondly, there is no need to design and implement either a new language and run-time system, nor a new operating system kernel, instead the inheritance based approach is applicable to any object-oriented programming language. This is contrasted with the approach taken by several other research efforts being undertaken in the same area including Clouds [*Dasgupta et al. 85*], Argus [*Liskov and Scheifler 83*], TABS [*Spector et al. 85a*], Camelot [*Spector 87*], Avalon/C++ [*Herlihy and Wing 87*], and ISIS [*Birman 86*].

4.1 Object-Oriented Programming

Object-oriented programming is a style of programming that differs from conventional programming styles by concentrating upon modeling entities from the real-world as logical objects, and the interactions between real-world entities as communication between such objects.

An object is an *instance* of some *type* or *class* (it will be assumed that the words *class* and *type* are freely interchangeable from this point on). Each individual object consists of some data structure (its *instance variables*) and a set of operations (its *methods*) that determine the external behaviour of the object. The operations provided by an object have access to the instance variables and can thus modify the state of the object. Furthermore, the type of an object determines precisely what operations may be applied to it. An object-oriented

program then consists of a sequence of operations applied to a particular set of objects.

The relationship among types is a relatively natural model of what happens in the real world. For example, one thing is often regarded as being like another except for certain differences. Thus a lion is like a domestic cat only it is larger and more ferocious. This relationship is expressed in object-oriented programming languages through the *inheritance* mechanism.

4.2 Type Inheritance

Having noted that in reality some objects are like other objects, a method of expressing this relationship is needed. This is accomplished by means of a *type hierarchy*. Type hierarchies arise due to the property of *sub-typing* in object-oriented languages by which one type is allowed to be a sub-type of another. For example, given some type A, a new type B can be created such that B is a sub-type of A. Certain terminology is associated with this behaviour. Given the type structure defined here then B is a *sub-type* of A, and conversely, A is the *super-type* of B. Alternatively, using the terminology of C++ [*Stroustrup 86*], then A is called a *base* type, and B is called a *derived* type.

Deriving new types from existing types has several implications. Firstly, it is permissible that wherever an instance of the base type (A) is expected (for example, as a parameter to some operation), then an instance of the derived type (B) may be supplied in its place. Secondly, (and as a consequence of this first point), the attributes of the base type must be *inherited* by the new derived type. Such attribute inheritance ensures that instances of type B are capable of behaving like instances of type A should the need arise. It is this latter property of inheritance that will be made use of later in this chapter to provide instances of

user-defined types (objects) with the ability to perform concurrency control operations upon themselves.

The situation described above illustrates *simple* type inheritance, where the new type inherits from only a *single* parent type. More complicated arrangements are possible that allows a derived type to have more than one parent. Such a situation is termed *multiple inheritance.* The two forms are shown diagrammatically in Figure 4-1, where Figure 4-1(a) illustrates simple sub-typing, and Figure 4-1(b) illustrates multiple inheritance.

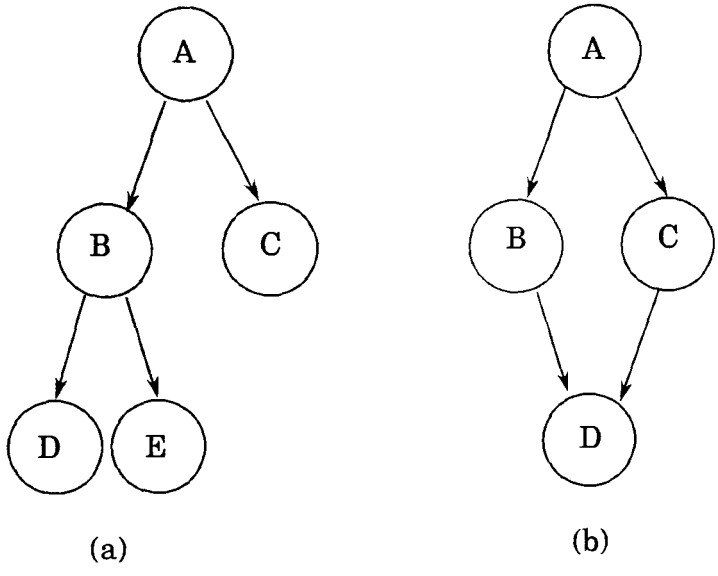

Figure 4-1: Simple and multiple inheritance

Inheritance is a very useful property that allows new types to share attributes of their parent type(s). The question that arises, though, is precisely what is inherited from the parent type: only the interface description, operation code, instance variables, or some combination of all of these? Furthermore, how are these inherited attributes viewed, by both the type doing the inheriting, and by any new types further down the hierarchy?

The fact that instances of a derived type should behave as instances of their base type in certain circumstances requires that the interface to the derived type must at least include the interface to the base type, and so at least the interface to the base type must be inherited by the derived type.

Another problem raised by the use of inheritance is the access the new type has to the attributes of the base type. One approach is to regard inheritance simply like any other form of access, so that the derived type has no special privileges and can only use the public interface to the base type. Adopting this approach often leads to making more attributes of the base type visible in the interface than strictly necessary simply because it is envisaged that they might be useful in designing future derived types. Alternatively, the act of deriving a new type can be considered different to normal usage of the base type, so that the implementor of a derived type should have additional privileges over and above those provided by the usual public interface. For example, direct access to the instance variables of the base type might be allowed or access to the private operations of the base type could be permitted.

One of the important properties of object-oriented languages, that they share with languages that support data abstraction, is that of *encapsulation*. Encapsulation (also known as *data hiding*) ensures that the internals of a type are not visible outside the type boundary. Thus the type provides a *black box* which only performs those operations defined by its interface. This hiding is important in that it allows the representation of the object, and even the way the interface is implemented, to be changed freely, providing that the actual interface to the type is not changed in any way.

Inheritance can compromise this encapsulation, since if the instance variables of a base type are directly visible to the derived type then a change in the way the base type is implemented could ripple throughout the entire hierarchy, requiring changes in all of the derived types of the changed type.

For example, consider a type that represents a matrix. Such a type will need instance variables that represent the matrix bounds together with some storage for the actual matrix elements (say a simple two-dimensional array). If these variables are visible (that is, they can be accessed directly) then any user of the matrix type will be able to view these variables. If at some time in the future the internal representation of the type was changed totally (to use a list of elements rather than an array because the matrix was sparse), then all users of the matrix type may be affected. By allowing the instance variables to be visible the implementor of the type has lost the ability to arbitrarily change the representation of the type without informing all users of the type of the change. This argument, also noted by Snyder [*Snyder 86*] implies that access to inherited instance variables should only be provided through inherited operations of the base type (so-called *access* operations).

The visibility of any inherited operations can also be problematical. Frequently the new derived type will change or refine the semantics of the operations it inherits from its parent(s) to make them more applicable to itself. In addition the new type may add new operations of its own or possibly restrict the use of others. While this permits specialisation, the degree of control over inherited operations varies from language to language. For example, in Smalltalk-80 [*Goldberg and Robson 83*], inherited operations can be refined in the derived class but they cannot be excluded from the interface except by refining the operations either to do nothing or to return an error. A similar approach is adopted by Objective-C [*Cox 86*]. This approach has the undesirable characteristic that as the hierarchy becomes deeper the interfaces to objects

potentially become more and more complex and cluttered with operations inherited from all of their ancestors.

Such unrestricted inheritance can also compromise encapsulation, since if a base type provides some operation to access its instance variables for a derived type's use, then that operation may also form part of the public interface to the derived type so that anyone may use it. For example, the implementor of the matrix type described earlier may have provided some operation that allowed the derived type access to information that would normally be kept private. Once defined, this operation may be available not only to all of the derived types of the matrix type but also to all the users of any of those derived types. Solving this problem requires recognising that the base type will have two classes of users: implementors of derived types and general users, and so each class of user should have a different abstract view of the base type. This distinction is made by languages such as C++, which enables certain attributes of a type to be declared *protected.* That is they can only be used by derived types of the type providing the protected attribute, thus ensuring they do not become publicly available unless one of the derived types explicitly makes them so.

In C++ and Trellis/Owl [*Schaffert et al. 86*], operations and instance variables may also be declared to be either *public* or *private.* Private operations and instance variables are only accessible to other operations of the type and do not form part of the public interface. Furthermore, if a derived type *privately* inherits a base type, then all of the public operations and instance variables of the base type become private variables of the derived type. On the other hand, if a derived type *publically* inherits a base type then the public operations and variables of the base type become public attributes of the derived type also.

Notions of private inheritance have implications regarding precisely where a type may be used. Recall that earlier in this section it was stated that if B was a derived type of A, then whenever an A type object was expected it was permissible to supply a B type object instead. This is acceptable in the Smalltalk-80 model where all operations are inherited publicly, since all the operations available in the A type object are also supplied by the B type object. However, in languages such as C++ and Trellis/Owl the operation may have been removed from the public interface of the object in the derived type. Hence such languages perform strict compile time checking to ensure that this situation is not permitted to arise.

The ability to refine and specialise operations implies that some operation binding must be performed at run-time. For example, suppose a type implements the operation *Describe*, the purpose of which is to cause an instance of the type to describe itself in some fashion (for example, by printing an ascii description of itself on some output device). A derived type inheriting this operation is certain to refine it so as to describe instances of itself, not its parent, which would be the meaning otherwise. Under most circumstances the compiler can detect the type of the object and ensure that the correct version of *Describe* is invoked when required. However, the type rules make it possible to supply an instance of a derived type whenever a base type is required. This implies that the object cannot simply be treated as being of the base type, rather a lookup must be performed at run time to determine the actual type of object supplied so that the correct version of *Describe* is actually called. This run-time lookup is called *dynamic binding*.

As an example, consider a type that maintains a list of objects. Ideally, objects of arbitrary type should be able to be inserted and removed from such a list, otherwise it would be necessary to implement different types of lists for each type of object. This type of generic list is easily constructed if it is designed to manipulate entries of some basic type (call it *List_Entry*). Types that are to be

inserted into a list are thus declared to be derived types of this base type and thus can be inserted into a list with ease.

Given such a list a program may wish to print out descriptions of all objects in it. To do this the program simply selects each entry in the list (in some fashion) and invokes the *Describe* operation of that entry. Since the compiler cannot detect what type of object will be on the list, the determination of which particular implementation of the *Describe* operation to invoke has to be made at run-time.

4.2.1 Type Inheritance in C++

The language that will be used to develop all of the examples henceforth in this thesis is C++ [*Stroustrup 86*], a language developed from C [*Kernighan and Ritchie 78*]. C++ is a superset of C, incorporating facilities for data abstraction, type inheritance and operator overloading.

The abstraction and inheritance features are related to those of Simula-67 [*Birtwhistle et al. 73*] and are based upon the *class* concept. Classes in C++ can currently only inherit from a single base class, although a version of the language supporting multiple inheritance has been developed [*Stroustrup 87b*].

Classes are defined in the manner shown in Figure 4-2. In this example a new class *File* is created that is derived from a public (as indicated in the class header by the keyword *public*) base class *LockCC*. Using the terminology of the previous section, *File* is a sub-type of *LockCC*, and *LockCC* is the super-type of *File*. Instances of the class *File* will have two private instance variables (*current_pos* and *page_count*), a protected variable (*fd*), and a set of public operations (*open*, *read*, etc.).

```
class File: public LockCC
{
      int current_pos;   // private stuff
      int page_count;

protected:
      int fd;

public:
      File ();
      ~File ();

      int open (char*, mode);
      int read (char*, int);
      int write (char*, int);
};
```

Figure 4-2: An example C++ class

In order to guarantee correct initialisation of objects when they are created, a special operation termed a *constructor* is automatically called when an instance of the class is created. This operation is a public operation that has the same name as the class itself (in this case *File*). Despite being public, the constructor operation cannot be called directly. Its function is to perform a type-specific initialisation of the newly created object. A complementary operation (called a *destructor*) is likewise called whenever the object is destroyed. Its name is that of the class preceded by a '~', (in this case *~File*).

Any operation or variable of a class can be declared as either *public*, *protected* or *private*. Normally, class definitions are written as illustrated here, with the private attributes first, followed by the protected attributes, and finally the public attributes, although it is possible to intermix them. Since *File* is declared to be publicly inheriting *LockCC* then all of the public attributes of *LockCC* (variables and operations) are also considered to be public attributes of *File* also.

C++ is a strongly typed language with compile-time binding of operation names to the code that implements them. However, as was noted in the previous section, there are occasions where dynamic binding must be used otherwise

objects could not be treated as instances of their parent type and passed to operations that expected them to behave as instances of their parent type. In C++ this is handled by declaring such operations as *virtual*. As illustrated in Figure 4-3 the only distinction between a normal function and a virtual function is the

```
class Shape
{
      void Move ();
      ....
public:
      virtual void Draw ();
      ....
};
```

```
class Circle: public Shape
{

      ....
public:
      virtual void Draw ();
      ....
};
```

Figure 4-3: Virtual functions in C++

occurrence of the keyword *virtual* before the operation declaration. The occurrence of this keyword indicates to the compiler that it should generate code to cause a run-time binding of the code that implements the operation based upon the type of the object. In this example the operation *Draw* is defined in both the base class (*Shape*) and the derived class (*Circle*), such that a call to *Draw* must determine at run-time which particular implementation to invoke based upon the type of the object currently under consideration.

Similar situations can arise with the base class operation *Move*. Given that once an object has been moved it will probably need to be redrawn at its new position, then the code that implements the operation *Move* may possibly be coded to make a call to *Draw*. Since in this particular example *Circle* is inheriting the definition of *Move* unchanged from *Shape (Circle* does not define its own version) the same code will be executed for instances of either type, however, depending upon the type of shape being moved then the appropriate version of *Draw* must be invoked. Hence, dynamic lookup is still required.

4.3 Concurrency Control in Object-Oriented Systems

This section examines some of the existing systems that are claimed by their authors to be object-based or object-oriented, paying particular attention to the mechanisms they use to implement concurrency control. For each system, an attempt is made to determine how flexible the concurrency controller is, and to assess the ease by which new user-defined types encompassing concurrency control can be created.

4.3.1 Clouds

The first system under examination is *Clouds* [*Allchin 83, Dasgupta et al. 85, Kenley 86*]. Concurrency control in Clouds is based upon standard two-phase locking, extended to cover lock modes other than simple read and write. If required, locks may be released explicitly under programmer control. In addition, Clouds supports nested atomic actions, so the concurrency controller obeys a slightly modified version of the nested locking rules advocated by Moss [*Moss 81*]. Requests to lock objects may be made either implicitly (the compiler inserts code to automatically lock the object) or explicitly using Clouds system calls, depending upon how the particular operation for an object has been defined.

Cloud's objects consist of volatile and permanent data segments and a set of operations upon those data segments. All objects are uniquely named and sharable. Application programs and user-defined types are coded in the language *Aeolus* [*LeBlanc and Wilkes 85*]. As indicated above, Aeolus supports two types of interaction with the Clouds concurrency controller depending upon whether locking is being performed implicitly or explicitly. If implicit locking is being performed then operations have to be classified as either readers (signified by the presence of the keyword *examines* in the text of the implementation of the operation) or writers (signified by the keyword *modifies*). Once so identified the

compiler inserts appropriate calls to the Clouds kernel to set appropriate read or write locks on the object as part of the standard operation prologue.

More explicit control can be obtained by appropriate declaration and use of instances of the basic Aeolus *lock* type as part of the definition of a user-defined type. A lock type is used to declare variables which can be used to implement type-specific locking for a user-defined type. Lock type declarations include the specification of a compatibility list that is used to determine whether a lock of a given mode can be set or not. In addition, locks possess *values* that allow the programmer additional control over the compatibility of locks.

A simple lock declaration is illustrated in Figure 4-4. In this example a new

```
type file_lock is lock ( read : [read],
                         write : [] ) domain is string(20)
```

Figure 4-4: Clouds lock type

lock variable is created called *file_lock*. This lock has two modes, *read* and *write*, which obey the traditional rules concerning lock compatibility (that is multiple readers, but a single writer). This lock is further identified by a string, the need for which will be described shortly. Such a lock might be used as part of the implementation of a type that represented a traditional file system directory.

Locks thus declared may be set, tested, and released as part of the execution of an operation using the primitives *Setlock*, *Testlock*, and *Releaselock* respectively. *Setlock* sets a lock of the given mode on the named instance from the lock domain. Thus, if the call of Setlock illustrated below:

```
Setlock (file_lock, read, "myfile")
```

was made as part of the execution of an operation, then a read lock would be set upon the lock *file_lock* using the string *myfile*. By associating values with locks,

Clouds allows programmers to increase the level of concurrency an object supports. For example, if the following Setlock call:

```
Setlock (file_lock, write, "hisfile")
```

was made in addition to the earlier call shown above, then it would succeed, despite the apparent incompatible mode (reads conflict with write), due to the fact that the lock specifies a different value. Effectively, the values associated with locks provide the illusion that locks are being applied at a finer granularity than they actually are.

The *Testlock* operation on locks is provided to enable the programmer to determine if attempting to set a lock would block, prior to actually executing the *Setlock* call. Testing the value of a lock does not guarantee that the lock will remain free, since two concurrent actions could both test the lock, find it free, and attempt to set it. Depending upon the lock mode required and the compatibility between locks then both may succeed, or only one. The programmer must be aware of this possibility and use additional mutual exclusion primitives if the action must not block.

The Clouds scheme is interesting due to the way that locks are permitted to have values which give the illusion that locks are being applied at a finer granularity than they actually are. For example, the lock type illustrated in this section could have been used in the implementation of a filesystem directory type where it would have given the illusion that individual files were being locked, not the directory object itself. Even so, since the Cloud's kernel implements the concurrency controller, all objects are currently limited to using two-phase locking, and furthermore, all applications must be programmed in *Aeolus*. Finally, the system is not object-oriented (by the definition of chapter one) since it does not support inheritance, rather it is best described as object-based [*Wegner 87*].

4.3.2 Argus

Argus [*Liskov 84, Liskov 88, Weihl 84*] is a distributed programming language (and system) that supports nested atomic actions. It is derived from the programming language Clu [*Liskov et al. 79*]. In Argus, programs are structured as a collection of operations on *guardians* [*Liskov and Scheifler 83*]. Each guardian consists of a set of local data objects and processes for manipulating those objects; thus guardians are object managers. Objects within a guardian can only be manipulated by processes within that guardian. Each guardian provides a set of *handlers* (operations) which constitute the guardian's public interface. Handler calls are executed by a new process, with each call executing under the control of an atomic action.

In addition to the provision of some basic data types that are atomic (that is, recoverable and serialisable) such as integers and arrays, Argus also supports the construction of user-defined data types that are similarly both serialisable and recoverable. Concurrency control over the built-in atomic data types is via standard two-phase locking using traditional read and write locks, with inheritance of locks as defined by Moss. Locks on built in atomic types are automatically set and released by the system without any provision for programmer control. User-defined types are similarly restricted if they are implemented using only the basic atomic data types. In order to permit higher levels of concurrency than this built in locking strategy would normally allow, the programmer must build user-defined types using non-atomic types (types whose use is not constrained by locks) in conjunction with the basic, built-in atomic types. For example, a type that represented a queue could be constructed as a non-atomic array of atomic entries. Since the array itself is not constrained by serialisability, several independent atomic actions can modify the queue and insert and remove entries from it. This level of concurrency would not be possible

if the array was itself atomic since modification of the array would set a write lock on it automatically.

The major problems with Argus stem from the fact that the concurrency control is totally implicit and automatically invoked whenever any of the basic atomic types are manipulated. Thus, in order to increase concurrency, the programmer has to play potentially dangerous tricks by mixing atomic and non-atomic types. As with Clouds, Argus is best described as an object-based system, since it does not fulfill the definition set out in chapter one.

4.3.3 TABS

The TABS (TransAction Based System) project [*Spector et al. 85a*] at Carnegie-Mellon is in many ways similar to the Argus project at MIT. TABS provides *data servers* that encapsulate one or more data objects. These data servers are similar to Argus guardians in that they are essentially recoverable object managers.

TABS is built upon the Accent kernel [*Rashid and Robertson 81*] and the various components of the TABS system (such as the *Transaction* manager and the *Recovery* manager) communicate with one another by sending messages addressed to *ports*. In order to ease the burden of programming such message transfers a remote procedure call facility called *MatchMaker* [*Jones et al. 85*] is used. *Matchmaker* takes descriptions of procedure headers and outputs client and server stubs that manage the packing and unpacking of the data into messages and the appropriate dispatching of the correct procedure in the server.

Data servers use locking as their synchronisation mechanism using standard two-phase locking. Locking is explicit in that the data servers must explicitly call the TABS routine *LockObject,* supplying an object identifier and a mode, in order to set a lock. If the lock is not available the server is made to wait.

Like Clouds, TABS also has primitives to test if a lock is set. However, it also has a *ConditionallyLockObject* routine that locks the object if possible, or returns immediately otherwise. This avoids the need for the separate mutual exclusion necessary in Clouds.

There is no unlock facility in TABS. Objects are only unlocked when the action that locked them commits or aborts, following the standard nested locking rules.

4.3.4 Camelot

Given the experience of TABS, the designers of that system are now in the process of producing CAMELOT (CArnegie-MEllon Low Overhead Transaction facility) [*Spector 87, Spector et al. 87*]. In many ways the influence of TABS is apparent in the philosophy of Camelot - indeed the structure of a Camelot node bears a considerable resemblance to the structure of a TABS node. Thus Camelot also uses data servers that encapsulate objects. It is, however, built on top of the *Mach* operating system [*Jones and Rashid 86*], which is a BSD4.3 UNIX compatible system

Camelot provides support for two compatible types of concurrency control: standard two-phase locking and hybrid atomicity [*Weihl 84*]. Hybrid atomicity makes use of timestamps generated when atomic actions commit to provide more information about the serialisation order of atomic actions, and hence permit the concurrent execution of some operations that other concurrency control techniques might have serialised. Hybrid atomicity thus combines aspects of both static (timestamping) and dynamic (lock-based) concurrency controllers. The mixed locking and timestamping protocol briefly mentioned in chapter two (section 2.4.12) is one example of a hybrid atomic concurrency controller.

As with TABS the concurrency control (of either form) is explicit with locking being provided via a call to the routine *Camlib__Lock* which takes a lock name and a mode as parameters. Similarly there are routines to test and set a lock (*Camlib__TryLock*), and determine the status of a lock (*Camlib__LockStatus*). As with Clouds, however, Camelot has added an explicit unlock call (*Camlib__Unlock*) to enable locks to be released early.

Support for hybrid atomicity requires the use of timestamps in addition to locks and requires that objects explicitly take part in the process of action commitment. Camelot implements this by allowing servers to declare routines that will be called whenever they become involved in the commitment or abortion of an action.

Camelot is claimed to integrate the best features of several systems. Thus it uses the optimised commit protocols of R* [*Mohan et al. 86*], the nested transaction mechanism of Argus, and the virtual memory and recoverable storage mechanisms of TABS. The system is very flexible, permitting much tailoring of the implementation of object servers, however, the interface is complex and requires use of some unorthodox programming techniques. For example, recoverable objects are modified using a macro rather than the conventional programming language concept of assignment. Furthermore, clients are always aware of the client/server relationship that exists in the system with calls on the operations supported by a server being coded differently to other procedure calls.

4.3.5 Avalon

Not strictly a separate system in its own right, Avalon [*Herlihy and Wing 87*], is an attempt to provide programmers with a set of linguistic constructs designed to give explicit control over transaction-based processing of atomic objects.

The Avalon constructs are implemented as extensions to some host language such as C++, and are currently hosted upon the Camelot system. In many respects Avalon resembles Argus; the principal differences occurring in the way that user-defined atomic data types are implemented.

Within Avalon/C++ advantage is taken of the inheritance properties of the language, such that new atomic data types are created by deriving them from a system-defined type called *atomic*. This base type provides a monitor-like facility for mutual exclusion, and provides virtual functions for action commit and abort. It is the provision of these latter functions that allows Avalon objects to implement the property of hybrid atomicity [*Herlihy and Weihl 88*].

Currently, Avalon is the only other system (known to the author) that is making use of inheritance in any way, however, its base system (Camelot/Mach) provides it with many facilities for object recovery and concurrency control and therein lies the major problem. Many of the characteristics of the underlying system are visible to the programmer and considerable care must be taken to ensure that these characteristics are handled in the implementation of any user-defined types. For example, the programmer must be aware of the method by which the system implements recovery, otherwise it is easy to make the system inconsistent.

4.3.6 ISIS

The ISIS project from Cornell university [*Birman 86*] aims to produce fault-tolerant implementations of objects automatically from fault-intolerant program specifications. The resulting objects are then known as *resilient* (or more precisely *k-resilient*) objects.

ISIS replicates the code and data of each object at least $k+1$ times, while ensuring that the replicated program behaves exactly like a non-replicated program obeying the same specification. Resilient objects are represented at a set of sites by components that are capable of executing requests sent to them via remote procedure calls. Each request is handled as a separate atomic action.

Concurrency control in ISIS is explicit to the implementor of type since it is difficult to infer an efficient concurrency control algorithm without knowledge of the semantics of the operations of a type. Thus ISIS requires the provision of a single site concurrency control algorithm, which is transformed into a distributed one. ISIS basically supports two-phase locking but locks are classified into two distinct types.

- Nested two-phase locks. These obey the standard nested two-phase rules.
- Local two-phase locks. These obey standard two-phase rules but are always released at action commit or abort regardless of whether the action is nested or not.

ISIS locks can belong to one of four distinct modes: read, write, promotable read, and previous committed version read. The first two behave in the standard manner expected, the others are described in the following paragraphs.

Promotable read locks are designed to overcome the problems associated with lock conversion. In essence they are exclusive read locks that may be promoted to write locks. Since they are exclusive then only one action can hold such a lock and thus promotion of such a lock to a write lock will not cause deadlock which could otherwise occur.

Previous committed version locks are intended for actions that can be classified as read only. By allowing access to a previously committed version of an object both reads and updates can be allowed to proceed in parallel. This is essentially an implementation of the two-version two-phase locking strategy described in chapter two.

The ISIS system described above was implemented; but its designers were not happy with the level of concurrency the system supported, or the ease of creating resilient objects. They have now embarked upon ISIS-II, which has similar goals but is based upon the notion of Virtual Synchrony [*Birman and Joseph 87*], rather than serialisability as its correctness criterion. The designers feel that this technique is much better suited to building highly concurrent distributed applications. It remains to be seen whether their confidence will be justified in reality.

4.3.7 Some Conclusions

All of the systems described in the previous sections have been claimed by their designers to be either object-based or object-oriented. While the systems do indeed support the concept of an object as an encapsulated entity, it is interesting to note that all of the systems have adopted the approach of building either a new language or system, or possibly both to provide this concept. Only Avalon has attempted to use the capabilities of an existing object-oriented language and provide a simple means of permitting user-defined types to be serialisable and recoverable. In addition, all of the systems have chosen to use locking (in one

form or another) as the basic (and often unchangeable) concurrency control technique. Only ISIS in its latest incarnation has attempted to break this mould.

The remainder of this chapter will show how it is possible to avoid this commitment to a single concurrency control technique by providing a flexible framework for the implementor of a user-defined type to use. Furthermore, the technique used does not require a new language or system but can be applied to any object-oriented language.

4.4 Concurrency Control via Type Inheritance

This section describes a novel approach to providing individual objects with their own concurrency controller by making use of the property of type inheritance. In particular, the design and implementation of a concurrency controller based upon the common technique of two-phase locking is described. In many respects two-phase locking is an ideal concurrency control technique since it makes all decisions about whether to grant locks based upon purely local information. Thus, in a conventional distributed system this might be site-local information. Here this locality is taken to its logical extreme and concurrency control decisions are made using information purely local to the individual objects themselves.

The concurrency control type designed in the rest of this chapter is intended to support standard two-phase locking using only the lock modes of read or write which obey the traditional rules with respect to conflict. While this may seem highly restrictive, it is shown later in the chapter how simple modifications overcome these restrictions with ease, further demonstrating the flexibility of the type-inheritance based approach. Furthermore, chapter six, describes how more explicit type-specific locking can be implemented in an equally flexible manner.

There are, however, numerous issues that need to be resolved first. For instance, what is the interface to the concurrency control type as seen from user-defined types that are derived from it? In addition, is the provision of locking implicit in that the derived type need take no action, or is it explicit requiring the operations of the derived type to invoke appropriate operations of the concurrency control type directly? Then too there is the problem of how to represent the lock requests themselves. If the interface provided a call of the form:

```
SetLock (Mode);
```

what is the form of the *Mode* parameter?

The following sections attempt to answer these questions and come to some conclusions about the resulting design.

4.4.1 An Overview of the Concurrency Controller

In many respects the preferences for how the concurrency control type should be presented to the designer of a new type have already been betrayed in Figure 4-2.

The concurrency control type can be inherited by any user-defined type that wishes to make use of it. For the moment it will be assumed that there is only this single concurrency control type, and that all user-defined types that require concurrency control will make use of it. Thus the aim is to provide a base type - which will actually be called *LockCC* (standing for Lock-based Concurrency Controller) - from which all user-defined types should be derived. In effect this makes all user-defined types merely derived types of a basic concurrency controlled type. This concurrency control type is a *lock manager*. It permits locks to be set providing that the basic conflict rules would not be violated by doing so. The type is strictly a manager in that it does not create locks itself but merely

ensures that locks created by the user are set and released in accordance with the rules of two-phase locking.

Given this approach, some determination needs to be made as to what operations this concurrency control type should provide, such that user-defined types have as much flexibility as possible over the types of locking policy they follow.

In addition, as has been previously stated, the intention is not to modify a language or its compiler. This precludes the automatic, implicit approach adopted by Argus, or the compiler-based approach of Clouds for determining when locks should be set on objects, since it is not possible, in general, automatically to determine when a lock should be set, or more problematically, what particular type of lock should be set. Therefore an explicit approach has been adopted and the interface to the concurrency control type provides specific operations for the manipulation of locks.

Note, however, that the use of a concurrency controller is only explicit to the *implementor* of the type that is actually derived from the concurrency control type, not to the eventual user of this user-defined type. That is, when an operation upon an object is invoked, a lock will be set because the code implementing the operation explicitly sets a lock. However, as far as the invoker of the operation is concerned, the acquisition of the lock is simply a side-effect of the operation. Thus, as is illustrated in Figure 4-5, the implementor of the operation *open* has created a new lock object and passed that to the concurrency controller via the *setlock* routine. However, as far as the actual caller of the *open* operation is concerned, this concurrency control activity has occurred implicitly and is simply a side-effect of the execution of the *open* operation.

```
int File::open (char* fname, mode openmode)
{
      lockstatus openstatus;
      // First set an appropriate lock if possible
      openstatus = setlock (new Lock(openmode));
      if (openstatus == REFUSED)
            return ERROR;
      // now actually open the file and do any other housekeeping
      ...
}
```

Figure 4-5: Outline *open* operation for the *File* class

This explicit approach is not too bad a choice. As has been pointed out by others, increased levels of concurrency are possible by providing the type designer with explicit access to the concurrency controller for the type, and although at the moment it is assumed that only simple read and write type accesses will be made and standard conflict rules will be utilised, it will be shown in section 4.10 and later in chapter six that further use of inheritance provides the flexibility to adopt other approaches.

Bearing these points in mind, Figure 4-6 shows the skeletal declaration of

```
class LockCC
{
      Lock_List locks_held;                  // List of all currently held locks
      Semaphore* mutex;                      // For mutual exclusion purposes
      ...                                    // Other CC state as necessary

      virtual boolean lockconflict (Lock*);

public:
      LockCC ();                             // Initialise concurrency controller
      ~LockCC ();                            // Cleanup

      lockstatus setlock (Lock*);            // set lock on this object
      ...
}
```

Figure 4-6: The *LockCC* class

this basic concurrency controller type *LockCC*. For the moment it will be assumed that this base type only provides the ability to set locks on objects via the *setlock* operation. Furthermore, it will be assumed that the caller is executing

under the auspices of some atomic action, although description of how this is achieved is delayed until chapter five.

The basic operation of *setlock* can then be described as follows: *Setlock* is responsible for setting a lock upon the object derived from this base type. The type of lock required is determined by the parameter passed as part of the call. This parameter is of type *Lock* and contains sufficient information to allow the concurrency controller to determine if this particular lock can currently be set. Locks will be described more completely in section 4.5. *Setlock* returns a status to indicate the success or failure in granting the requested lock. Normally, when the lock cannot be granted due to conflict, the calling process is blocked and the call will only return when the lock has actually been granted. There are, however, instances when the call can return with an error status - this point will be considered further in section 4.8.

The boolean function *lockconflict* is used by *setlock* to determine whether any two locks conflict or not. It returns *true* if setting the lock would cause conflict, *false* otherwise. The routine is declared *virtual* to ensure that any type derived from *LockCC* could implement its own notion of conflict (this topic shall also be explored in section 4.6). Note, however, that the function is also *private*, thus ensuring that the only way it will be called is through one of the public functions of *LockCC*, in this case the public function *setlock*.

At this point in the design an operation to allow a lock to be released has deliberately not been included in the interface. For the moment it will be assumed that lock release is accomplished automatically in some fashion; later in this chapter (in section 4.10.3), and in chapter five, a method of achieving this effect will be described.

To be able to determine whether any particular *setlock* request can be honoured, the concurrency controller of an object maintains a list of lock objects that are currently being both held and retained (in order to obey Moss's nested locking rules). By scanning this list, the concurrency controller can decide if granting the request would cause conflict to occur. Two separate lists could have been used; one for the holders of locks on the object and one for the retainers of locks on the object, however, since it is assumed that the lock objects themselves can be interrogated as to their type, and both lists may need to be searched anyway when attempting to set a lock, the two types are kept on a single list.

Finally, since the concurrency control operations may be being executed by several atomic actions concurrently, a traditional semaphore *mutex* is provided to enable simple mutual exclusion during these operations.

4.5 Locks as Objects

One of the key characteristics of the systems described in section 4.3 was that, despite the fact that they were claimed by their designers to be object-oriented, none of them were consistent in this view. Thus a lock was often regarded as a primitive (and unchangeable) system type, as indeed was the interface to the concurrency controller.

This thesis wishes to take a different view of locks. That is, locks are regarded exactly like any other object in the system. Thus locks are objects (or more precisely locks are simply instances of a particular lock type).

This approach has several advantages. Firstly, locks can be created and manipulated in the same way as any other object in the system. Secondly, new language features or modifications to the run-time environment are not required to support them. Thirdly, the approach is very flexible, particularly if advantage

is taken of the basic inheritance properties of the language (this latter point will become clearer in section 4.10).

Figure 4-7 shows a skeletal declaration of one possible *Lock* type. Instances

```
class Lock
{
      lockstatus current_status;          // status, e.g. HELD
      modetype lockmode;                  // mode of lock, e.g. READ
      Uid owner;                          // identity of lock owner
      ...                                 // other private
                                          // variables and operations

public:
      Lock (modetype);                    // Lock object initialiser
      ~Lock ();

      modetype getlockmode ();            // Interrogation operations
      lockstatus getstatus ();
      Uid getowner ();
      ...
}
```

Figure 4-7: The *Lock* class

of this type can be declared whenever they are needed by the programmer, and it is instances of this type that are passed to the concurrency control type *LockCC* as the parameter to *setlock*.

This *Lock* type encapsulates as part of its private state all of the information that might need to be known about any particular lock instance. For example, it maintains information about the current mode of the lock (say READ), the current status of the lock (*held* or *retained* in accordance with Moss's locking rules), together with any other information that might be deemed necessary such as some notion about the owner of the lock (typically this will be the identifier of the atomic action under whose control the lock was set). Following the arguments made earlier in this chapter about encapsulation it also provides a set of operations to retrieve this internal information should it be required rather than make the information directly accessible.

Locks have a constructor function, which is used to initialise a new *Lock* object when it is first created. This constructor ensures that all of the instance variables have appropriate values for such newly created locks.

Any given instance of this *Lock* type can be in one of three states. It is initially *free* after it has been created. It becomes *held* after it has been successfully supplied to the *setlock* operation, and may then become *retained* if the atomic action that created it performs a nested commit. These states naturally conform to Moss's notions regarding held and retained locks. Once in a *held* or *retained* state a lock object will stay in one of those two states as appropriate until it is eventually destroyed.

In this design the mode of a lock object is considered to be immutable; that is, it cannot be changed once the lock object has been created. Thus, having declared a lock object to be a read lock, then the lock object is always a read lock. If a write lock is required, a new lock object with the appropriate mode must be created. The reasons behind this philosophy relate to the way locks are expected to be used. In database systems for example, the only reason to change the mode of a lock is due to the notion of *lock conversion*. In the system being described here such lock conversion is not allowed. The effect of lock conversion is, however, permissible and the manner in which it is achieved is described in section 4.10.1.

It might be argued that the mode of a *Lock* should not be determined by an instance variable at all, but rather, should be determined by the actual basic type of the lock. That is, use should be made of the sub-typing mechanism of the language to create new types of lock rather than maintain a single *Lock* type. If this approach was followed, then the system would need a *ReadLock* type, a *WriteLock* type, and so on, for as many different types of lock as were needed. Naturally, all of these lock types could be derived from the basic *Lock* type in the manner shown below:

```
class ReadLock: public Lock
```

While this scheme seems elegant, it is not without its problems. Recall that the concurrency control type, as one of its basic actions, is required to compare locks for conflict. While the mode of a lock remains as an instance variable this is easy to achieve, since the concurrency controller is effectively comparing *Lock* with *Lock*, a valid operation to attempt. However, if the mode is somehow encoded as part of the basic type, there are problems. For example, what does it mean to compare an instance of a *ReadLock* type with an instance of a *WriteLock* type? Given that the meaning could be expressed in the language, problems can arise later if further new lock types are introduced, since there must be some way of expressing how these new lock types compare with the old types. Furthermore, the old lock types must also be changed so that they know how they conflict with the new lock types. Thus, in order to avoid such complications, the mode of the lock is maintained simply as an instance variable of the basic *Lock* type.

In addition the owner of a lock is set when it is created to be the identifier of the creating action (recall that it was assumed that execution of the operation was proceeding under the auspices of some atomic action). Should a lock be set when the process is not executing as part of an atomic action then a fake identifier is created and the lock is flagged as being a non-action lock. Such information is naturally held in the instance variables of the *Lock* type.

4.6 Inside the Concurrency Controller

The two previous sections, have given a basic overview of the concurrency control type and the lock objects that it manipulates. This section, examines the concurrency control type in detail and gives a more precise definition of its interface and internals.

4.6.1 The Setlock Operation

As described earlier, *setlock* is the only operation of the concurrency controller that is publically visible. This operation is responsible for taking the user-provided lock object and performing a conflict check between it and all of the other lock objects that the concurrency controlled object is currently managing. A code skeleton for this operation is shown in Figure 4-8.

```
lockstatus LockCC::setlock (Lock* reqlock);
{
      boolean conflict = TRUE;              // assume there is conflict
      do
      {
            P(mutex);                       // grab semaphore
            if ((conflict = lockconflict(reqlock)))
            {
                  V(mutex);                 // conflict exists so...
                  sleep();                  // wait for a while
            }
      } while (conflict);                   // check repeatedly
      locks_held.insert(reqlock);           // add lock to list
      ...
      V(mutex);                             // release semaphore
      return (GRANTED);
}
```

Figure 4-8: The *setlock* operation

As illustrated here, *setlock* attempts to determine whether conflict exists by calling the *lockconflict* operation. If this operation returns the result *TRUE*, then conflict exists between the requested lock and (at least) one of the other locks currently set on the object. In this case, the semaphore is simply freed and the caller is made to sleep for some period of time. How this sleep is implemented is not important, in that it may be a busy wait, or a simple wait for a fixed interval, or any other acceptable means of blocking the operation. However, the concurrency controller does not assume that because the *sleep* call has returned that the conflict must now be resolved. In particular, if the conflict had been caused by two other locks conflicting, the release of one might have triggered the wake up, despite the fact that conflict still exists.

4.6.2 The Lockconflict Operation

In many respects *lockconflict* is the heart of the concurrency control type since it is this operation that determines whether or not there exists a conflict between the requested lock and all of the currently held locks.

This determination of conflict is done by comparing the mode of the requested lock object with the modes of all of the other lock objects currently being held upon the concurrency controlled object. A simple version of *lockconflict* that only considers locks that obey the traditional read and write conflict rule is given as Figure 4-9.

```
boolean LockCC::lockconflict (Lock* reqlock)
{
      Lock_Iterator next(locks_held);
      Lock* heldlock;

      while ((heldlock = next()) != Null)  // iterate over all locks
      {
            if (heldlock->getowner() != reqlock->getowner())
                  switch (reqlock->getlockmode())
                  {
                        case READ:
                              if (heldlock->getlockmode() == WRITE)
                                    return TRUE;
                              break;
                        case WRITE:
                              return TRUE;
                  }
      return FALSE;
}
```

Figure 4-9: The *lockconflict* operation

Since *lockconflict* must check whether a conflict exists between the requested lock and (possibly) all of the currently set locks, it is convenient to employ some mechanism that delivers each lock in turn to *lockconflict* for consideration. In this case, an instance of the class *Lock_Iterator* (called *next*) is employed for precisely this purpose. When created, the constructor for the *Lock_Iterator* class ensures that the first call to *next* will deliver the first lock from the list specified as the parameter to its constructor. Subsequent calls to

next deliver each succeeding list entry, until all have been delivered when a result of *Null* is returned. Having retrieved a lock instance, *lockconflict* determines whether the mode of the requested lock (passed as a parameter) conflicts with that of the lock it has just retrieved via *next*. This conflict check makes use of the public operations of the *Lock* objects themselves to determine each lock's mode and owner.

Since it is assumed that this basic version of the concurrency controller obeys the simple multiple reader, single writer, policy then if the requested lock mode was *write* then the existence of any other lock applied by a *different* action must cause conflict (recall that locks set by the same action cannot, in general, cause conflict with each other).

4.6.3 Some Disadvantages of this Design

As described above the design has certain disadvantages. The most notable one is that despite the fact that locks are objects and are thus encapsulated the *lockconflict* operation must still be able to take two such objects and compare them for conflict. This state of affairs means that if a new lock mode was added to the basic lock type (for example, by deriving a new type of lock from it) then appropriate modifications must also be made to the implementation of *lockconflict*.

This is possible, since *lockconflict* was deliberately declared *virtual* with precisely this point in mind. Thus the user-defined type (that is, the type actually derived from *LockCC*) can redefine the operation of *lockconflict* to take advantage of the new lock modes. Having done this, then whenever *setlock* called *lockconflict*, the run-time lookup performed would automatically ensure that the appropriate version of the operation was actually invoked.

This solution is, however, somewhat unattractive, since this could imply that *lockconflict* ends up being redefined in many types, possibly incorrectly. What is required is some way of allowing a standard version of *lockconflict* to determine whether conflict exists without having explicit knowledge of all of the possible different types of lock that might exist.

One possible way this could be handled is by representing the conflict information as some form of boolean matrix, such that the conflict check amounts to little more than indexing into this matrix using the requested mode and the held mode as indices. All that would be required then would be some way of informing the concurrency controller of the correct matrix to use, which could be handled as part of the constructor mechanism perhaps, or through provision of a *setmatrix* type of operation.

This approach is viable, but really needs compiler support to be implemented efficiently. In fact a similar scheme is adopted in Clouds [*Kenley 86*], where each *SetLock* call in Aeolus translates into a call on the Cloud's kernel with the additional parameter of a compiler computed lock compatibility clause deduced from the compatibility clause given when the lock was declared. This clause is simply a bitstring table which can be accessed very efficiently and thus conflict checks reduce to bit tests in the Cloud's kernel.

Problems with this approach can occur when type-specific locking is considered, since compatibilities are now on specific instances. For example, given some directory object, two write locks may be permitted providing they access different entries in the directory. Thus the compatibility clause cannot simply say that write is compatible with write, since this is only true if other conditions are also met (that is, different entries are being written).

Clouds avoids this problem by parameterising locks using user-supplied values. So for the above example, standard multiple reader, single writer compatibility clauses can still be specified, since write locks would be applied to different values (i.e. the particular names in the directory, rather than the directory itself).

Avalon/C++ [*Herlihy and Wing 87, Herlihy and Weihl 88*] employs a similar technique, except that it requires that the conflict table be built dynamically by the object constructor as part of the initialisation of the object. Obviously such an approach may have significant run-time overhead, particularly if the object has a complex compatibility matrix and many objects of that particular type are created.

At the moment the system being described in this chapter cannot use the Cloud's approach because the basic *Lock* type is not parameterisable in the sense that Cloud's locks are. Chapter six shows how such types of lock can be simply constructed from the basic *Lock* type using inheritance. Furthermore, given a wish to avoid the potential overhead of the Avalon approach, an alternative method must be found. One such approach is described in the following section.

4.7 A Revised Concurrency Controller

The problems outlined at the end of the previous section come about because the concurrency control type (in particular the operation *lockconflict*) has to be able to interpret the mode (and owner) information held within the lock objects in order to determine whether conflict exists. Thus the implementation of the conflict check depends upon the the semantics of the information supplied by the lock objects. The end result of which is that changes to the implementation of the lock objects will probably require changes to the concurrency control type also.

What is required is some way to decouple this dependency, such that new types of lock can be created without also having to make modifications to the concurrency control type. Fortunately, this decoupling is surprisingly simple to achieve in the object-oriented environment that has been adopted in this chapter. Essentially, all the conflict check is doing is comparing two lock objects for equality, where equality in these circumstances means that the two locks are in some sense compatible. This implies that the responsibility for determining conflict should be delegated to the actual lock objects themselves. By doing so the problem is solved in one simple operation.

The solution then is to provide the *Lock* type with an operation which allows two instances of the type to be compared. This could be done simply by providing a routine of the form:

```
boolean Lock::compare (Lock* otherlock);
```

This routine takes advantage of the fact that one lock object can be asked to compare itself against another. Fortunately, however, C++ provides a much more attractive alternative through its *operator overloading* capabilities. Using these capabilities it is possible to redefine the meaning of the standard operators (+, -, =, etc.) for user-defined types. Thus expressions of the form:

```
X = Y + Z
```

are valid providing that the meaning of such an expression can be deduced. For example, if *X*, *Y* and *Z* where all basic types (say integer) this statement behaves exactly as expected. Furthermore if all three variables were of a particular user-defined type (say the type *complex*) then, providing routines for handling the assignment operator (=), and the addition operator (+) had been defined for the type, the statement remains valid.

In C++ the name of an operator function is the keyword *operator*, followed by the operator itself. For example, *operator+* would be the name of the function that implemented the addition operator. An operator function is declared like any other operation (and can be used in exactly the same way); use of the operator itself is merely syntactic sugar for calling the function itself. Thus given a suitably declared operator function, $X + Y$ is interpreted as *X.operator+(Y)* or, in other words, call the *operator+* function of the object *X* supplying the object *Y* as a parameter.

So, in order to implement the object-oriented conflict check, the meaning of the not equal operator (*!=*) is redefined, such that if *L1* and *L2* are instances of the *Lock* type, then the expression *L1 != L2* returns the boolean value *true* if the modes of the two locks conflict, and returns *false* otherwise. Thus the declaration of the *Lock* class now becomes like that illustrated in Figure 4-10. The conflict

```
class Lock
{
      lockstatus current_status;          // status, e.g. HELD
      modetype lockmode;                  // mode of lock, e.g. READ
      Uid owner;                          // identity of lock owner
      ...                                 // other private
                                          // variables and operations
public:
      Lock (modetype);                    // Lock object initialiser
      ~Lock ();

      modetype getlockmode ();            // Interrogation operations
      lockstatus getstatus ();
      Uid getowner ();
      ...
      virtual boolean operator!= (Lock*);
}
```

Figure 4-10: The revised *Lock* class

operation is declared as *virtual* for precisely the same reasons that *lockconflict* was declared in the same fashion, that is, it is probable that a programmer will wish to redefine the notions of what constitutes conflict for different types of locks (all of which will now be derived from the basic *Lock* class). This conflict operation is part of the public interface of the *Lock* class for the following reason.

Since locks are independent objects, they can only be manipulated through their public interfaces, thus in order for another object (in this case the user-defined object derived from *LockCC*) to compare two locks, there must be a public function available to do it. The code to implement this basic conflict check is virtually identical to that of the original *lockconflict*, and is shown as Figure 4-11.

```
boolean Lock::operator!= (Lock* otherlock);
{
      if (otherlock→getowner() != owner)   // only check if locks owned by
                                           // different actions
            switch (lockmode)
            {
                  case READ:               // held mode is read
                        if (otherlock→getlockmode() != READ)
                              return TRUE;
                        break;
                  case WRITE:              // held mode is write
                        return TRUE;
            }
      return FALSE;
}
```

Figure 4-11: The *Lock* conflict algorithm

Now that this minor change has been made, it is possible to remove the keyword *virtual* from the definition of *lockconflict* and make it into a simple private operation. In addition, *lockconflict* itself becomes far simpler. The resulting interface and code is shown as Figures 4-12 and 4-13.

```
class LockCC
{
      Lock_List locks_held;                // List of all currently held locks
      Semaphore* mutex;                    // For mutual exclusion purposes
      ...
      boolean lockconflict (Lock*);        // Now private operation

public:
      LockCC ();                           // Initialise concurrency controller
      ~LockCC ();

      lockstatus setlock (Lock*);          // Set lock on this object
      ...
}
```

Figure 4-12: The revised *LockCC* class

```
boolean LockCC::lockconflict (Lock reqlock*);
{
      Lock_Iterator next(locks_held);
      Lock* heldlock;

      while ((heldlock = next()) != Null)  // iterate over all locks
      {
            if (*heldlock != reqlock)      // check for conflict
                  return TRUE;             // found - return error
      }
      return FALSE;
}
```

Figure 4-13: The revised *lockconflict* operation

Thus, the dependency between the two types (that is, the *Lock* type and the concurrency control type *LockCC*) has been removed such that it is now possible to redefine conflict for different types of lock while still relying on the concurrency controller to behave in the manner dictated by the requirements of two-phase locking.

4.8 Deadlock

As was pointed out in chapter two, locking protocols are prone to deadlock, and thus the concurrency control type described in the preceding sections is similarly capable of becoming deadlocked. Regrettably, solving this deadlock problem is not as easy as in some of the other systems that have been considered so far in this thesis.

Recall that in conventional centralised systems deadlock was usually allowed to form and was then detected and broken typically by aborting one of the deadlocked transactions. Detection frequently required the building and scanning of a wait-for graph, the nodes of which were transactions, with arcs indicating that a transaction was waiting for another. As was pointed out in the discussion of such an approach in chapter two, building this wait-for graph was complicated and made far more expensive by the introduction of distribution into the system, since concurrency controllers at each site had to exchange their local

wait-for graphs, in order to build a global wait-for graph that indicated the relationship of every waiting transaction in the system.

Utilising such a scheme in the system described here magnifies the problem still further. Although not explicitly stated previously (although perhaps implied) it is assumed that since each individual object is responsible for its own concurrency control so there will be a concurrency controller for each object active in the system. This is not an unreasonable assumption to make since the encapsulation properties of objects suggests that the concurrency control information should be private in order to allow objects to behave autonomously, at least as far as making concurrency control decisions is concerned. Thus, given the potentially large number of active objects, then in order to establish even a local wait-for graph could require substantial communication amongst the objects.

One possible approach to this problem could require that each object's concurrency controller stored sufficient information about its state into some single location on a per site basis, and then a separate process could attempt to use this information in an attempt to detect deadlock. Such an approach is probably untenable due to problems of determining when the information was consistent. This is not to say that the approach is impossible, merely that a simpler approach is available which will be described later in this section. The need for such a complicated deadlock detection system remains unconvincing at the present.

Alternatively, the wound-wait or wound-die scheme of Rosenkrantz *et al.* [*Rosenkrantz et al. 78*] (described in chapter two) could be adopted as a method of deadlock detection. Using this approach requires that there exists some means of determining age, so that the potential victim can be established when deadlock is suspected. This may require either an extra instance variable in the *Lock* class,

or possibly structuring the owner identifier such that it could be used as a timestamp.

There is, however, an even simpler solution. Make use of the traditional mechanism of timeouts to determine deadlock. Of course this strategy may mean that deadlock is falsely detected through using too short a timeout, or alternatively deadlock remains undetected for a period of time due to using too long a timeout period. However, these consequences must be accepted as the price of utilising so simple a scheme.

It is interesting to note that both Clouds and Camelot currently use timeouts for precisely this purpose, although recent reports on Camelot indicate that provision of a deadlock detector is being considered. However, since both systems are built upon special kernels, with concurrency control as part of that kernel, building such a detector is a far simpler task since all of the concurrency control information is located centrally in the kernel.

Having decided to use timeouts the problem arises of where to use them. Clouds places timeouts on *actions*; that is, if an atomic action has not completed within a given time limit it is aborted. In the system described here there are several possible options. Firstly, it would be possible to implement timeouts as a property of the *Lock* type itself, or secondly, supply a timeout parameter to the *setlock* call, or finally build the timeout into the actual concurrency control type *LockCC*.

All of these approaches are viable, and each has the same effect. Namely, if a lock cannot be set before the timeout period has elapsed then *setlock* should return with a status of *refused*. Given such a return status, the onus is then on the client to decide what to do next. The programmer may give up, try to set the

lock again, or whatever. Thus the system does not impose any particular policy upon the object designer.

It could be argued that simply returning a status is a potential source of error, particularly if the caller chooses to ignore the returned value (or simply forgets to check it). Ideally, lock refusal constitutes *exceptional* behaviour and should be handled by some appropriate *exception handler* [*Goodenough 75*]. However, since C++ does not currently support exception handling, return codes must be persevered with, error-prone as they may be.

The discussion above has stated that there are several options open in order to incorporate a timeout mechanism into the basic concurrency control type. Since each is possible, they are briefly described in the following sections.

4.8.1 Modifying the Lock Type

The first alternative allows instances of the *Lock* type itself to carry the timeout value with them. Setting the value of the timeout could be handled in many ways. For example, it could be set to some default value in the *Lock* constructor (or, by appropriate overloading of the constructor, set to some specific value). Similarly, public operations could be provided to allow the timeout value to be set. Indeed, a combination of both approaches is possible.

4.8.2 Extending Setlock

Instead of modifying the basic *Lock* type, the *setlock* operation could be modified such that it took another parameter which indicated the timeout value to use for this particular call. If the language supports a default parameter mechanism whereby parameters not supplied in a call are set to default values then this is a particularly attractive technique since existing code does not

require changing (the default value would be used), yet the programmer can now specify a particular timeout value if required.

This approach is supported by C++ and so it is also possible to define the interface to *setlock* as follows:

```
lockstatus setlock (Lock*, int timeout = 20);
```

In this case, if no second argument is supplied on any given call then the default value (in this case 20 time units) would be used.

4.8.3 Modifying the Concurrency Controller

In the same way that instances of the *Lock* type could be modified to carry a timeout value, so similar modifications could be made to the basic concurrency control type *LockCC* so that one of its instance variables is just such a timeout value. Setting the value of the timeout can then be handled in basically the same way as it was handled for the *Lock* type, that is, the timeout value could be set to some default value in the *LockCC* constructor (or, by overloading of the constructor, set to some specific value).

Of the three options presented, this latter one is probably the least flexible, since it only allows a single timeout value to ever exist for the object. Both of the two previous approaches allow different timeouts to be supplied with each lock request, thus providing the maximum flexibility. For example, with both of the previous approaches, if the request to set a lock was refused, the caller may wish to try again but with an increased timeout value. This is impossible with this last approach.

4.9 Handling Atomic Action Nesting

As described so far in this chapter the concurrency control type is incapable of handling nested atomic actions for two reasons. Firstly, when attempting to set a lock the concurrency controller takes no account of the available ancestry information. That is, when two locks are compared, they are considered compatible if they belong to the same action or if their modes are compatible. Secondly, although the *Lock* type has a status indicating whether it is *held* or *retained,* the concurrency controller does not use this in any way.

Both of these problems can be overcome by simple additions to the concurrency control type *LockCC*. Overcoming the first problem requires the addition of a private operation *isancestorof* to the concurrency control type. This operation determines whether the owner of each held or retained lock is an ancestor of the owner of the requested lock. This relationship is then tested in accordance with the nested locking rules given in chapter three.

The second problem requires that locks have their status changed when an atomic action commits. There are two distinct cases here. If a nested action is committing then the locks should be propagated to the parent action, otherwise if the action is a top-level one, then the locks should actually be released. Thus another operation, *propagate*, is added to *LockCC*. This operation has the task of ensuring that the ownership of any locks held by the object on behalf of the action is changed to that of the parent action, and that the status of the locks are similarly changed from *held* to *retained.*

In addition to lock propagation, lock release is also not handled by the concurrency control type. This situation clearly needs amending, otherwise locks would persist as long as the object itself was active. To this end, the operation *releaselock* is also added to *LockCC*. Given these two operations (*propagate* and

releaselock), it is then necessary to decide precisely which should be called when an atomic action commits.

There are two alternatives to consider; which is followed depends upon how atomic actions have been implemented. For example, *propagate* could be called directly by the atomic action system implementation when a nested atomic action is being committed, leaving *releaselock* to be called only when a top-level action commits. Alternatively, *releaselock* could always be called, and it could determine whether to propagate the lock or release it based upon the action nesting level prevailing at the time. Which approach is followed is determined by the implementation of the atomic action system. Thus, if the atomic action implementation distinguishes between top-level and nested commits such that different protocols are followed, then *propagate* should be called for nested action commit, with *releaselock* only being called when the top-level action commits (or aborts).

4.10 Other Issues

This section considers some other relatively minor issues that have not been considered elsewhere.

4.10.1 Lock Conversion

In chapter two it was noted that it was possible for an atomic action to first set a read lock upon an object and then at some later point in time decide to set a write lock on the object as well. This procedure was termed *lock conversion.*

Earlier in this chapter it was stated that such conversion would not be allowed due to the immutability of the mode of each lock object. This does not mean the same same effect cannot be achieved; rather it must be achieved in a somewhat different fashion. The illusion of lock conversion can be achieved automatically using the inheritance based scheme of this chapter with no further

work or modification to the existing design for the following reasons. Firstly, locks from the same action are not considered to conflict with each other Thus even though an action may already have set a read lock upon an object, attempting to set a write lock at some later time will be allowed providing that no *other* action is also holding a read lock on the same object.

If another action does hold a read lock then a conflict will exist and the attempt to set the write lock will not be allowed until the conflicting lock is released. This is correct behaviour since the net effect is that eventually only a single action is manipulating the object and the attempt to set the write lock will then succeed. This means that the list of locks on the object will consist of the new write lock plus the original read lock, both of which belong to the same action.

This conversion process is of course prone to deadlock if two independent actions both attempt to convert their existing read locks to write locks, since they will each end up waiting for the other to release the read locks they respectively hold. However, this deadlock can be handled in the same way as before, so that one of the requests for a write lock will eventually time out and be refused, causing an error return. What happens after this error return is determined by the implementor of the type.

As an alternative approach, the ISIS strategy could be adopted and an explicit *promotable read* mode lock could be declared that is basically exclusive in nature. With this type of lock it would then be possible to disallow the illusion of conversion of normal read locks to write locks.

Within the system under consideration such promotable read locks are simple to implement without further modification to the basic scheme. Once again use is made of the type inheritance capabilities of the language and a new

lock type - the *PLock* (illustrated in Figure 4-14) - is created, together with an appropriate declaration of its conflict operation (Figure 4-15).

```
class PLock: public Lock
{
      virtual boolean operator!= (Lock*);

public:
      PLock (modetype);
      ~PLock ();

      ...
}
```

Figure 4-14: The *PLock* class

```
boolean PLock::operator!= (Lock* otherlock);
{
      switch (lockmode)
      {
            case READ:   // Read compatible with all except Write
                  if (otherlock->getlockmode() == WRITE)
                        return TRUE;
                  break;
            case PREAD: // Pread ok with Read or same owner Write
                  if (otherlock->getlockmode() == READ)
                        break;
                  if (owner != otherlock->getowner())
                        return TRUE;
            case WRITE: // Exclusive unless same owner
                  if (owner != otherlock->getowner())
                        return TRUE;
      }
      return FALSE;
}
```

Figure 4-15: The *PLock* conflict algorithm

Plocks are identical to *Locks* except that they support the additional mode PREAD (for promotable read) and have their own version of the conflict check. This check no longer allows lock attempts by the same action to proceed unhindered. Instead it checks *all* attempts to set a lock for conflict regardless of the source of the request. Thus in this case WRITE lock requests from the same action will always cause conflict with existing lock requests from the same action. The only way a WRITE lock can be granted using this conflict check is if the same action had already acquired a PREAD lock (or already holds an existing WRITE

lock). This particular implementation allows READ locks to be compatible with PREAD locks, however, only one PREAD lock is allowed on the object at any given time (it is assumed that the action does not attempt to set two or more PREAD locks on the same object, although the conflict check could easily be made to cope with this situation). This ensures that there can be at most one attempt to convert such a lock to a WRITE lock, thus avoiding any possibility of deadlock.

Either of these two approaches is acceptable, but the very fact that both can be supported in so simple a fashion emphasises once more the flexibility of the basic design and the applicability of the type-inheritance approach.

4.10.2 Managing the Lock List

As locks may be inherited from child actions, there is likely to come some time when the list of locks being maintained by the concurrency controller for an object becomes unwieldy and requires pruning. For example, if an action had been retaining a read lock on the object, and then inherited a write lock from one of its children it would end up retaining two locks, one in each mode.

Obviously, in this case the read lock is no longer strictly necessary and can be released. This process of *lock merging*, must ensure that the correct lock is released (here the read lock, not the write lock) and thus requires that the lock modes form some total order. Given that lock modes can be ordered then it is a simple matter to ensure that the lesser is released when the merge occurs.

Ordering of locks can be handled once again by a simple modification to the basic *Lock* class. All that is required is a new virtual function *ordering*, that compares the modes of the two lock instances and returns an indication as to which has the stronger mode (or whether both have equal strength modes). Alternatively, (as before) one of the standard operators could be overloaded (say $<$) to perform the same function.

4.10.3 Ensuring Two-Phase Locking

In order to ensure that the concurrency controller follows strict two-phase locking it must not release any locks until the atomic action commits or aborts. This requires that certain operations of the concurrency controller (in particular *propagate* and *releaselock*) are called by the atomic action system when an atomic action commits or aborts rather than directly by the programmer. In order to do so, the atomic action system needs to be informed as to which objects each action has manipulated and the locks that have been set. The precise form of this information is given in the next chapter, however, what follows is a brief overview of the processing involved.

Essentially, what happens is that when a lock is set, an indication is sent to the atomic action manager giving it sufficient information to identify the lock object and the actual object upon which the lock is being set. Then, as part of the standard commit processing performed by the atomic action manager the lock information registered with the manager is used to call the *releaselock* operation of the object, passing the lock identification as a parameter. Since this call only occurs as part of the commitment of an atomic action, the following of the strict two-phase protocol is assured.

As an aside it may be noted that it is also possible for the two-phase policy to be subverted deliberately by explicit use of *releaselock* by the programmer. If this occurs it is assumed the programmer knows what he (or she) is doing. In this respect our system is similar to both Clouds and Camelot which allow the same operations and make the same assumptions. In reality there are instances where *releaselock* must be explicitly called by the programmer. This situation arises if a call was made to *setlock* while the program was not executing as part of any atomic action. In this situation the programmer must release the locks explicitly.

In order to assure that a two-phase policy was still being followed once a lock was released explicitly it is possible to refuse to set further locks, but that would not overcome the possible problem of cascading aborts that might then follow.

4.11 Summary

This chapter has considered how to apply one of the concurrency control techniques of chapter two to an object-oriented environment. In doing so individual objects have been made directly and explicitly responsible for their own concurrency control which it is argued is the correct thing to do, bearing in mind the properties claimed for objects, particularly with regards to encapsulation.

This control was added in a novel and evolutionary way by using the type-inheritance capabilities of the implementation language. This had the highly desirable features of being both flexible and not requiring modifications to, or the design and implementation of, either a new language or operating system.

Taking this approach further, it was claimed that locks ought to be objects in precisely the same sense as any other object in the system and so should not be regarded as pre-defined (and thus frequently immutable) system types. This approach is radically different to that adopted by the other object-based systems that have been considered in this chapter.

In support of this approach, a *Lock* type was designed that supported the basic modes of read and write and it was shown how by giving it an appropriate interface such an object could be used by a concurrency control type. A two-phase locking based concurrency control type was then designed that could be inherited by user-defined types, such that in conjunction with the *Lock* type the correct two-phase behaviour was obtained.

Finally, the problem of deadlock was considered, and it was explained how this could be handled by use of the simple expedient of timeouts. Note that there is no commitment to adopting this approach in the design, since other approaches are possible, however, they are more costly. The experience of the designers of other object-based systems shows that the use of timeouts has, in general, proved adequate. The need for a more complicated deadlock detection scheme is, as yet, unnecessary.

Throughout this chapter it has been claimed that using type-inheritance in the manner described here is a flexible approach to adopt. In chapter six further examples will be given in support of this claim.

Chapter 5
Implementation in Arjuna

This chapter shows how the concurrency control type designed in the previous chapter was implemented as part of one particular system - *Arjuna*[†]. Arjuna is an object-oriented programming system that supports the construction of reliable distributed application programs.

The following sections describe Arjuna in more detail. In particular they describe the Arjuna system model and the class hierarchy upon which the entire system is based. They then show how the concurrency control type designed in the previous chapter is integrated into this hierarchy and consider the facilities that are required to enable the concurrency controller to function as part of the Arjuna system.

The chapter then describes some of the problems that the model of computation employed by Arjuna has on the implementation of the concurrency control type, together with ways by which these problems can be overcome.

Finally, the chapter shows the actual implementation of the concurrency control type of the previous chapter in Arjuna and gives sample performance details for this particular implementation. A more complex example is then described. This latter example is based upon a simple diary system that allows users to note when specific events are due to happen and is designed to show the

---

[†]In the Hindu epic *Mahabharata*, Arjuna is a warrior prince whose chariot is driven by Lord Krishna.

ease by which the facilities of Arjuna can be used by a programmer to create concurrency controlled objects.

5.1 Arjuna

Arjuna [*Shrivastava 86, Dixon et al. 87, Parrington and Shrivastava 88, Shrivastava et al. 88*] is an object-oriented programming system that supports the construction of reliable distributed applications. The initial goal of the project was to utilise as much as possible of the theoretical work on reliability that had been carried out at Newcastle University over the years [*Shrivastava 85*]. In addition some practical work was also available including a remote procedure call (RPC) mechanism that supported orphan killing. This mechanism - *Rajdoot* [*Panzieri and Shrivastava 88*] - was already being modified to incorporate facilities for multicast remote procedure calls [*Hedayati 88*] based upon a new multicast communication system [*Hughes 86*] which it was felt would be helpful in several areas, but particularly in the commit processing mechanism.

Arjuna is being implemented in the language C++ [*Stroustrup 86*] upon a set of UNIX workstations connected by an Ethernet. As has been emphasised earlier in this thesis, the aim of the project has been to provide support for reliable distributed programming without resorting to producing a new programming language, operating system, or combination thereof. Rather, the project aims to exploit features provided by the implementation language and the host operating system.

Objects in Arjuna are *persistent* (their lifetime exceeds the lifetime of the program that created them) and are the main repositories for holding the state of the system. Objects are normally stored in an object repository named *Kubera* [*Dixon 88*], which provides the necessary stable storage mechanisms to ensure that node crashes do not destroy objects stored within it. Kubera is a general purpose object store and holds not only the images of persistent objects, but also

certain critical information about the system such as the state of any atomic actions in the process of being committed.

5.2 The Arjuna System Model

The basic layered architecture of Arjuna is shown in Figure 5-1. Objects in

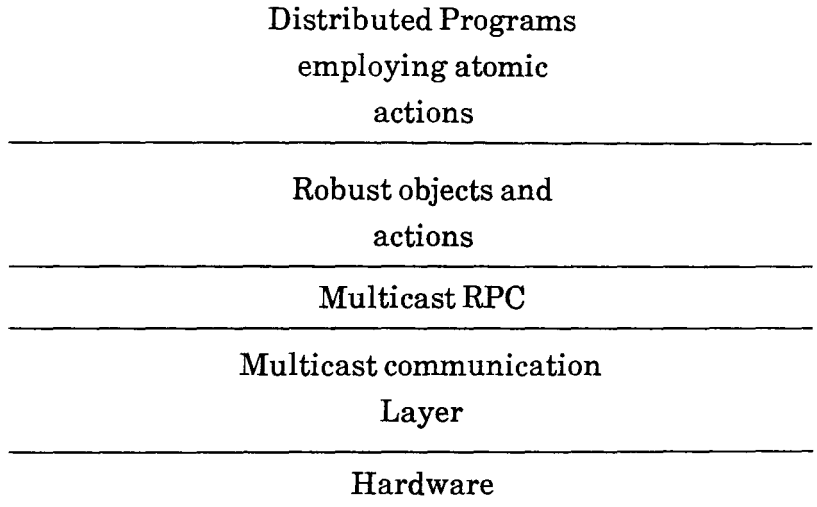

Figure 5-1: The architecture of *Arjuna*

Arjuna may be either *passive* or *active*. When in a passive state an object resides in the object store of the node at which the object is located. Arjuna objects are assumed to be located in their entirety at only a single site.

In order for an operation to be performed upon an object the object must first be *activated*. Once activated it remains active until the top-level action responsible for its management commits, or the action manipulating the object aborts. Note that objects may actually be activated by nested actions but providing that the nested action commits the object will remain active and will be inherited by the parent action. Arjuna thus differs from systems such as Argus [*Liskov 88*] and Camelot [*Spector 87, Spector et al. 87*] where object servers

(guardians in Argus terminology) are permanently running processes that are automatically started as part of node start up, and are guaranteed to be restarted after a node crash.

For the sake of consistency and simplicity Arjuna makes no attempt to differentiate between local and remote objects. That is, local objects are handled in the same way as remote objects. In practice this means that even local objects are accessed via remote procedure calls (RPCs) [*Nelson 81, Birrell and Nelson 84*]. While this may seem inefficient the uniformity of access that it affords has its benefits from the point of view of stub generation. In particular the use of remote procedure calls can be completely hidden from the programmer.

To make the distribution of objects hidden from the programmer Arjuna employs a stub generator [*Wheater 88*] that takes definitions of the interface to a type and produces an equivalent *stub type*. This stub type provides the same interface to the programmer as the original type, only the implementation of the actual operations of the type has changed. Instances of the stub type are termed *stub objects*.

Each operation of the stub type is responsible for packing the parameters of the operation into a form suitable for transmission over the communications medium and invoking a remote procedure call to a *server* process at the site where the object actually resides. This server then unpacks the parameters, performs the requested operation locally upon the object, packs the result and returns it to the *client* stub object. The client stub object waits for this reply and when it is received unpacks the result and returns it to the caller exactly as if the call had been performed locally. This sequence of events is shown in Figure 5-2.

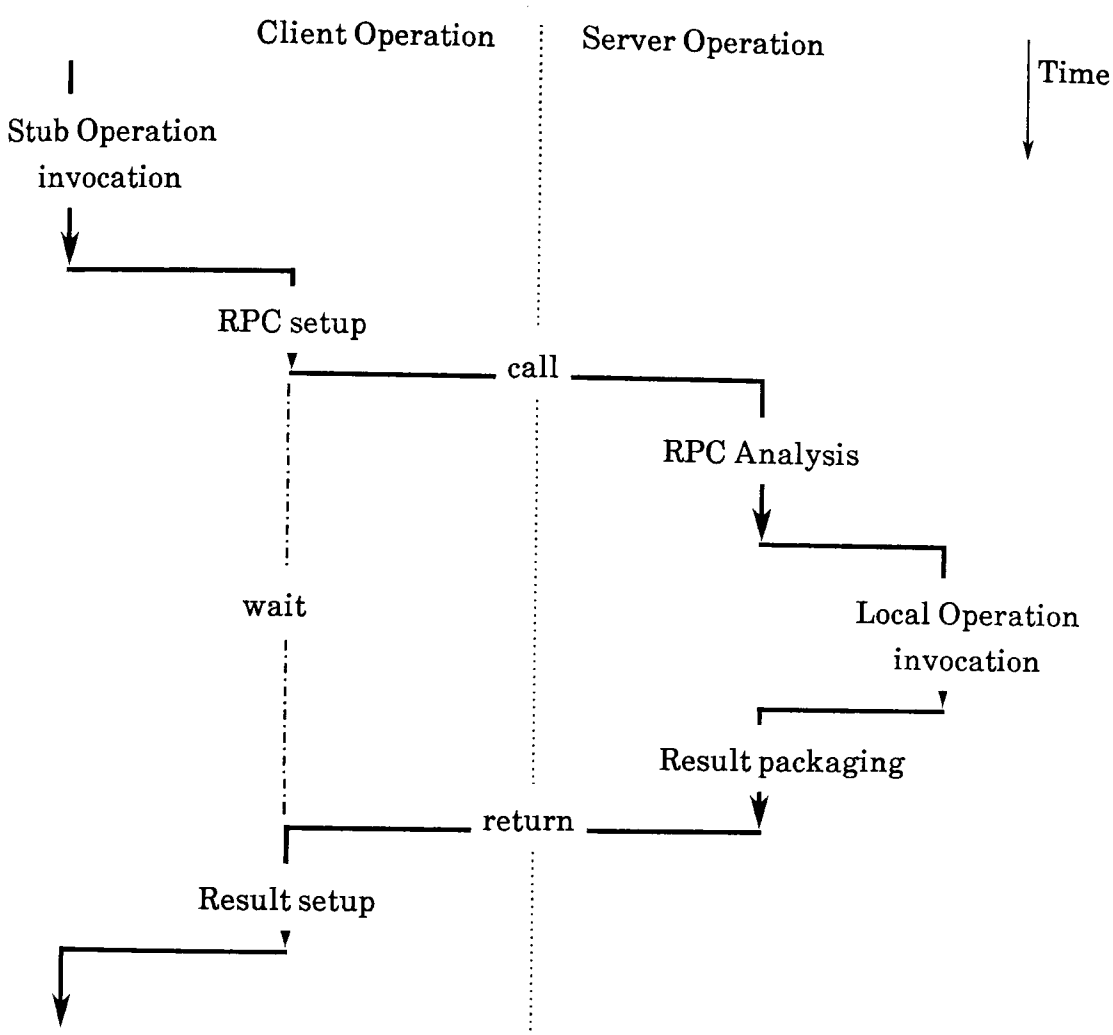

Figure 5-2: Remote operation invocation

This procedure is transparent to the programmer since whenever an instance of a type is declared in the program, an instance of the corresponding stub type is instantiated instead. Thus the programmer continues to invoke operations upon the stub objects as if these stub objects were the real objects.

The stub object and the code that has to be executed in order to pack and unpack the parameters for each of the operations of the actual object (at both the client and the server) are automatically produced by the stub generator from the interface definition.

Recall that in C++ objects have special operations known as *constructors* and *destructors* associated with them. The code produced by the stub generator takes advantage of this fact to determine when to create the server process for each object. When the stub object first comes into scope (that is, it is created) the constructor for the stub object is called. This constructor determines the site at which the real object resides by interrogating some name server and then makes a remote procedure call to a manager process running at that site requesting creation of a server process. When the server process has been created the manager sends back to the stub object constructor an address by which the server process may be contacted directly. All future remote procedure calls are directed to the server process without further involvement of the manager at the remote site.

Similarly, when the stub object goes out of scope, the destructor operation ensures that the communication channel is terminated after first instructing the server to terminate (which in turn will cause the remote objects to be passivated).

The RPC system employed by Arjuna is a modification of Rajdoot, a system that provides *exactly once* semantics. That is, if the client receives a reply then exactly one execution of the called operation has taken place. If the client does not receive a reply then either one, none, or a partial execution of the operation may have taken place. The simplest course of action to take in this situation is to abort the action from which the call was made (assuming that the operation is executing under the control of some atomic action).

Although not shown above, there may be further recursion in the system since a server may itself be a client to some other server. Thus at any instant there may be many clients each with possibly multiple servers, each of which may, in its own right, be the client of yet more servers.

Such an approach naturally leads to a tree-like structure of client and server processes and this was the model employed originally by Rajdoot. This implied that remote procedure calls destined for the same object but which originated from different nodes resulted in the creation of additional server processes. Thus it was possible that any single object might have several servers active for it at any given time.

Since this situation was considered undesirable, (it provides several management difficulties) this basic RPC mechanism was modified so that servers can now be shared by more than one client, providing that the clients are related. The implications of this will be considered later in this chapter in section 5.5.2 and again in chapter six.

5.3 Atomic Actions in Arjuna

Arjuna is unlike any of the other object-oriented systems reported in the literature (and briefly described in the previous chapter) that have been developed over recent years in that every major entity in the system is an object. This idea even extends to the notion of presenting an atomic action as simply another object in the system, as opposed to it being implemented as part of an operating system or built into a special programming language.

Thus atomic actions in Arjuna are manipulated and declared in the same way as other objects. In particular, there is a class called *Action,* a skeleton of which is shown as Figure 5-3. This class provides the basic operations associated with atomic actions as outlined in chapter three and leads to programs that resemble the simple example shown as Figure 5-4 which illustrates a sequential nested action B inside the top-level action A. Action management and the implementation of the failure atomicity properties of objects are not the concern

```
class Action
{
      ...                                   // private action management
                                            // functions and variables
public:
      Action ();
      ~Action ();

      Begin_Action ();
      Commit_Action ();
      Abort_Action ();

      Action* Parent ();
      ...
}
```

Figure 5-3: The class *Action*

```
main ()
{
      Action A, B;                          // declare the two actions

      A.Begin_Action();                     // commence action A
      {
            ...
            B.Begin_Action();               // start nested action B
            {
                  ...                       // operations of action B
            }
            B.Commit_Action();              // commit B
            ...
      }
      A.Commit_Action();                    // finally commit A
      ...
}
```

Figure 5-4: The class *Action* in use

of this thesis and are only covered briefly here to give the reader an overview of the Arjuna system. For precise details see [*Dixon 88*].

Instances of the class *Action* maintain as part of their private state all of the necessary information regarding the current status of the action (running, committing, aborting, etc), together with a special list of records that records information required to achieve the properties of failure atomicity and permanence of effect. Also held on this list are records detailing actions taken by

(or yet to be taken by) the concurrency controller of each object. This will be described in more detail in section 5.5.

5.4 The Arjuna Class Hierarchy

One of the key concepts of Arjuna is its use of the properties and facilities of the implementation language C++ and the host operating system to provide support for reliable distributed programming using atomic actions. This support is added by the declaration and use of appropriate classes responsible for implementing the various portions of atomic action management.

These classes form a hierarchy, a basic illustration of which is given as Figure 5-5. At the root of the entire hierarchy is the class *Object*. This class

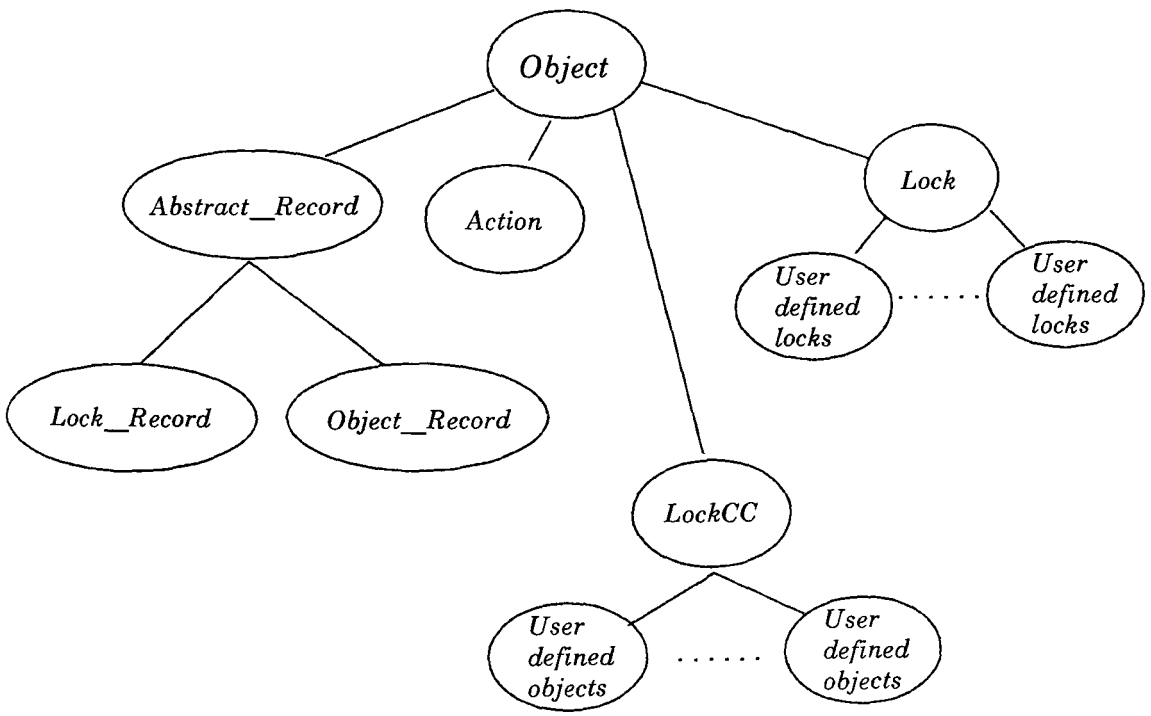

Figure 5-5: The *Arjuna* class hierarchy

provides the basic facilities used by all of the other classes. In particular, it contains the name of each object (in terms of a unique identifier) and operations to determine the size and type of an object. It is also responsible for interaction with the object store, especially with respect to object activation and passivation.

Since *Action* is a class derived from *Object* it inherits the attributes of that class. This allows, for example, actions to be named by their unique identifiers rather than any other way, such as associating some form of hierarchical name that reflects the action nesting. However, it should be pointed out that since *Action* provides an operation called *Parent* such a hierarchical name could be generated if required simply by following this parent chain back to the top-level action (which is identified by the fact that it has no parent). The implementation of the concurrency control type *LockCC* uses this unique identifier to associate all of the locks of any given atomic action together.

Instances of the class *Abstract_Record* are not meant to be instantiated in any way (in fact all of the operations of this class are designed to return errors if they are invoked). Instead, *Abstract_Record* is used as a template for the declaration of several other management utility classes. The operations provided by *Abstract_Record* correspond to those of *Action* and include such operations as *begin, top_level_commit, abort*, etc. The use of one particular type derived from *Abstract_Record* - the *Lock_Record* - will be described in section 5.5.

In section 5.3 it was mentioned that instances of *Action* maintained a list of records for management purposes. In actual fact these records are simply instances of the record types derived from *Abstract_Record*, that is, *Object_Records, Lock_Records*, etc. Whenever an operation on an instance of *Action* is performed it, in turn, causes the equivalent operations to be invoked on each record instance held on its *Record_List*.

When an atomic action commits, *Action* invokes the equivalent commit operation for each record in its current *Record_List*. When an nested atomic action commits certain information must be propagated to the parent atomic action. This propagation is necessary so that if an object was manipulated for the first time by the nested atomic action then the parent of the action can assume responsibility for the management of the object. Similarly if the object was already known to the parent then the duplicate information can be discarded since the action already knows about the object. Thus the *Record_List* behaves in a similar fashion to a *recovery cache* [*Lee et al. 80*].

5.5 Adding the Concurrency Controller to Arjuna

The final sections of previous chapter outlined the mechanism by which the concurrency control type designed in that chapter could be integrated into a system that supported atomic actions. This section shows how that mechanism is provided in the Arjuna system.

As can be seen from Figure 5-5 the concurrency control type described in the previous chapter is in actual fact derived from the root class *Object* and thus inherits the capabilities it provides. Since all of these features are really required by the user-defined objects themselves, then *LockCC* publically inherits *Object*, and is itself publically inherited by the user-defined types.

5.5.1 Ensuring Strict Two-Phase Locking

As was pointed out in chapter four, the concurrency controller of an object needs some way of recording with the atomic action system that it has set a lock upon the object. When the action commits the concurrency controller for the object can then be instructed either to propagate, or to release the lock as appropriate, depending upon whether the committing action is nested or not. In Arjuna this communication is enabled by the ability of the concurrency controller

for an object to add records to the *Record_List* of the appropriate action. For this purpose (although not shown in Figure 5-3) *Action* also provides as part of its public interface an *add* operation, the basic declaration of which is shown below:

```
int add (Abstract_Record*);
```

In addition, there is always a pointer to the current action available under the name *Current_Action*. Thus whenever a lock has been successfully set upon an object the concurrency controller for that object can inform the atomic action system of the fact by simply executing the following statement:

```
Current_Action->add (new Lock_Record(reqlock->get_owner(), this));
```

This statement creates a new instance of the class *Lock_Record* and passes to its constructor the unique identifier of the owner of the lock object and a pointer to the actual concurrency controlled object. This newly created record is then added to the list maintained as part of the state of the current action.

The declaration of the class *Lock_Record* is shown in Figure 5-6. The only information this class maintains is the owner of the lock (in terms of the unique identifier of the atomic action setting the lock) and a pointer to the appropriate concurrency controlled object. Furthermore, these private variables can only be set through the constructor operation; no further manipulation of them is permitted. Thus, each instance of *Lock_Record* contains sufficient information to enable appropriate actions to be taken to ensure that locks set by an action are all correctly propagated or released depending upon the ultimate fate of the action. Since locks are themselves objects (instances of the class *Lock* which is derived from *Object*), they possess a unique identifier by which they too can be named.

```
class Lock_Record: public Abstract_Record
{
      Uid* action_uid;                    // unique id of owner atomic action
      LockCC* object_address;             // pointer to concurrency controlled
                                          // object

      virtual void pack (Image* );        // Arjuna required operations
      virtual void unpack (Image* );

public:
      Lock_Record (Uid*, LockCC*);        // create new lock record
      ~Lock_Record ();

      virtual void begin ();              // operations lock records respond to
      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_ ();

      virtual Record_Type TypeIs ();      // operations required by Action
      virtual UnTyped Value ();
      virtual int ordering ();
      ...
}
```

Figure 5-6: The class *Lock_Record*

Lock_Records must re-implement all of the functions they inherit from *Abstract_Record* since the base class operations are designed to return an error if they are ever invoked. As a rule most of these operations are simply redefined to be null operations, so that if they are called the operation returns immediately.

In Arjuna, the implementation of the class *Action* makes a distinction between the commitment of a nested atomic action and the commitment of a top-level atomic action. There are seven distinct operations required to cope with this. *Nested_prepare* and *nested_commit* are invoked when a nested action commits. *Top_level_prepare, top_level_commit,* and *top_level_abort* are invoked as appropriate during execution of the two-phase commit protocol.

The implementation of all of these operations is simple as far as instances of *Lock_Record* are concerned since all they are required to do is trigger the release or propagation of the associated lock at some object depending upon whether a

commit or an abort is being performed. In section 4.10.3 of the previous chapter two possible approaches to lock release and propagation were described, and it was stated that which approach to adopt depended upon how the atomic action system treated nested action commitment. Since *Action* distinguishes between nested and top-level commits the implementation of these operations for the type *Lock_Record* are different and are shown as Figure 5-7.

```
void Lock_Record::nested_commit ()
{
      object_address->propagate (action_uid);
}

void Lock_Record::abort ()
{
      object_address->releaseall (action_uid);
}
```

Figure 5-7: The implementation of nested commit and abort for *Lock_Record*

The additional functions provided by *Lock_Record* are for the benefit of the atomic action management system. For example, *ordering* is used during the record list merging process, while *Value* simply returns the value of the *action_uid* member variable.

Once a decision has been made to abort or commit a top-level action, that decision should carried out regardless of any crashes by any node in the system. Arjuna handles this by utilising a form of *intentions list* coupled with the ability for instances of each of the action management classes to save sufficient information about themselves in the object store.

Basically what happens is the following. When top-level action commit is invoked, *Action* uses facilities provided by *Object* to retrieve the state of each of the records currently held on its record list and saves that state in the object store. The commit operation of *Action* then invokes the *top_level_prepare* operation

upon each record in turn. If this succeeds for all of the records in the action then the first phase of the commit process is considered successful and the intentions list (which records the unique identifier of each record) is also placed in the object store. The second phase is then started which requires re-scanning the record list performing the *top_level_commit* (or *top_level_abort* if the action is being aborted) operation on each record and then removing the corresponding record from the object store.

Since the information that needs to be saved differs from record type to record type, each provides a type-specific *pack* function which is called automatically by *Object*. This function packs the state of the record instance into a contiguous block of memory that is handled by a class called *Image*. For symmetry purposes there is also an *unpack* function which performs the reverse operation. For more complete details of this procedure, see [*Dixon 88*].

5.5.2 Implications of the Arjuna System Model

Earlier in this chapter the system model employed in Arjuna was described. This section describes what effects this model has upon the basic concurrency control scheme designed in the previous chapter.

The most obvious problem that arises comes about due to the fact that it is possible for an object to be managed by more than one server at any particular site. This can quite naturally lead to severe consistency problems. As was pointed out in section 5.2 the model of computation supported by *Rajdoot* was extended to allow for server sharing. However, such sharing was only allowed between related processes, which in this context means servers that have come into existence due to the execution of related atomic actions. This situation is shown in Figure 5-8.

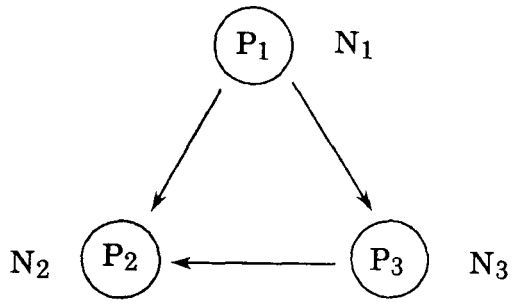

Figure 5-8: *Arjuna* process structure

In this example a client program P_1 (at node N_1) has accessed an object at node N_2 resulting in the creation of the server process P_2. P_1 has also accessed another object at node N_3 resulting in the creation of the server P_3. An operation performed by P_3 (on behalf of the original client) is then assumed to require the invocation of some operation on the object at N_2 already served by P_2. Thus in this case P_3 is allowed to share the server P_2 with the original client P_1.

If, however, some totally unrelated action running at N_3 attempted to perform some operation on the object managed by P_2 then an entirely new server (call it P_4) would be created. Thus the object at N_2 ends up by being served by both P_2 and P_4. It is the task of the concurrency controller for the object to ensure that this situation does not lead to any inconsistencies. The concurrency controller does this by controlling when an object is activated.

Recall that objects in Arjuna, when passive, are stored in the object store *Kubera*. This object store is particularly simple minded in that, while it ensures that the object is stored reliably, it contains no access control mechanisms whatsoever. Thus if two servers attempt to activate an object, both would be allowed to load it from the object store independently of each other without any access check taking place.

Since this is the case, it is the concurrency controller for the object that must determine when the load of the object from the object store should take place. It would be possible, for example, for the object to be loaded as part of the invocation of the constructor operation at the object's server, but that leads to precisely the situation we are trying to avoid.

Consider the simple case provided by the basic system where the object can only set read or write locks upon itself (ignoring the potential problems caused by lock conversion for now). If the object is only being read then having multiple servers, each with a copy of the object does no harm whatsoever, since each server has a copy of the latest state of the object. Problems only arise when (at least) one of the servers wishes to modify the object in some way. Obviously each server cannot be allowed to have its own copy of the object in this case, since the modifications each performed would not be reflected in the final object state. Rather only the modification performed by the last committed server would be reflected - leading to the classical problem of lost updates. Thus only one server must be allowed to modify an object at any given time. Fortunately, this situation is fairly easy to achieve since in order to modify the object the server would have to first obtain a write lock on the object, and by the standard locking rules only one such lock can exist at any one time.

Obviously the modification should take place upon the latest state of the object which requires that the object is only actually activated (and thus cause its state to be loaded into the server) after an appropriate lock has been acquired. For this purpose, the base class *Object*, provides a routine called *activate*, the purpose of which is to determine if the object has been activated, and if not cause it to be activated by loading the latest state of the object into the server. If the state has already been loaded, *activate* simply returns without doing anything.

Using this mechanism it is simple to ensure that only one server has modify access to an object and thus lost updates are not possible. The sequence of events is thus now:

(1) The stub object comes into scope and causes creation of a server process at the home site of the actual object. The server process sends its address to the stub object so that communication between the client and the server requires no third party. As part of the server creation an instance of the required object is created and initialised into a (type-specific) default state.

(2) The client program invokes some operation on the stub object which is translated into a remote procedure call to the appropriate server requesting the execution of the operation.

(3) The actual code for the requested operation attempts to set a lock of the appropriate type upon the object by calling *setlock*.

(4) If the lock cannot be granted the server blocks until it can be. Once the lock is granted, *setlock* calls *activate* to ensure that the actual object state has been loaded into the server from the object store (that is, the object becomes active at this point in time).

(5) The *setlock* call returns, the requested operation is performed and the results are returned to the client.

As can be seen from the above description *activate* must be called on every attempt to set a lock on the object. While this may seem to be an unnecessary overhead it is probably not so, since in the majority of cases *activate* will return immediately without doing anything.

An alternative approach is the following. The state of the object is automatically loaded as part of the invocation of the object's constructor at server startup. This means that read accesses do not need to perform *activate* calls since they already have the state present. Only calls that modify the object require *activate* calls to ensure that once the lock is set the latest current state is loaded into the server.

While this latter scheme will work in the simple case that has been under consideration here, it still has its problems. The most notable problem is that under this scheme the concurrency controller for the object needs to know which locks will be set when the object is to be modified, to be sure that *activate* is called when locks of that type are set.

In the simple case where only read and write locks may be set this is easy to determine, since write locks are the only lock type that allows object modification and so the concurrency controller for the object could simply check for those. However, as shall be seen in the next chapter, when the types of locks that may be set on an object are extended in a type-specific manner this check is not guaranteed to be effective.

Of course a requirement could be made that all locks in addition to a mode (such as *read* or *write*) also carry some indication as to their type (say either *examine* or *modify* along the line of Clouds [*Dasgupta et al. 85*]) which could then be tested by the concurrency controller of the object, but for now this approach is discounted and instead calling *activate* on every attempt to set a lock is preferred.

The above discussion has assumed that the object is not a new object that is being created by the calling action. If this were the case then the call to *activate* is strictly unnecessary, since the object does not yet exist in the object store. Given

that the status of an object can be determined by calling an operation provided by *Object* it is simple to avoid the activation in this case.

Lock conversion could also be a source of problems if two servers attempted to convert their read locks to write locks since lost updates could reappear. However, the techniques of the previous chapter suffice to cover this situation effectively. Firstly, if the extended *Plock* style of locking is used then only one server will be allowed to convert from read access to write access on an object, thus ensuring that only one server will update the object. Alternatively, using the normal style of locking, the conversion would cause deadlock to occur, which, in turn, would cause one of the conversion requests to fail, thus still leaving only one server capable of updating the object.

5.6 Further Complications

The previous section showed how the use of multiple servers for each object in Arjuna could cause inconsistencies to occur without modifications to the basic concurrency control scheme described in chapter four.

The following sub-sections outline some other problems that this model of computation has together with means by which they can be overcome.

5.6.1 Concurrency Control State

The problems already described in the previous section, which were to do with the actual state of the object, have analogous problems to do with the state of the concurrency controller for the object. This section describes these problems and considers ways by which they can be solved.

The state of the concurrency controller for an object is essentially determined by the list of *Lock* objects that it is currently holding or retaining on behalf of the various atomic actions ongoing in the system. In order to be able to

decide whether any individual request to set a lock can be granted, the concurrency controller for the object needs complete knowledge of all the locks that are currently set upon the object. Unfortunately, if multiple servers exist for an object then this information is actually distributed amongst all such servers.

Thus the concurrency controller in each server for a single object only knows what locks have been set by that server. This is an intolerable situation since it can lead to inconsistent decisions by the concurrency controller. For example, consider an object for which two servers exist. It could be that one server holds several read locks on the object, while the other server is being asked to set a write lock on the same object. In this case, without knowledge of the read locks held by the first server, the second server may inadvertently allow the write lock to be set, creating obvious consistency problems. This problem can be overcome in several possible ways:

Using Multicast Communications

One possible way by which this problem of incomplete knowledge could be overcome is by allowing the concurrency controllers in each server to communicate with each other so that a consensus can be arrived at whenever a lock is to be set.

However, since servers are created dynamically, determining how many servers exist and how to communicate could be a problem. One solution is to use multicast communication, such that the servers for an object all form part of a common multicast group [*Hughes 86*]. Thus when servers are dynamically created they *join* the appropriate group and when they die they *leave* the appropriate group. It is then the task of the multicast communications software to ensure that all members of the group receive messages.

Creating such a multicast group is a simple procedure. Since each server is responsible for an object that has a unique identifier, this unique identifier can be used to generate a unique multicast group identifier for the purpose of communication. Furthermore, since all of the servers for the same object would see the same unique identifier for the object, they will each generate the same group identifier, thus ensuring that all active servers for an object (and only servers for that object) would be capable of sending and receiving multicast messages for that group. It further follows that any new servers created for the object would also be capable of generating the same multicast identifier, and so they too could participate.

Unfortunately, while seemingly simple, this solution does not fit in well with the basic organisation of the system. One problem is caused by the client/server relationship and the use of remote procedure calls. Recall that each server is effectively in a loop, receiving remote procedure calls, obeying them, and returning the results to the client. Furthermore, the code that dispatches the call to the correct operation has been generated automatically by the stub generator from the interface to the object. The use of multicast communications then causes problems, since the server is either expecting a remote procedure call, or is obeying one already. In either case the arrival of a multicast call is unexpected and additionally the dispatch code does not know how to handle it.

There is too an equally difficult problem regarding precisely what information to exchange. What effectively has to happen is that the server attempting to gather the lock information must recreate the entire list of *Lock* objects within itself. Thus, all of the other servers must transmit the locks they are currently holding in a form that allows them to be recreated at the receiver. This is difficult to achieve without breaking the encapsulation properties of the *Lock* objects.

Neither of these problems is unsolvable, however, there are simpler solutions to the problem, therefore, this approach is considered no further.

Using the Arjuna Object Store

Fortunately a much simpler and more universally applicable way of overcoming the problem is possible which requires making use of the Arjuna object store itself to transfer information between the individual servers that are collectively managing an object. This approach is only open because in Arjuna locks are simply regarded as objects by the system and are thus equally eligible to be stored in the object store.

The implementation requires modifications to both the *Lock* class and the concurrency controller class *LockCC*. Recall that to ensure mutual exclusion the *LockCC* class (Figure 4-12) maintains a semaphore that is acquired and released by each operation (for example *setlock* (Figure 4-8)) before the internal state of the concurrency controller is manipulated. Thus in addition to acquiring and releasing the semaphore, each operation must ensure that the current concurrency control state is retrieved from (and stored in, respectively) the object store. To handle this, each call of the *P* semaphore operation is simply replaced with a call to the new private operation *loadstate*. Similarly, each call of the semaphore *V* operation is replaced by a call to the operation *unloadstate*.

All the *loadstate* operation is required to do is acquire the semaphore (to maintain the property of mutual exclusion) and then cause the concurrency controller state to be loaded from the object store (this is shown as Figure 5-9). This operation first obtains the image of the concurrency control information for the object from the object store and then proceeds to rebuild the internal list of lock objects based upon the information it finds in the retrieved image.

```
//
//   Lock and load the concurrency control state. First we grab the
//   semaphore to ensure exclusive access and then we build the held
//   lock list by retrieving the locks from the object store.
//

void LockCC::loadstate ()
{
      int count;                          // retrieve this many locks
      Uid* u;
      Lock* current;                      // retrieving this lock
      Image* I;                           // image retrieved from store

      mutex->P();                         // grab semaphore
      if ((I = lock_store->unload(get_Uid(), LockCC::type())) != Null)
      {                                   // pick apart the image
            I->unpack(&count);            // how many locks in store
            for ( int i = 0; i < count; i++)
            {                             // retrieve and rebuild lock
                                          // information
                  u = new Uid();          // lock unique id
                  u->unpack(I);
                  current = new Lock(u); // create empty lock
                  current->unpack(I);          // unload image into it
                  locks_held.insert(current);  // Then add to lock list
            }
      }
}
```

Figure 5-9: The *loadstate* operation of *LockCC*

The encapsulation property of objects requires that individual locks must rebuild themselves, since only the implementor of the *Lock* class knows the internal state of a *Lock* object, only he has sufficient knowledge to know what parts of that state are required to be saved in the object store such that the lock could be recreated when necessary. Thus an empty lock is created which is then made to perform this operation. The resultant lock is then simply added to the internal lock list.

As might be expected, *unloadstate* is simply the reverse of this operation, which causes the state of the concurrency controller for the object to be replaced in the object store prior to then releasing the semaphore. In this case an empty image is first created into which each lock object is made to pack itself. The

resulting image of the concurrency controller's state is then stored in the object store.

The packing and unpacking of objects requires that each type supplies both a *pack* and an *unpack* operation which can be used to cause instances of the type to pack up their state into an image, and similarly retrieve their state from a supplied image.

This is only one possible approach, in that the entire state of the concurrency controller for an object is built into a single image for storage in the object store. An alternative approach would be to store each lock in the store individually (since *Lock* is derived from *Object* this is equally possible to achieve) and then have the concurrency control state simply be a list of those individual objects, rather than the objects themselves. However, this solution imposes greater overheads than the one adopted due to the creation of the extra images, thus it has not been adopted here. As will be seen in the next chapter, a modified Arjuna system model will remove the need for moving the state around at all.

Since locks must be able to pack themselves into an image, an appropriate *pack* operation must be defined for them. Figure 5-10 illustrates just such a *pack*

```
virtual void Lock::pack ( Image* I )
{
      I->pack(isactionlock);              // pack up type,
      I->pack(current_status);            // held or retained
      I->pack(lockmode);                  // mode
      owner->pack(I);                     // and owner
}
```

Figure 5-10: The *pack* operation of *Lock*

operation. Packing of locks is simple in that the private variables of the class indicating the lock type, status and mode are directly packed, followed by the owner of the lock. Since the owner of the lock is an instance of the class *Uid*, it is

asked to pack itself, for the same reasons regarding encapsulation that have been made earlier.

With these modifications it is now possible to tolerate (at a price, particularly in performance) having multiple servers for an object at least as long as only the simple multiple reader, single writer approach to concurrency control is followed. However, as shall be seen in the next chapter, enhancing the level of concurrency recreates these problems once more.

Using Shared Memory

Another possible solution to the problem of concurrency control state is to use shared memory. One approach to using shared memory requires that all of the locks are either originally allocated in, or moved to (it does not really matter which), a region of memory that can be shared between all of the servers for an object.

For this scheme to be effective all the locks for a given object must either reside in one particular shared memory region which the concurrency controller of the object knows how to access, or alternatively, there is a single shared memory region per system which is organised in such a fashion that it is possible to identify which locks belong to which concurrency controller, since in general there will be many objects active, and hence many different concurrency controllers, each of which will need to know only those locks set by itself.

While seemingly attractive, the use of shared memory in this fashion is discounted for two reasons. Firstly its management and organisation is complex, and secondly (and in this, case more importantly) the proper support facilities for it are not currently available in the current implementation environment.

However, by adopting a slightly different approach the use of shared memory is still possible. The approach adopted is to provide a shared memory manager that has basically the same interface as the object store. Using this approach the *Image* of the concurrency control state must still be created, but instead of it then being stored in the object store, it is placed in a region of shared memory. Since access to shared memory should be faster than access to disk this approach should provide an increased level of performance.

This approach is also useful in that since the interface to the shared memory manager resembles the interface to the object store the changes to the concurrency controller are minor to implement it.

Sample performance figures for both of these approaches (object store and shared memory) are given in section 5.7.1.

5.6.2 The Problem of Server Lockout

The previous section showed how the basic problem of state management could be overcome by moving the necessary state information in and out of the object store (or shared memory) as it was needed.

This section describes another problem which may cause atomic actions to be aborted unnecessarily. Consider the diagram of Figure 5-11. This diagram illustrates the processes that will have been created if a top-level action A creates two concurrent nested actions B and C (all of which are executing at site N_1 although they need not necessarily be so) to perform some work on its behalf, each of which is manipulating the same object at some remote site N_2. According to the rules stated in section 5.5.2 since the two nested actions share a common ancestor then both will use the same remote server process (in this case S). This is where the problem occurs.

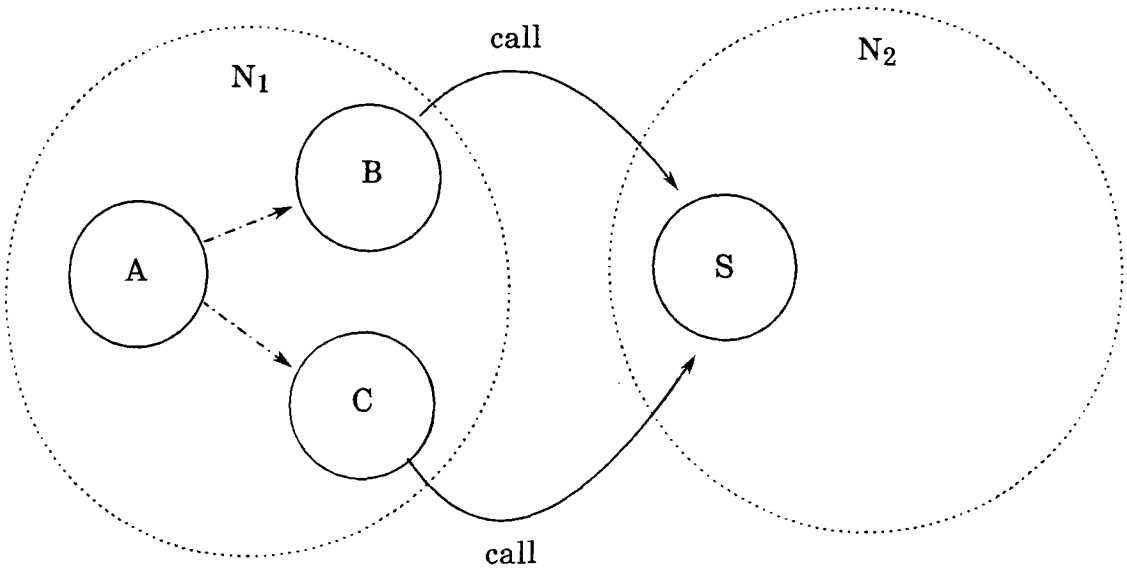

Figure 5-11: Concurrent nested action structure in *Arjuna*

There are two distinct cases here. In the first, the server S may already be executing an operation on behalf of the action B. In this case it is deaf to any incoming requests for service from the action C. Since the operation may take an arbitrary length of time to complete the RPC mechanism may incorrectly assume that the server is dead and give an exceptional return. The only safe course open to the action C is then to abort itself, which is completely unnecessary. This is, of course, a problem with the underlying RPC mechanism.

The second problem in this area relates to the action the concurrency controller for the object undertakes when it detects conflict. At present it simply causes the server to sleep for a short period before retrying the lock request. Once again this will cause the server to become deaf to other clients. Consider the following; B has set a read lock on the object and a request from C arrives that

requires a write lock. In this case, a conflict will (correctly) be detected and S will be made to sleep. In doing so, S is now deaf to any call from B that might cause the conflicting lock to be released, such that when C's request is attempted again the conflict still exists and S will sleep once more. Eventually, the timeout on the *setlock* call will decide that the write lock for C cannot be set and will give an exceptional return, which is likely to result in C being aborted. In addition, since the server appears deaf, the RPC mechanism might similarly cause B to be aborted also.

It should be stressed here that this behaviour does not cause any inconsistencies to appear in the system, rather it may simply abort actions unnecessarily. The revised architecture in the next chapter will also tackle this problem.

5.7 The Concurrency Controller in Arjuna

In chapter four, a basic concurrency control type was described, together with some modifications to allow it to work in a nested atomic action environment. Earlier parts of this chapter have illustrated the further restrictions that Arjuna placed upon the design and has outlined some means by which these can be accommodated. This section presents the current Arjuna version of the concurrency control type to illustrate one particular implementation of the concurrency controller and gives some indication as to the performance of this implementation.

Figure 5-12 shows the interface presented by *LockCC* to users. To ensure that strict two-phase locking is obeyed a user-defined type should only make use of the *setlock* operation. All of the other publically available operations (such as *propagate*) are intended to be called only by the atomic action implementation as part of commit or abort processing. However, as was noted in the previous chapter the concurrency controller is capable of setting locks (and following two-

```
class LockCC : public Object
{
      Lock_List locks_held;                        // the actual list of locks set
      Semaphore* mutex;                            // for mutual exclusion purposes
      Object_Store* lock_store;                    // repository for locks

      void loadstate ();                           // CC state loader
      void unloadstate ();                         // and unloader
      void freestate ();                           // state ditcher

      void dorelease (const Uid*, releasetype);    // actual lock releaser
      boolean lockconflict (const Lock*);          // conflict checker
      boolean isancestorof (const Lock*);          // check ancestry
                                                   // information

protected:                                         // Arjuna specific operations
      virtual void pack (Image*);
      virtual void unpack (Image*);

public:
      LockCC();
      ~LockCC();

      status setlock (Lock*, int timeout = 100); // user visible setlock
      status releaselock (const Uid* lockid);    // release one particular lock
      status releaseall (const Uid* actionid);   // release all locks for a given action
      void propagate (const Uid* actionid);      // propagate all locks to parent action

      // functions inherited from Object

      virtual TypeName type();
};
```

Figure 5-12: The *Arjuna* version of *LockCC*

phase locking rules) even if the program is not executing under the control of an atomic action. In these circumstances, it is the responsibility of the programmer to call *releaselock* explicitly to release such non-action locks. The programmer should also be aware that locks set outside of an action will conflict with those set as part of an action, since the locks will be given different owner identifiers.

Figure 5-13 shows the implementation of the *setlock* operation itself. This implementation uses the default parameter mechanism of C++ to provide a simple timeout mechanism that can be overridden at any single call as described in section 4.8 of the previous chapter.

```
//
// setlock: This is the main user visible operation. Attempts to set
// the given lock on the current object. If lock cannot be set, then
// the lock attempt is retried timeout times before giving up and
// returning an error. This gives a simple handle on deadlock.
//

status LockCC::setlock ( Lock* reqlock, int timeout )
{
      boolean conflict = TRUE;                    // assume there will be conflict
      status returnstatus = REFUSED;              // matching return status

      if (Current_Action != Null)                 // set up lock owner
            reqlock->setowner(Current_Action->get_Uid(), TRUE);
      do
      {
            loadstate();                          // recover entire state
            if ((conflict = lockconflict(reqlock)))
            {                                     // there is conflict so...
                  freestate();                    // free state
                  timeout--;                      // decrement timer
                  sleep(5);                       // wait a bit
            }
      } while ((conflict) && (timeout >= 0));
      if (!conflict)
      {     // no conflict so set lock
            locks_held.insert(reqlock);           // add to local lock list
            if (Current_Action != Null)
            {                                     // add lock record to action list
                  Current_Action->
                        add(new Lock_Record(reqlock->getowner(), this));
            }
            activate();                           // trigger object load from store
            returnstatus = GRANTED;               // lock granted successfully
      }
      unloadstate();                              // exit critical region
      return (returnstatus);
}
```

Figure 5-13: The *Arjuna* version of *setlock*

Similarly, Figure 5-14 illustrates the conflict operation *lockconflict*. This uses the capabilities of the individual lock objects to determine whether lock modes conflict and performs an ancestry check to determine if any found conflict is with one of the requesting actions ancestors. Thus it implements Moss's nested locking rules.

```
//
// lockconflict: Here we attempt to determine if the provided lock is
// in conflict with any of the existing locks. If it is we use nested
// locking rules to allow children to lock objects already locked by
// their ancestors.
//

boolean LockCC::lockconflict ( const Lock* reqlock )
{
      Lock* heldlock;
      Lock_Iterator next(locks_held);            // the iterator over locks
      boolean isconflict = FALSE;                // assume no conflict

      while ((heldlock = next()) != Null)        // get next lock
      {
            if (*heldlock != reqlock)            // check for conflict
            {
                  if (!isancestorof(heldlock)) // not quite Moss's rules
                  {
                        isconflict = TRUE;
                        break;
                  }
            }
      }
      return (isconflict);
}
```

Figure 5-14: The *Arjuna* version of *lockconflict*

The astute reader will note that this check does not follow Moss's rules exactly. If it did then a nested action would not be allowed to lock an object that had been locked by any of its ancestors. Moss regarded this situation as being a deadlock between the action and the conflicting ancestor and thus disallowed it. This interpretation was viewed as being too restrictive in a general object-oriented environment and has thus been relaxed so that children are allowed to lock objects that have been locked by their ancestors. However, care must obviously be exercised. In particular, parents must not assume that their children will not modify objects that they themselves have locked.

5.7.1 Performance

This section describes simple experiments that were carried out to determine the performance of the concurrency controller as it has been implemented in Arjuna. The performance figures given here are derived from an

untuned, experimental implementation, the primary purpose of which was to establish the feasibility of the type-inheritance approach adopted in this thesis. All of the tests were carried out on a Sun-3/160 computer that had 4Mb of main memory. The tests were carried out when the machine was lightly loaded and were executed many times to obtain an average time for each test. The Arjuna system was compiled using version 1.1 of the C++ compiler in conjunction with the standard system C compiler.

Basic Performance

Table 5-1 gives the basic performance characteristics concerning the creation and deletion of essential system objects, in this case *Locks* and *Lock_Records*. Creation of such object requires dynamic acquisition of the basic

Type	Creation Time (microseconds)	Deletion Time (microseconds)
Lock	347	232
Lock_Record	334	.352

Table 5-1: Basic system performance

storage (via the standard system memory allocator, *malloc*), followed by execution of the constructor function for the type (and all of its base types). Thus for instances of the type *Lock*, the constructors for both *Lock* and *Object* are invoked.

Performance of the Concurrency Controller

In section 5.6 of this chapter methods by which the multiple server model currently used by *Arjuna* could be tolerated were described. These methods, which required the loading and unloading of the state of the concurrency controller for an object from the object store (or a region of shared memory),

naturally have a performance penalty. In this section a simple experiment is described which attempts to quantify this penalty.

The test carried out was particularly simple. A new type was created (derived from *LockCC*), the sole operation of which simply executed a predetermined (as indicated by an argument to the type's constructor) number of calls on *setlock* to set compatible locks (that is, READ locks) and to time how long this took. This operation was then executed a number of times, and the average time each call to *setlock* took to execute calculated. The test was executed under the control of a single top-level action which was aborted each time.

This test was repeated with a version of *LockCC* that maintained all of the lock information in memory without attempting to move the locks around, a version that used the shared memory technique, and a version which utilised the object store, as described in section 5.6. The results of these tests are given below as Table 5-2 and graphically as Figure 5-15. To determine the effect that

Number of Locks	In Memory (milliseconds)	Using Shared Memory (milliseconds)	Using Object Store (milliseconds)
20	4.2	36.6	41.0
40	4.9	89.8	92.5
60	6.1	125.9	162.9
80	8.1	242.4	246.7
100	9.8	343.8	342.7

Table 5-2: Performance with action

execution of the test under the control of an atomic action was having the same tests were repeated without using an atomic action. These results are shown as Table 5-3 and Figure 5-16.

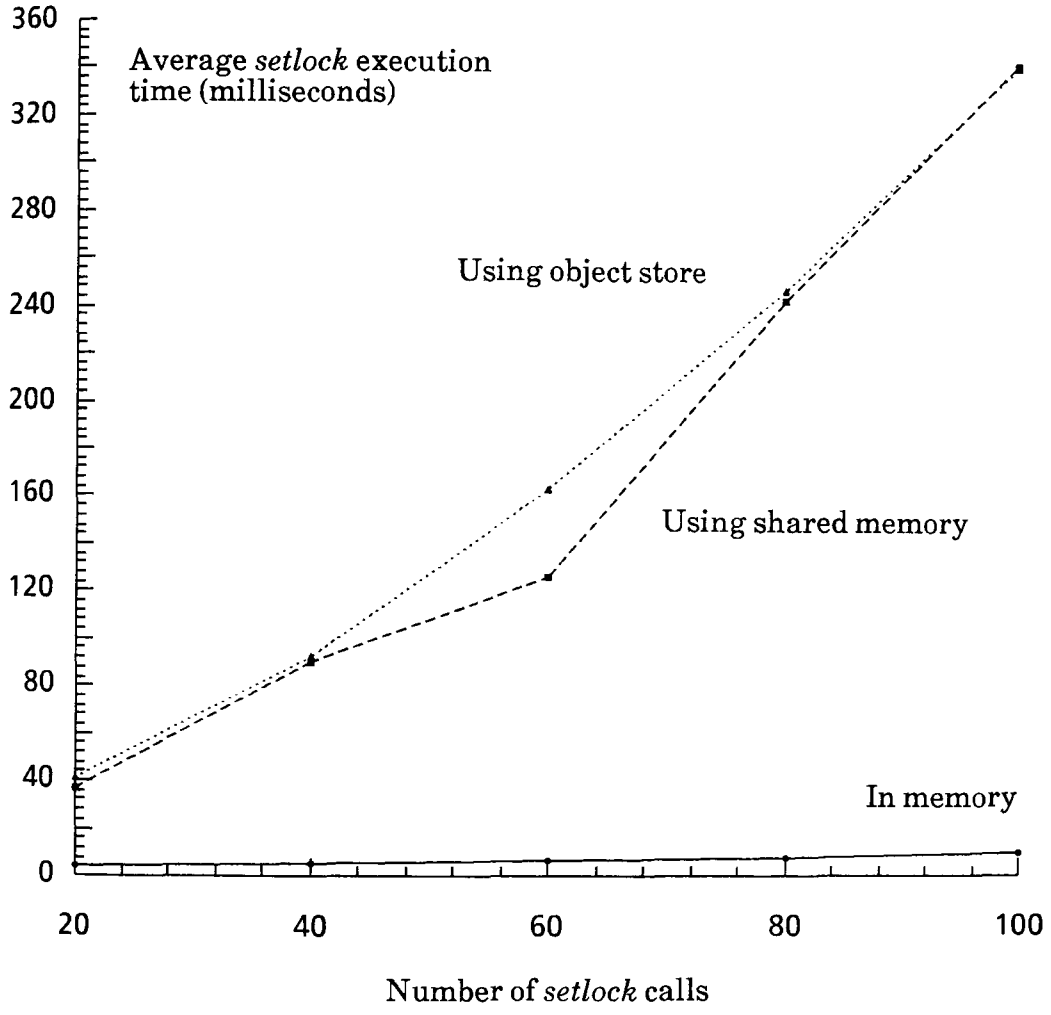

Figure 5-15: Comparison of versions of *LockCC* under action

These results betray some interesting characteristics. As expected, any attempt to cater for multiple servers causes a considerable performance penalty. However, the difference between the version of the concurrency controller using shared memory is consistently, but not significantly, faster than the version that uses the object store. This suggests that the overhead of creating and dismantling

Number of Locks	In Memory (milliseconds)	Using Shared Memory (milliseconds)	Using Object Store (milliseconds)
20	2.6	29.2	34.4
40	3.7	58.9	64.7
60	4.8	90.3	96.9
80	6.7	124.9	132.0
100	7.7	169.8	178.1

Table 5-3: Performance without action

the image of the concurrency control state is the performance bottleneck and attention in that area might prove fruitful.

The effects of the presence of an atomic action when attempting to set a lock are also illuminating. Without the presence of an action, the concurrency controller has approximately linear performance in that it takes, on average, twice as long to set fourty locks as it does to set twenty locks. In the presence of an action this linearity no longer holds, such that as the number of locks increases then the average time taken to set a lock rapidly becomes excessive, such that by the time a hundred locks have been set, the response time is approaching half a second. This deterioration in performance can be attributed to the creation of the *Lock_Records* (which has a fixed overhead), and more importantly, to the addition of these records to the atomic action management structure, which because it is behaving as a cache has to be scanned at each insertion to see if the record already exists. So as the number of records increases this scan takes longer to perform.

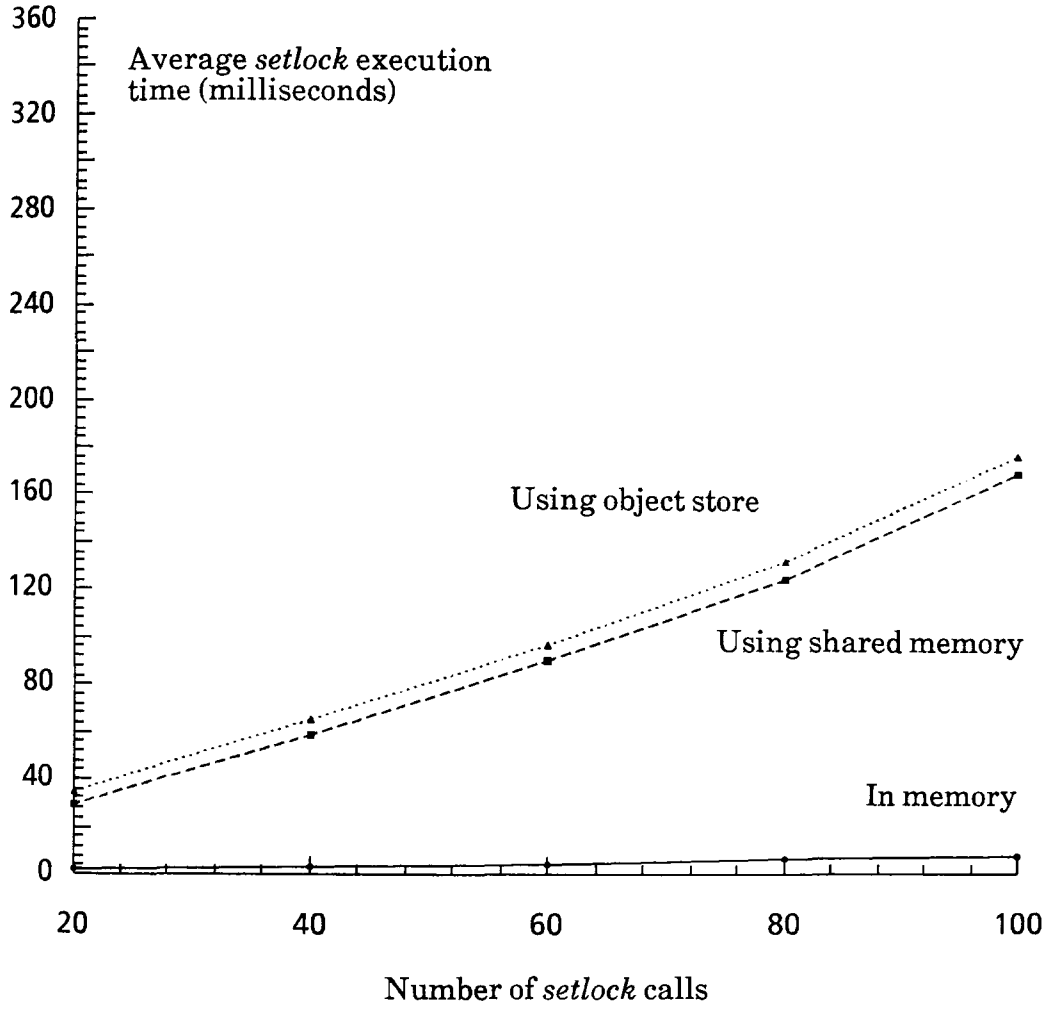

Figure 5-16: Comparison of versions of *LockCC* without action

5.8 A Complete Arjuna Example

In this section a more comprehensive example is described to outline how a simple user-defined type can be implemented. The user-defined type in question is meant to be used as part of a diary system, allowing users to note events that will take place at various times of day. The basic relationships between the objects used in this example is shown as figure 5-17. At its heart is the class *Day*

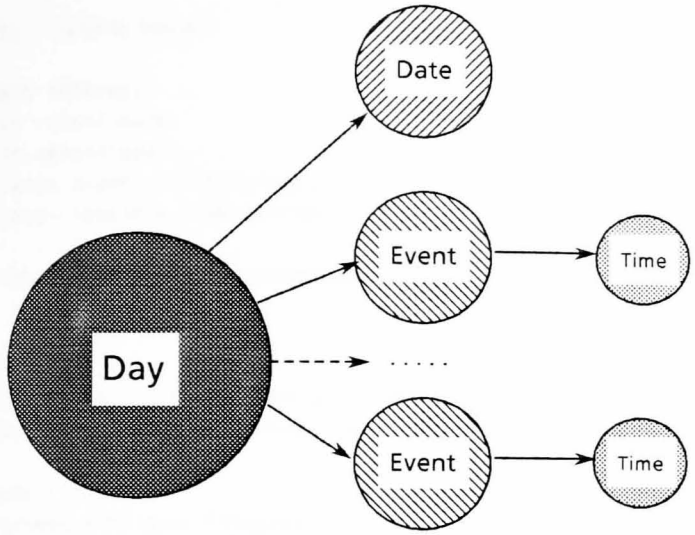

Figure 5-17: Object relationship for class *Day*

(Figure 5-18). Instances of this class represent a single day within a diary and thus have an instance variable that records the date the instance represents.

For simplicity, events are only allowed to occur at predefined timeslots throughout the day, and have durations that are finite multiples of the length of each timeslot. The granularity of each timeslot is determined by a compile time constant. Similarly, there are only a set number of timeslots per day and these occur between some starting hour and a finishing hour (say 7am to 8pm). Since some events naturally last all day (for example, birthdays) such events are flagged as *special* and are stored independently of other events. Special events appear to start at midnight and have a duration of twenty four hours.

Events are implemented as another user-defined type and are shown as Figure 5-19. Each event starts at some particular time, has a duration, and a character string that describes the actual event. Events can be created in several

```
enum eventtype { NORMAL, SPECIAL };

class Day : public LockCC
{
      Date thisday;                          // date of this calendar page
      int normalcount;                       // event counters
      int specialcount;
      Event* events [slotsperday];           // the actual events for today
      Event* specials [specialslots];        // the special events for
                                             // today
      boolean slotinuse [slotsperday];       // flag which timeslots are
                                             // currently being used

      void init ();                          // basic initialisation
      boolean set (Event*, eventtype = NORMAL); // set up an event
      boolean purge (Event*, eventtype = NORMAL); // purge an event

protected:                                   // Arjuna specific
      virtual void pack (Image*);
      virtual void unpack (Image*);

public:
      Day ();                                // create entry for today
      Day (int, int, int);                   // and for specific date
      Day (Uid*);                            // Arjuna specific
      ~Day ();

      boolean setevent (Event*);             // set normal event
      boolean setspecial (Event*);           // set special event
      boolean purgeevent (Time*);            // delete event that starts at
                                             // given time
      boolean purgespecial (Event*);         // delete special event
      boolean freeday ();                    // indicate if no events for
                                             // this day
      Event *getevent(Time*);                // return event for given time
};
```

Figure 5-18: The class *Day*

ways: either from scratch by specifying time, duration and event; by copying an existing event; by accepting a default duration, etc.

Event is derived from *LockCC* so that individual events could be locked if required, although in this particular implementation this capability is not utilised (because the operations of *Event* do not make calls on *setlock*). Instead, concurrency control occurs at a higher level (in this case at the *Day* level) when an event is set.

```
class Event : public LockCC
{
      Time starttime;                          // event start time
      int duration;                            // event duration in minutes
      char* eventstring;                       // event description

      void buildevent (Time*, int, char*);

protected:                                     // Arjuna specific functions
      virtual void pack (Image*);
      virtual void unpack (Image*);

public:
      Event (Event*);                          // build new from old
      Event (char*);                           // event that last all day
      Event (Time*, char*);                    // event at specific start time;
      Event (Time*, int, char*);               // event with specific time interval
      Event (Uid*);                            // Arjuna specific
      ~Event ();

      Time* getstarttime ();
      int getduration () { return duration; }
      char* getevent ();

      boolean operator== (Event*);             // compare two events
};
```

Figure 5-19: The class *Event*

By deriving *Event* from *LockCC* it is possible to apply the *unload_image* operation provided by *Object* to instances of *Event* to yield an image capable of being stored in the object store. This approach has been taken so that events might be given independent existence at a later stage. A simpler alternative would have been to allow access to the *pack* and *unpack* operations directly as part of the public interface to instances of *Event* as was done with instances of *Lock* earlier in this chapter.

Given these basic types then the various event type setting routines are implemented as calls on the private operation *set* which is shown as Figure 5-20. This operation simply sets a write lock on the day (by calling *setlock*), calls *modified* (provided by *Object*) to record the fact that the state of the object is about to be changed, and then updates the internal state of the day object as appropriate.

```
boolean Day::set ( Event* ev, eventtype slottype )
{
      int slotcount, startslot;
      int howlong = ev->getduration();
      Time* when = ev->getstarttime();

      setlock(new Lock(WRITE));                    // lock day
      modified();                                  // indicate modified state
      switch (slottype )                           // now to the real work...
      {
      case NORMAL:
            if (normalcount == slotsperday)
                  return FALSE;                    // no more slots left
            slotcount = howlong / minsperslot;
            if ((slotcount * minsperslot) != howlong)
                  slotcount++;
            startslot = (when->gethour() - starthour) * 2  + when->getmin() / minsperslot;
            for (int i = startslot; i < startslot + slotcount; i++)
                  if (slotinuse[i])
                        return FALSE;              // not all required slots free
                                                   // - error return
            while (slotcount-- > 0)
                  slotinuse[startslot++] = TRUE;
            events[normalcount++] = new Event(ev);
            break;
      case SPECIAL:
            if (specialcount == specialslots)
                  return FALSE;
            specials[specialcount++] = new Event(ev);
            break;
      }
      return TRUE;
}
```

Figure 5-20: The implementation of *set* for the class *Day*

Each individual instance of the *Day* type can be stored in the object store. To enable this to occur appropriate declarations are needed for the type-specific *pack* and *unpack* functions. The *pack* function of *Day* is shown as Figure 5-21. Naturally this requires individual events to be packable and each event instance is required to pack itself into the supplied image. The end result of this is that the image for a day contains its date, event counters, and all of the events for that date. This entire image is then stored in the object store. It would also have been possible to store events in the object store individually, instead of collecting them all into a single image but that has not been done here.

```
virtual void Day::pack ( Image* I )
{
      int i;

      thisday.pack(I);                        // get date to pack itself
      I->pack(normalcount);                   // pack up event counts
      I->pack(specialcount);
      for (i = 0; i < normalcount; i++)       // now get each event to pack
            events[i]->unload_image(I);       // itself into the supplied image
      for (i = 0; i < specialcount; i++)
            specials[i]->unload_image(I);
      for (i = 0; i < slotsperday; i++)
            I->pack(slotinuse[i]);
}
```

Figure 5-21: The implementation of *pack* for the class *Day*

Finally, Figure 5-22 shows a simple test program that uses the *Day* and *Event* types. This example reveals the ease by which the Arjuna system can be

```
#include "Day.h"
#include <Action/Action.h>
#include <stream.h>

main (int argc, char **argv)
{
      Day Today;                                  // diary page for today
      Day Xmas(25,12,88);                         // and one for christmas
      Time* start = new Time(10, 0);
      Action A;                                   // execute under control of action

      A.Begin_Action();                           // start action
      if (Today.setevent(new Event(start, 30, "Project Meeting")))
      {
            if (Today.setevent
              (new Event(new Time(10,30), 15, "Another Project Meeting")))
            {
                  cout << "Successfully set both events!";
            }
      }
      Xmas.setspecial(new Event("It's Christmas Day!"));
      A.Commit_Action();
}
```

Figure 5-22: A simple test for *Day* and *Event*

used. By simply deriving the class *Day* from *LockCC*, and adding simple calls to the operation *setlock*, instances of the class *Day* have been made concurrency controlled. Declaration of an appropriate *pack* operation, and a call on the operation *modified*, has further made instances of *Day* recoverable (in conjunction

with the use of an atomic action as illustrated in Figure 5-22) and capable of being placed in the object store. Of course similar small changes have to be made to the class *Event* but these changes have not been shown here. By these changes, a user-defined type that was not designed with concurrency control in mind originally has had it added simply and correctly.

5.9 Summary

In this chapter we have considered how the concurrency control type designed in the previous chapter could be implemented in a real system, in this particular instance *Arjuna*.

Arjuna is novel in treating all major system entities as objects and thus applying the object-oriented paradigm in a uniform fashion. Unlike other object-oriented systems such as Argus [*Liskov 88*], Clouds [*Dasgupta et al. 85*], TABS [*Spector et al. 85*] and Camelot [*Spector 87*] the system has been implemented completely using only facilities available in the implementation language and the host operating system.

As has been shown, the simple system model employed in Arjuna, particularly with regard to server management has caused some problems for the implementation of the concurrency control type, however techniques by which these problems can be overcome have been developed, implemented and their performance measured. The use of these techniques was made possible because of the uniformity by which the object-oriented paradigm has been applied in Arjuna. While the approach adopted in overcoming the problems may not be the most efficient (consider loading and unloading locks to and from the object store or shared memory for example) it does nonetheless work.

As was indicated by the performance figures in section 5.7 the performance of the concurrency controller can be significantly enhanced if it can be ensured that only a single server exists for an object. In the next chapter a revised system model for Arjuna will be described that achieves this aim. In addition, the design of some other concurrency controllers will be discussed, all of which make use of the basic approach of type inheritance that is advocated by this thesis.

Chapter 6
Alternative Approaches to Concurrency

The previous chapters have claimed that by designing and implementing concurrency control on a per-object basis there has been a gain in flexibility. In particular it has been stated that type-inheritance has allowed a flexibility not possible in other object-oriented systems which have broadly the same design goals. This chapter aims to justify this claim further.

First the chapter shows how it is possible to devise a type-specific locking scheme for the object-oriented environment under consideration in this thesis in a simple manner and indicates the requirements that this places upon other parts of the system. It then considers some of the other methods of concurrency control that have been examined in chapter two, including specifying levels of object granularity and multi-version approaches, and indicates how these approaches might be handled under the object-oriented environment using type inheritance.

Finally the chapter considers how an implementation of an optimistic style of concurrency control could be achieved in the object-oriented environment of this thesis. This strategy, based on work by Herlihy [*Herlihy 86*] requires a different recovery scheme to the normal state-based mechanism usually used to implement the failure atomicity property of atomic actions, but the discussion shows that the type-inheritance approach is flexible enough to cope with the requirements of an optimistic concurrency control technique providing the required underlying recovery mechanisms can also be provided.

6.1 Type-Specific Locking

Many other researchers have pointed out that by taking into account the semantics of the operations available upon an type, then an increased level of concurrency can be supported [*Schwarz and Spector 82, Allchin 83, Garcia-Molina 83, Schwarz 84, Weihl 84*]. The term type-specific is, in this context, somewhat misleading since it does not mean allowing user-defined types to follow any arbitrary locking policy, but instead means that the standard two-phase locking rules are still being followed but with modes other than simple read and write.

In [*Schwarz and Spector 82*] type-specific lock conflict tables are defined based upon both the operation to be performed and all of its formal parameters. An example of such a table is shown as Figure 6-1.

		Held Lock Mode		
		DirModify(σ)	DirLookup(σ)	DirDump
Requested Mode	DirModify(σ)	n	n	n
	DirModify(α)	y	y	n
	DirLookup(σ)	n	y	y
	DirLookup(α)	y	y	y
	DirDump	n	y	y

Figure 6-1: Compatibility matrix for directories

In this example locks may of three distinct types:

- DirModify(σ). This indicates that an action has inserted or deleted an entry with a key string of σ.
- DirLookup(σ). This indicates that an action has attempted to observe the entry with a key string of σ.
- DirDump. This indicates that an action has performed a dump of the entire directory.

Because this conflict table makes use of the parameter string σ, in addition to the actual mode of the lock, it allows actions to proceed concurrently that would not otherwise have been allowed had the operations been classed as simple reads and writes. For example, it allows concurrent writes to the directory providing such writes manipulate different directory entries. Locks obeying this particular protocol are easy to implement in Clouds [*Dasgupta et al. 85*] as was shown in section 4.3.1 in the previous chapter. This section shows how the basic technique of using type inheritance provides a simple method of achieving the same effect by building on the design of the *Lock* type of chapter four.

First, a new type of lock (call it a *TypeLock*) is derived from the basic *Lock* type. Second, the conflict check for this new type of lock is then implemented. These steps are illustrated by Figures 6-2 and Figure 6-3.

In order to show that this conflict operation is correct and does follow the conflict matrix shown earlier, consider the case where one action has already acquired a *DirModify*(α) lock (that is, an instance of *TypeLock* has been created that has α as the value of the instance variable *Id*), and a second action is also attempting to set a *DirModify*(σ) lock.

```
class TypeLock: public Lock
{
      SomeType Id;                          // Something to identify the lock
      ...                                   // e.g. the string σ or α
      virtual boolean operator!= (Lock*);

public:
      TypeLock (SomeType, modetype);
      ~TypeLock ();
      ...
      SomeType GetId ();                    // operation to access extra
      ...                                   // information
}
```

Figure 6-2: The *TypeLock* class

```
boolean TypeLock::operator!= (Lock* otherlock);
{
      if (otherlock->getowner() != owner)   // if different owners
      switch (lockmode)
      {
            case DirLookup:                 // holding DirLookup
                  if ((otherlock->getlockmode() == DirModify) &&
                        (Id == ((TypeLock*)otherlock->getId())))
                              return TRUE;
                  break;
            case DirModify:                 // holding DirModify
                  if (otherlock->getlockmode() == DirDump)
                        return TRUE;
                  if (Id == ((TypeLock*)otherlock->getId()))
                        return TRUE;
                  break;
            case DirDump:
                  if (otherlock->getlockmode() == DirModify)
                        return TRUE;
                  break;
      }
      return FALSE;
}
```

Figure 6-3: The *TypeLock* Conflict Algorithm

In this case, the comparison will proceed as follows:

- The two instances of *TypeLock* will belong to different atomic actions, therefore the full conflict check will be performed.

- Since the existing lock has a mode of *DirModify* , the conflict operation first checks to see if the new lock has a mode of *DirDump* (this corresponds to the entry in the bottom row of the first column in Figure 6-1).

- Having passed this first check the *Id* fields of the two locks are compared. Since they are different (one is α, the other is σ) then the two locks are not considered to conflict and the result is thus *False*. This is precisely the behaviour dictated by the conflict matrix of Figure 6-1.

Similar arguments can be followed for all of the other entries in this particular conflict matrix. Thus the new type-specific lock implemented by the type *TypeLock* does indeed obey the locking rules of Figure 6-1 correctly.

One characteristic of this particular implementation is that the conflict check still appears to take a *Lock* object pointer as its parameter despite the fact that when it is actually called the parameter will actually be a pointer to a *TypeLock*. This is a quirk of C++ in that once defined, a virtual function declaration cannot change. Thus having been defined to take *Lock* object pointers in the base class *Lock*, the definition must remain the same in the *TypeLock* class definition. In reality this causes no problems since one type of lock is derived from the other and thus by the rules of object-oriented programming a pointer to a *TypeLock* may be passed to a routine expecting a pointer to a *Lock*. Furthermore, the invoked operation assumes in advance that its parameter type is really a pointer to a *TypeLock*, not a pointer to a *Lock*, and acts accordingly. Making this assumption may be dangerous if the programmer accidentally mixes the use of *Locks* and *TypeLocks*. This can be overcome by the inclusion of an additional check in the conflict operator for *TypeLock* that the parameter is of the required type. The result of the comparison in this situation is programmer defined.

In addition this particular implementation bears out the point that was made in the previous chapter with regards to lock modes. Recall that in that chapter the justification for calling an *activate* routine after setting every lock was because it was not possible to guarantee the ability to detect when write-type locks were being set. This example illustrates precisely this point. Here the lock modes are not simply read and write, and although they can easily be classified as examine or modify types as was suggested, this classification can only be done with additional information from the user.

This approach is flexible in that the additional information regarding the extra lock modes and the extra semantic information (in this case an identifying value) are all entirely user specified. Furthermore, this extra information is not limited to a single piece of information; rather it can be as many pieces as deemed required. For example, the compatibility function might be constructed such that locks are compatible if both the identifier in conjunction with some other value obey some condition (or set of conditions). This is in contrast with the Clouds approach, where locks can only be parameterised with one additional value. In fact the lock type developed here (by appropriate modification of the conflict operation) will perform precisely like any Clouds lock.

6.1.1 Some Problems

From the point of view of the concurrency control type designed in chapter four the scheme described in the previous section is perfectly acceptable and would appear to produce serialisable executions of the individual actions. However, a blind implementation like this would fail in the context of the Arjuna [*Shrivastava et al. 88*] system due to interactions between the concurrency controller for an object and other parts of the system. The following sub-sections describe some of these problems.

Multiple Servers

Depending upon the definition of the semantics of the type then the concurrency controller for that type might allow multiple writes on instances of the type providing that such writes manipulated different portions of the state of the object in question. This is acceptable in Arjuna providing that there is only a single server for the object. However, given that in the current implementation there might be several servers for an object in existence, then the old problem of lost updates rears its head once again.

Consider such an object being managed by two distinct servers. If different parts of the object are being modified, both servers will be granted write access to the object (since the conflict rules allow for this possibility) and will proceed to modify it. The problem now arises as to what happens when the two servers attempt to commit their changes. Since the existing Arjuna system transfers entire objects to the object store the effect of one of the writes will be lost.

What is required is some way by which only the modified state is entered into the object store. This could possibly be handled by defining appropriate *pack* and *unpack* routines for the object so that only the modified portions are transferred, but this would still probably require a method of retrieving the entire object state since it is bound to be needed somewhere. Using this technique the object store would no longer contain complete versions of each object, but some base version together with a set of incremental changes to that base version, the result of applying which would yield the current object state. Using such a technique also requires changes to the basic concurrency controller over and above simply defining a new lock type. This is because the basic concurrency controller assumes that a simple call to *activate* will obtain the entire latest state of the object. By using incremental transfer this is unlikely to be the case.

Note that this problem only arises because of the possibility of the existence of multiple servers. If only a single server existed, then the entire state of the object is being maintained by that one server and the entire object can be committed as normal. Furthermore the basic concurrency controller can remain unaffected.

As a side issue it should also be can noted that using only a single server also alleviates the problem of concurrency control state as described in section 5.6 in the previous chapter, in that it would no longer be necessary to have to resort to loading and unloading the state of an object's concurrency controller to and from the object store (or shared memory) upon every interaction with the concurrency controller.

Recovery Management

Although not previously stated explicitly Arjuna currently uses a state-based recovery scheme. That is, whenever an object is first modified within the scope of an atomic action, a copy of the current state of the object is taken so that should the action abort, then this state can be restored. Thus the server for the object maintains the current state and modifications to the object thus take place directly on this state.

This has an effect on the level of concurrency permitted by an object. For example, consider some object that is meant to represent a counter, that is, it can have (at least) *increment* and *decrement* operations applied to it. Now there is no reason why increment and decrement operations should not be allowed to proceed in parallel with each other (with the proviso that suitable short-term mutual exclusion is also employed to prevent corruption), since the order in which two increments, or an increment and a decrement are executed should be irrelevant. Thus a type-specific lock that had a conflict operation defined like that of Figure 6-4 might be constructed. In this function, increments and decrements only

```
boolean IncLock::operator!= (Lock* otherlock);
{
      if (otherlock->getowner() != owner)
      switch (lockmode)
      {
            case Increment:
                  if (otherlock->getlockmode() == Read)
                              return TRUE;
                  break;
            case Decrement:
                  if (otherlock->getlockmode() == Read)
                        return TRUE;
                  break;
            case Read:
                  if (otherlock->getlockmode() != Read)
                        return TRUE;
                  break;
      }
      return FALSE;
}
```

Figure 6-4: The *IncLock* conflict algorithm

conflict with reads, thus the two should be allowed to proceed in parallel.

Unfortunately this has a disastrous interaction with the recovery system (it will be assumed for now that only a single server is managing the object). Consider some object X that initially has the value 5. If two concurrent actions A and B both attempt *increment* operations it would be expected that providing both actions commit the result would be that X has the value 7. Which is indeed the result with only a single server.

However, consider the following sequence of events. Action A sets an *increment* lock on X and changes the value to 6, in doing so it records the old value as 5. Similarly, action B sets an *increment* lock and sets the value of X to 7, recording the old value of 6 (since there is a single server and the concurrency control has allowed the two actions simultaneous access to the object). B then commits producing 7 as the final value for X, while A aborts, and thus restores the prior value of X to what it believes it should be, in this case 5!

The problem here is caused by the fact that although the concurrency control method commutes, the recovery method does not. One possible way to avoid this is to use an alternative recovery method based on *intentions-lists*. Using this approach changes to an object are not actually applied until the action that made them commits. Thus the individual increments for A and B would only be applied when those actions committed. Naturally each action needs to be able to see the effects it has performed, leading to the notion of a *view* of an object. Each action's view is simply the effects of any changes it has made applied to the last committed version of the object. This type of recovery approach is the basis of Argus [*Liskov 88*].

Alternatively, rather than simply re-instate some past object state, the recovery system invoke some specific compensating operation. Obviously, such compensating operations are highly dependent upon the semantics of the operations that have been performed by an action.

It is interesting to note that Allchin [*Allchin 83*] also has this problem since he uses a state-based recovery scheme as well. However, he overcomes it in an interesting fashion by allowing objects to be notified of when actions start, commit and abort, and thus objects are able to override the default recovery mechanism and indicate what result should be returned whenever they detect these events.

It must be stressed here that these problems are caused by the interaction of the concurrency controller of an object and the underlying recovery system and system execution model, not by any inherent problems with the concurrency control design itself. Fortunately, the recovery system of Arjuna is flexible enough to allow the approaches that have been suggested here to be followed, given sufficient implementation effort by the user.

Despite these problems a simple directory type obeying the conflict rules of Figure 6-1 was implemented using the type-specific lock type shown as Figures 6-2 and 6-3. This implementation, when executed in a carefully constrained environment (that is, only a single server) did appear to function correctly, further supporting the basic ideas expounded in this thesis.

6.2 Multiple Levels of Granularity

As was pointed out in chapter two, increased concurrency can also be obtained by changing the granularity at which the concurrency control is applied. Thus, for example, if the simple file example introduced in chapter four which applied locks at the file level is reconsidered, there could obviously be an increase in the level of concurrency if locks were applied at (say) the level of a page.

Figure 6-5 shows how just such a file type could be implemented using the

```
class File: public LockCC
{
      int page_count;
      Page** pages;
      int current_posn;
      ...
      virtual void pack (Image*);
      virtual void unpack (Image*);

public:
      File (char*);
      ~File ();

      int read (char*, int);
      int write (char*, int);
      int lseek (int);
      ...
};
```

```
class Page: public LockCC
{
      char buffer[PAGESIZE];
      int size;

      ...
      virtual void pack (Image*);
      virtual void unpack (Image*);

public:
      Page (Uid*);
      Page ();
      ~Page ();

      int read_page (char*, int, int);
      int write_page (char*, int, int);
      ...

};
```

Figure 6-5: The *File* and *Page* classes

type-inheritance technique. Firstly, the two basic classes involved; *File* and *Page* are described. The class *Page* only provides two operations, *read_page* and *write_page*, which use the basic (read and write) locking mechanism to set appropriate page level locks. The class *File* uses instances of the *Page* class to

represent individual pages within a file. Both the *File* object and all of its pages are held in the object store as separate objects.

When a *File* object is accessed, only it is activated; the actual pages are activated (and locked) as and when they are really needed. It will be assumed that the state of the file as stored in the object store essentially consists of a list of the unique identifiers of the enclosed pages. Thus in order to access any particular page the code for the read or write operations of *File* only has to check to see if the appropriate page has been activated, and activate it if it has not. Once a page has been activated it can only be manipulated through the *read_page* and *write_page* operations, each of which sets an appropriate page level lock (either read or write) before proceeding.

Using this approach, then providing that the size of the file (that is the number of pages it consists of) does not change, then it is not necessary to set any type of locks on the *File* object other than simple read locks. If the size does change then a write lock must be set on the file to prevent two actions both attempting to extend the file for example.

Once again this example was implemented under the current Arjuna system, and since it follows the traditional multiple reader, single writer policy for both *File* and *Page* objects functions correctly even in the presence of multiple servers. This implementation is, of course, quite susceptible to deadlock particularly if independent actions attempt to modify pages of the file that the other currently holds a read lock on.

It would, of course, have been possible to follow the description of multi-granularity locking described in chapter two more closely and set *intention* locks at the file level in a very simple and obvious fashion by defining a new lock type *ILock*, derived of course from *Lock*, and defining an appropriate conflict operation

for it. However, since it is not strictly necessary in this case that is left as an exercise for the reader.

It could be argued that the problems described in the previous section regarding modifying different parts of an object's state are caused by the fact that the object has been incorrectly designed. For example, rather than treat the directory as a monolithic whole, it should have been structured differently as a *container* object as has been done here for the *File* object.

Using this approach the directory object simply contains references to other objects that are actually contained within it (call them *direntry* objects), with each such object being totally independent - in particular responsible for its own concurrency control and recovery.

Such a directory object could be designed and implemented in the Arjuna system that obeys these rules; unfortunately it does not circumvent the problem of the previous section. Consider an insert of some directory entry, if it does not already exist then it is inserted. However, this insertion causes creation of a new *direntry* object, the unique identifier of which must be recorded as part of the containing directory. Hence the state of the directory object itself is being changed and the directory must therefore be locked.

Ideally, two inserts should be able to proceed concurrently if they are inserting different entries (again assuming short-term mutual exclusion is employed while critical data structures are updated), but since each insert modifies the directory state the original problem has reappeared. It is possible to define a type-specific lock function that allows the concurrent writes, but if there is more than one server for the object the lost update problem is back once again.

This observation leads to the unhappy conclusion that although the concurrency control scheme using type inheritance is extremely flexible, its implementation in Arjuna is compromised by the underlying system model, in particular the fact that an object may be managed by more than one server at any instance in time.

6.3 A Revised Arjuna System Model

Throughout this chapter, and as part of chapter five, whenever an attempt is made to increase the level of concurrency supported by an object by allowing multiple writes upon it the Arjuna system model conspired to thwart the attempt. This was due to the fact that multiple servers could exist for an object. In this section, this model is revised and it is explained how the problems outlined previously are thus solved.

The fundamental change that is made is to insist that any object may only have a single server associated with it whenever it is active. In actual fact this is not really a major change, since Arjuna currently only creates new servers when non-related actions attempt to access the object. All this change requires is that if a server already exists for an object, then it serves all clients, not some constrained subset of them.

However, this is not the whole story. One of the original design decisions of Rajdoot, the remote procedure call system upon which Arjuna is based, was that there would be multiple servers in order to ensure that a server could not become deaf to a call. By insisting that there is only a single server per object this problem has been re-introduced, since while the server is obeying one call, it is not listening for others. In addition, in order to be efficient, Rajdoot uses datagrams not virtual circuits for its client/server communication. This has the effect that if the server is busy, the call is simply lost. Of course, Rajdoot performs a few retries on behalf of the client, but it is still possible for the client to

incorrectly conclude that the server is dead, when it was in actual fact simply busy every time the client made a call. As was pointed out in section 5.6.2 of the previous chapter, this is unlikely to cause inconsistencies in the system, but is likely to cause actions to be aborted unnecessarily.

In order to overcome this problem parallelism must be introduced into the actual server itself; that is the server must become *multi-threaded.* A multi-threaded server behaves in a fashion similar to an Argus guardian. That is, whenever a new remote procedure call is received, a new lightweight process (or *thread*) is allocated to deal with the request, and the server then listens again for further requests. Obviously by following such a strategy, the server will never appear busy so long as it is capable of creating a new thread for each incoming call.

All threads created by a server share the same address space, thus all threads will see the same object state (that is, the current state), and furthermore, all of the concurrency control information will be available without resorting to the trick of loading and unloading locks to and from the object store. Naturally, since the threads appear to run in parallel with each other interference between them is possible, however, standard mutual exclusion techniques such as the use of semaphores provide an adequate solution to this problem.

Although UNIX (upon which Arjuna is currently hosted) does not currently support threads directly, they can be simulated. In addition both Amoeba [*Tanenbaum and Mullender 81*] and Mach [*Jones and Rashid 86*] directly support threads, with the latter system intended to be BSD4.3 UNIX compatible. Thus the adoption of this approach should be relatively simple.

Multi-threading also cures the problem outlined in the previous chapter with regards to the concurrency controller for an object sleeping when conflict is detected, since now only a single thread will be affected, not the entire server, as was the case.

6.4 Multi-Version Approaches

So far this chapter has managed to adapt the basic concurrency controller of chapter four to suit a variety of cases without making changes to anything other than the basic *Lock* type itself (by deriving new types of lock from it) and defining appropriate conflict relations upon such locks. However, in order to accommodate multiple versions of objects, some changes to the underlying concurrency control type itself must be made. This is not particularly surprising since the concurrency controller was designed to support the two-phase locking approach to concurrency control. In order to support the multi-version concurrency control technique a different basic concurrency controller is required.

However, before proceeding it is necessary to ask the question: what does multi-version concurrency control mean in a nested action environment? Reed [*Reed 78, Reed 83*] has already tackled this question for timestamp-based approaches where object versions have distinct lifetimes and the timestamps of nested actions are designed to be within the timestamps of their parent. Lock-based concurrency control with versions is, however, a different matter. Recall that in chapter two a non-nested, two version concurrency controller based upon the use of *certify* locks was described. The question is, can this technique be extended to a nested environment?

The problem is really one of version visibility. In the non-nested case only one new version existed and was only visible to its creator. Other actions were only allowed to read the previous version; if they wished to create a new version they had to wait until there was only one single committed version. Stearns and

Rosenkrantz [*Stearns and Rosenkrantz 81*] describe this as concurrency control using *before values* and described it in the context of distributed databases. What is required is some way of generalising this to the more general nested atomic action environment.

One possibility is to regard nested action commitment as not producing a new *globally* visible version of an object, since until the top-level action commits, the new version is still only tentative. Thus the full two version protocol need only be adopted at the commitment of the top-level action. In other words *certify* locks can only be set by top-level actions. Using this approach means that all actions, whether nested or not, that do not belong in the universe of the writer of an object, can only set read locks and thus are permitted to read the previous version of the object.

Consider the action hierarchy of Figure 6-6. If any of the actions A, B, or C

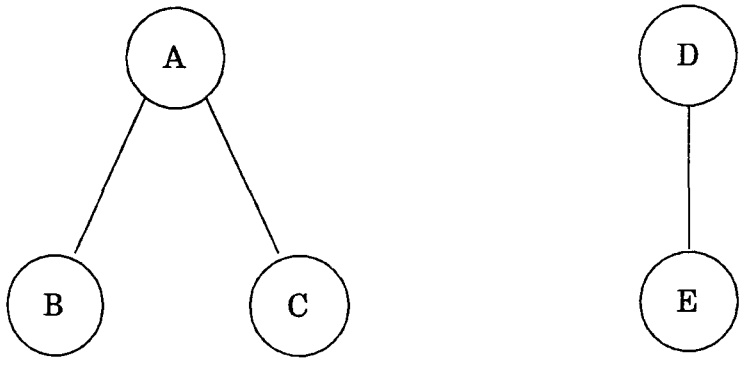

Figure 6-6: An example action hierarchy

acquires a write lock on some object X, then the action D and E will only be permitted to acquire read locks on the previous version of X until the top-level action A commits.

The problem then arises as to the visibility of this new version as far as the action tree rooted at A is concerned. Assume that action B has acquired a write lock on the object and is producing a new version. In this case, action C cannot be granted a write lock (since it conflicts with that currently held by B), but should it be granted a read lock and gain access to the previous version of the object? If the full protocol was adopted at all levels then the answer would have to be yes. However, care must be taken here in that the protocol cannot simply be followed blindly since the certification process blocks until all reads on the previous version terminated. In a nested environment this does not make sense for nested action commits since a real certified version of the object is not being produced at this point.

Essentially what is required is another type of lock (call it a *nested certify* lock) that has the property that it does not conflict with read locks held by *external* actions (for example D and E in Figure 6-6) so that the certification process does not have to wait for such actions, but it must conflict with internal actions (for example C), otherwise consistency might be compromised.

To show this, consider what could happen if B was actively creating a new version of some object X, and C was allowed access to the prior version of X. When B attempts to commit it sets certify locks on X, which if they did not conflict with C's read lock would succeed. B would then commit passing its locks and the new object version to A. The problem now is what to do about C: if it simply commits all is well, but if it attempts to convert its lock on X to a write lock (which it would be able to do) then the effect is as if both B and C had read the old version of X and both written new versions of X - a clearly unserialisable execution.

This situation could be detected by noting that the version read by a child is no longer the same as that known to its parent, but this complicates the implementation greatly with probably no additional advantages.

In addition the Arjuna system environment poses certain problems with version management. In particular since the object will be maintained by a single server, that server must always know which version of the object it is supposed to be using at each operation invocation. For example, say B was in the process of creating a new version when C attempted to set a read lock. By the standard two version rules this should be allowed with C gaining access to the previous version. However, since one server is involved (the actions are related) on every operation the server must be able to determine which version is current as far as the invoking action is concerned in order to return the correct response. Although once again not strictly a concurrency control problem (it is actually a problem of version management) it is easier to take a simpler approach that avoids these problems.

Thus the normal nested locking rules are obeyed as far as nested actions are concerned and the multi-version rules only apply at the top-level. This means that internal nested actions consistently see the latest version of the object, while external actions see the previous version.

It is interesting to note that the current Arjuna environment of multiple servers when actions are not related fits this scenario perfectly. Only one action tree can modify the object (and since the actions are related there is a single server handling this correctly), while other servers can freely be allowed to execute (in read mode) and are provided with the old state of the object. Given the disenchantment about multiple servers earlier this is an amusing outcome.

6.5 Optimistic Approaches

In many respects optimistic concurrency control poses similar problems to multi-version concurrency control in a nested environment. Recall that optimistic protocols are based upon the idea that it is easier to apologise after the event than to ask permission before it. Thus in an optimistic environment actions

execute completely without synchronisation and then immediately prior to commitment attempt to determine if any conflict has occurred due to the concurrent execution.

This process, called *validation*, is generally assumed to be cheaper than approaches based upon preventing conflict providing that validation succeeds sufficiently often. The serial validation scheme described in chapter two had the disadvantage that it was designed for a centralised database and required the concurrency controller to gather information about the read and write sets of actions and maintain this information for an arbitrary period of time, as such it is unsuitable for the object oriented environment considered in the thesis.

Recently, however, Herlihy [*Herlihy 86*] has proposed a pair of optimistic protocols suitable for object-oriented systems. In his protocols each object has associated with it a *serial-dependency* relation that allows conflicts between pairs of events to be ascertained at validation time. Each event is a pair consisting of an operation invocation and its corresponding response. This approach is comparable with pessimistic lock-based schemes in that while lock-based schemes use conflict to introduce delays, his optimistic protocols use conflict to determine if validation can be successfully completed.

Since the validation process is performed when an action attempts to commit it can validate more concurrent executions than might have been possible using a pessimistic approach because of the additional information available. In particular, under locking, a lock is acquired before an operation is performed, thus conflict is often defined in terms of operation invocations only. In contrast optimistic schemes require validation after the results of an operation are known, so conflicts can be defined between complete events.

Herlihy's protocols have obviously been influenced by the work of Weihl [*Weihl 84*] and Argus and rely on the underlying recovery system being based upon intentions-lists. This is because the intentions lists are applied in the order in which actions commit and also because the events specified can have semantics similar to those outlined for the counter object described in the previous section, for which it was shown that state-based recovery was inappropriate.

Herlihy defines two distinct protocols, called *forward* and *backward* validation. Each derives its name from the method by which it selects the actions it might be in conflict with.

Forward validation ensures that when an action commits it will not invalidate any other currently active action. Backward validation ensures that when an action is validated its execution has not been invalidated by the commitment of other actions that started after it did. This latter case is essentially the same as that defined by Kung and Robinson [*Kung and Robinson 81*].

One advantage of these protocols is that they may be used on a per object basis and freely mixed with certain other pessimistic protocols such as two-phase locking.

6.5.1 Optimism and Nesting

The same question that could be asked about multi-version protocols can also be asked about optimistic protocols, that is, how do they apply to a nested atomic action environment? Once again there are problems with respect to version visibility and also with validation. For example, consider the action hierarchy of Figure 6-6 once again. If actions A and D were using an optimistic protocol to update some object X then provided that both committed, everything would be correct. Similarly if either aborted there would be no problems.

However, consider now what happens if B and D are optimistically updating X. In this case when B attempts (and succeeds at) its validation the version of X it produced cannot be made visible as would normally be the case, since A might abort and thus undo the effects of B on X. Instead what must happen is that A must inherit all the changes made by B and the entire validation process must be repeated when A finally commits.

The implications of this are that the validation process in effect gets repeated several times (probably taking longer each time) depending upon the depth of nesting employed. It might be argued that this is wasteful and it would be better to only perform the validation on top-level actions. The problem with this approach is that such validation might fail due to a conflict that occurred in one of the children of the top-level actions that had executed a long time ago, and which would have been aborted then had the validation been performed at that time. The potential advantage of detecting the conflict early thus justifies the repeated validation.

As an example, assume that B was not properly validated at the time it committed, yet was in conflict with D. In this case A can never be validated if D commits before it, and if the action tree rooted on C takes a long time to complete (minutes or hours, as opposed to seconds) then the amount of wasted work could be enormous.

6.5.2 Implementing an Optimistic Policy

This section will outline how one of Herlihy's optimistic protocols could be implemented in the Arjuna environment. In order to do this several assumptions must be made, in particular it will be assumed that the underlying recovery mechanism is based upon intentions lists and that there is some kind of view

mechanism that allows an object to determine its current state based upon applying its updates to the last committed state.

Backward Validation

The protocol described is that of *backward* validation. That is an attempt is made to determine if execution of the validating action has been compromised by the commitment of other actions that have committed since the validating action started. In order to handle this, each object keeps a note of *Last(e)*, the most recent commit timestamp for an action that executed the event *e*. In addition, for each active action *A*, each object maintains *First(A,e)*, the logical time when the action *A* first executed the event *e*.

Objects can only validate an action *A* if $Last(e') < First(A,e)$ for each event e' that conflicts with each event *e* executed by *A*. This condition ensures that recently committed actions have not invalidated the execution of the validating action. In a sense this is equivalent to comparing the read and write sets of an action in the basic serial validation approach.

As with the implementation of two-phase locking events are modeled as instances of the class *Event* (Figure 6-7).

```
class Event
{
        ...
public:
        Event (EventType);
        ~Event ();
        ...
        EventType GetType ();
        virtual EventList* ConflictingEvents (Event*);
        ...
}
```

Figure 6-7: The basic *Event* class

Obviously this class is highly type specific (far more so than the basic *Lock* class) and real event types are expected to be derived from this basic type; thus little is said about its structure other than to describe the purpose of the operations *GetType* and *ConflictingEvents*.

ConflictingEvents will be used by the validation routine of the concurrency controller to generate a list of events that conflict with each event type that has been executed by the validating action. Once such a list has been generated then the concurrency controller can determine whether any of those events has been executed in a conflicting fashion by examination of the *First* and *Last* timestamps for the action and the event as described above. This routine must be *virtual* so that as new event classes are defined using the basic *Event* class, they can implement this generation appropriately.

GetType will be used by the concurrency controller to maintain its lists of events in a particular order. This should become clearer after the description of the actual concurrency controller class itself.

Similarly the actual concurrency controller is the class *OptCC* (Figure 6-8),

```
class OptCC
{
      ...
public:
      OptCC (int);
      ~OptCC ();
      ...
      void AddEvent (Event*);
      boolean validate (ActionId);
      void DoCommit (ActionId);
      void DoAbort (ActionId);
      ...
}
```

Figure 6-8: The Basic *OptCC* Class

from which the actual user-defined objects that use the optimistic approach are ultimately derived.

This class provides several basic operations:

- *AddEvent.* This operation is called by the user-defined type once some event has been performed to inform the concurrency controller for the type of the occurrence of the event. If the action performing the event is not currently known to the concurrency controller then it is noted. The event type is then determined and used to update the *First* information for the action if appropriate.

- *Validate.* This operation is automatically invoked when validation occurs. Its operation will be described more thoroughly shortly.

- *DoCommit* and *DoAbort.* Validate is called during the first phase of commit or as part of nested commit and only determines whether the action is valid at the object. In order that certain information about the object and the actions using it are also kept up to date (most notably the *Last* timestamps for events) these two operations perform additional housekeeping

The concurrency controller for the object maintains a set of lists (one per action) of the events executed by an action together with an indication of the time the action first performed that event. Note that each event need only be maintained once. Thus if an action performs the same event more than once, only the first occurrence is noted. The advantage of maintaining the lists on a per action basis as opposed to on a single list or on an event basis is that it becomes simple to traverse this list at validation time since then what is really required is an indication of what events a given action has participated in.

Since there is a need to ensure that the validation routine of the concurrency controller is called whenever an action commits, the same basic strategy can be adopted as for two-phase locking. That is (in this case) an *Optimistic_Record* is created that is logged with the atomic action system. This record identifies the action and the object so that when commit processing occurs the validation routine will be called. Since there is only the need to log the object's use by an action once, an *Optimistic_Record* is only created and logged if this is the first event executed by the action on the object.

Validation of an action at an object requires that each of the operations performed by the action is examined and a determination made as to whether any conflict has occurred by examining the *First* and *Last* timestamps. An outline of the process is shown as Figure 6-9. Although not strictly true C++ this outline

```
boolean OptCC::Validate(ActionId Id)
{
      EventList* ConfEvents;
      Event_Iterator next(Elist[Id]);
      Event* E1, E2;

      while ((E1 = next()) != Null)                    // iterate over events
      {                                                // executed by the action
            ConfEvents = ConflictingEvents(E1);
            Event_Iterator nextconf(ConfEvents);
            while ((E2 = nextconf()) != Null)          // iterate over conflicting
            {                                          // events
                  if (Last[E2] < First[E1])
                        return FALSE;
            }
      }
      return TRUE;
}
```

Figure 6-9: The validation algorithm

describes the process of validation without resorting to detailed data structure manipulation.

The key here is that for each event executed by the action a list of conflicting events is generated and then each event in this list is checked to determine if conflict has occurred. If it has validation fails and the routine returns with a failure indication.

If the action is a top-level one and finally commits then the routine *DoCommit* is called which updates the *Last* timestamp for each event executed by the action and then removes the information about the action from the concurrency controller.

6.6 Combining Approaches

One advantage of adopting object-based concurrency is that individual objects should be able to choose their own method of concurrency control from the wide spectrum of available methods. Unfortunately the choice cannot be as free as it appears since different methods serialise actions in different orders, thus it is possible to end up with a situation in which it appears as though action A executed before action B at one object, while action B executed before action A at another object.

This is clearly an undesirable occurrence and needs to be avoided. Weihl [*Weihl 84*] has developed techniques for classifying when different techniques are compatible. He calls the various classes of protocols *static atomicity*, *dynamic atomicity*, and *hybrid atomicity* depending upon how they affect serialisability.

Static atomicity characterises protocols such as multi-version timestamping, that is, those in which the serialisable order is determined statically. Dynamic atomicity characterises locking protocols i.e. those protocols that determine serialisability dynamically, while hybrid atomicity describes those protocols that combine characteristics of the other two. The optimistic protocols described in the previous section are hybrid atomic.

6.7 Summary

This chapter has considered how the basic philosophy of using type inheritance stood up to the task of implementing different approaches to concurrency control.

It has shown that the approach is particularly suitable for implementing so-called type-specific locking as advocated by several other researchers, and that such types of lock can be handled easily and flexibly by the type-inheritance technique. In particular the style of locking supported by Clouds proved very simple to emulate and implement by deriving a new type of lock (called *TypeLock*) from the basic *Lock* type of chapter four.

However, as has been seen, once such approaches are adopted the underlying system and execution model begin to play their part, such that seemingly correct implementations of conflict checks may still produce problems, most notably in the form of lost updates to objects. As a way of overcoming this problem in the context of the Arjuna system, a revised system model for Arjuna was introduced based upon the use of multi-threaded servers which removed the problems caused by the system model, but left those currently caused by the existing implementation of recovery.

In considering other forms of concurrency control, these problems have been highlighted even further, to the extent that the optimistic approach described in section 6.5 requires a completely different form of recovery mechanism to the one assumed in the previous chapter.

In addition, what it means to handle some of the available concurrency control methods in a nested atomic action environment as opposed to the single level environment in which they were originally conceived has been considered.

Such considerations have led to the idea that following the concurrency control protocol at all levels of the action hierarchy is not often a good idea.

It can be concluded, therefore, that standard two-phase locking using simple read and write locks imposes minimal requirements upon the rest of the system architecture. However, once different approaches to concurrency control are considered, for whatever reason, then the underlying recovery system and execution model begin to play an important part and must be equally flexible. Providing that this flexibility is available, implementing concurrency control via type inheritance appears to be a promising technique.

Chapter 7
Conclusions

This final chapter summarises the material that has been presented in this thesis and indicates some of the possible areas for future research.

7.1 Thesis Summary

The first chapter of this thesis postulated that building reliable distributed systems was difficult but necessary. As the demand for computing power increases, so to does the aspirations and expectations of those using computers as an essential part of their business.

Although the actual hardware and knowledge of how to construct true distributed systems is currently available, programming such systems is a complex task that is currently not very well understood. In particular application programs that execute on a distributed system can fail in very different (and often unexpected) ways to their centralised counterparts. In addition to the problems of failure, concurrent execution of programs, a necessity for high performance, introduces its own problems.

Programming can be difficult enough without having to worry about the problems caused by failure and concurrency, and so this thesis has turned to a particular methodology of program design as a means of handling the general issues of complexity, together with the use of a computing abstraction known as the atomic action to cope with problems introduced by the possibilities of failure and concurrency .

The methodology used, the so-called object-oriented paradigm, views programs as consisting of a collection of objects and a sequence of operations upon those objects. By taking advantage of the property of encapsulation it is possible to view any object simply as a black box. That is, the internal details of the structure of the object are unimportant, only its abstract behaviour is important. By structuring the system as a collection of objects with well-defined behaviour the overall complexity of the system is reduced to manageable proportions.

This thesis has adopted the attitude that this behaviour should also encompass an object's behaviour in the face of failure and concurrency. Thus individual objects should also be responsible for the provision of mechanisms that can cope with failure and concurrent access.

It is inevitable that if left to themselves then the programmers of each object type would invariably implement these recovery and concurrency control mechanisms in different, probably incompatible, and perhaps even incorrect ways. So, in order to introduce some order, atomic actions have been used as a means of co-ordinating the behaviour of objects when failure or concurrent access occurs.

Atomic actions have the important properties of: failure atomicity, that is the atomic action executes successfully to completion or appears not to have executed at all; concurrency atomicity, whereby the concurrent manipulation of objects by different actions is so constrained that it appears as though the actions had executed in some serial order; and permanence of effect, whereby once an action is complete the system will not arbitrarily lose its effects.

In order to ease the implementation of the properties of atomic actions on particular property possessed by object-oriented languages, that of type-inheritance has been utilised. Using this property user-defined types can inherit a set of basic capabilities that make the management of concurrency and failure

far simpler, and equally important, less prone to error than might otherwise be the case.

This thesis has been particularly concerned with the concurrency atomicity property of atomic actions and it has considered how this property might best be provided using the type inheritance property. As was noted in chapter two, there are a great many concurrency control algorithms in existence, and more are developed each month. Yet despite this, the applicability of these algorithms is limited by the environment assumed when they were developed.

The majority of such algorithms are designed for a centralised database environment. In such environments all data access is typically in terms of simple reads or writes of data and since the database is assumed to be centralised there is usually only a single, system-wide concurrency controller. This structure ensures that the concurrency controller and the atomic action (transaction) manager can easily collect sufficient information to establish a global view of the activity of the system. This global view enables certain problems such as that of deadlock to become relatively easy to detect and solve.

Even those algorithms and systems that are distributed still assume that there is only a single concurrency controller per site, so that although no single controller has global knowledge of the entire distributed system, each nonetheless still has total knowledge of the local system and by appropriate communication can form a reasonably accurate picture of the state of the entire distributed system.

In the envisaged object-oriented environment of this thesis, such assumptions are no longer valid since each individual object must make concurrency control decisions based only on purely local knowledge gathered as part of the normal invocation of operations upon the object.

The most suitable concurrency control techniques for this environment are those based upon two-phase locking, which has the particular property that in order to determine whether to grant a lock on some object it only needs purely local information about other locks that have already been granted on the object. Such information can be gathered in an automatic fashion as locks are requested and released on any object.

Unfortunately, locking protocols are prone to deadlock. In other systems such deadlock can be detected by communication amongst the individual concurrency controllers. In the object-oriented environment of this thesis such communication is likely to be prohibitively expensive since each individual object is maintaining its own concurrency control information and although it could possibly all be collected in one place (say an object store) occasionally, ensuring the consistency and currency (that is, how up to date the information is) would be difficult.

Instead this thesis has taken the convenient expedient of using timeouts. However, unlike other researchers timeouts are not placed upon the length of time an atomic action may execute, but instead timeouts are placed upon individual lock requests. Furthermore the atomic action is not automatically aborted if the timeout expires, rather an exception is returned to the caller so that he can take some more appropriate action should one be possible.

This approach is taken because the occurrence of a timeout when setting a lock may not be due to genuine deadlock, rather it could just be that a particular action is taking a long time to execute and is thus holding locks longer than expected, or alternatively the timeout period itself is too short. By returning an exception the programmer is allowed the option of retrying the lock request (perhaps with a different timeout value) in case deadlock was not truly the cause of the problem at all.

In addition this thesis regarded locks simply as objects in their own right, and thus they had the same basic properties as all other objects in the system. For example they could be stored in an object store like any other object, a property that was made use of to overcome some limitations of the underlying system model when an implementation of the ideas proposed in this thesis was considered in chapter five.

Specifying locks as objects also gained flexibility in the types of locks that the basic concurrency control type was able to support. For example, chapter six showed how, by making further use of type inheritance, several different types of type-specific lock could be derived in a simple fashion that amounted to little more than defining the new lock type (by deriving it from the existing *Lock* type) and giving an appropriate conflict detection routine for it.

This approach had one major advantage over that adopted by other researchers. It was possible to implement the ideas underlying this thesis without having to resort to designing and building either a new language and/or a new operating system kernel. Additionally the flexibility gained by this approach meant that the resulting system was not tied to the particular style adopted by the operating system or language. In fact, although the thesis has adopted the language C++ [*Stroustrup 86*], both to describe and implement the ideas, any other object-oriented programming language would have sufficed.

Furthermore, such type-specific locks could be defined without resorting to changes in the basic concurrency control type, thus it was possible to experiment with different types of lock simply to determine whether the increased level of concurrency such locks afforded was worthwhile in terms of the additional complexity introduced into the programming of the operations of the object.

Later, chapter six further demonstrated this flexibility by developing a concurrency control type based upon the optimistic approach. Using this it would be possible to build a system in which some objects used locking as their concurrency control technique, while others used an optimistic approach.

Of course, as was pointed out in that chapter, the flexibility of the approach could lead to problems in that some techniques are not compatible with others. In particular the various concurrency control techniques serialise actions in differing orders. However, the compatibility of the different concurrency control techniques has been researched by others, in particular by Weihl [*Weihl 84*], and so this is not regarded as a serious problem.

Throughout the thesis an explicit use of the concurrency control mechanism has been adopted. That is the programmer of a type (but not the user) has been required to provide explicit calls to the controller as part of the operations supported by a type. Although this is more complicated than providing implicit calls it has some advantages. Firstly it can be assumed that the programmer has explicit knowledge of the semantics of the type and is in by far the best position to determine what concurrency control is likely to be needed. Secondly, implicit invocation usually means that the operations of an object can only be classified as to whether they read or write the object since further semantic information is not available without careful analysis of the actual operations. Finally the implicit approach frequently needs compiler support in that the calls to the concurrency controller must typically be inserted into the prologue executed when an operation is called. The Clouds system from Georgia Institute of Technology [*Dasgupta et el. 85*] has both explicit and implicit mechanisms but relies on a special systems programming language called *Aeolus* and its associated compiler to implement it. Similarly, the Argus project from MIT [*Liskov 88*] pursues an implicit approach in the same fashion, although in his thesis Weihl argued that

the adoption of an explicit approach would enable higher levels of concurrency to be attained.

Finally this thesis has considered the effects that the support for concurrency control has on the underlying system model and upon the recovery system that is providing the failure atomicity property of the atomic action.

Concurrency control based upon two phase locking using the basic, simple policy of allowing multiple reads and exclusive write locks places the minimum of requirements on the rest of the system. It allows multiple servers for an object at a site, and only needs simple hooks into the atomic actions system to function properly. In particular it needs a way of indicating to the atomic action manager that a lock has been set so that the atomic action manager can, when appropriate, inform the concurrency controller of the object to release the lock as the action commits (or aborts).

However, as was shown in chapter six, even the addition of simple type-specific locking begins to place additional requirements on the underlying system. In particular the ability for an object to support concurrent writes to different parts of its state requires that there is either a single server maintaining the object that executes the concurrent writes, or alternatively that the object storage and retrieval mechanism is able to cope with partial object images.

Objects representing such things as counters place even more requirements on the system. Since such objects might allow concurrent writes on the same part of their state, then the recovery mechanisms themselves must be of a particular type; in particular the recovery mechanisms must commute in the same way that the concurrency control commutes. Thus, in this instance the recovery system must either be based on intentions lists, so that the changes made to an object only actually take place when the action that made them actually commits, or

alternatively user-specified compensation routines must be called if the action aborts. Simple state-based recovery which is perfectly acceptable for the simple locking policy, is inadequate to cope with such objects. In addition the optimistic algorithm also required that the recovery system be based on intentions lists so that changes made to the same objects by different actions appeared to have executed in the same serial order.

7.2 Future Work

Although this thesis has concentrated mainly on concurrency control techniques and their implementation in an object-oriented environment, chapter six has shown that the topic cannot be considered in isolation.

In particular, in order for type-specific locking approaches to be effectively implemented and tested, the underlying system model must be further revised. The most obvious area of revision lies in the area of server management. As was pointed out in chapter five, the basic remote procedure call mechanism has already been modified so that calls from related actions are directed to the same server; what we require is that an object is only ever maintained by one server once activated regardless of which action is using it.

This has some problems as far as the basic RPC mechanism is concerned since each server is essentially in a loop, first waiting for an RPC, executing it, and then returning the results. Given that there is only one server, there arises the increased probability that it might be deaf to requests since it may currently be busy serving another.

Chapter six showed a solution to this problem by providing multiple threads of control within each server process. Each thread of control behaves as a lightweight process that is cheap to create and schedule and shares the address space of the main process with all the other threads of control. Using such an

approach, each individual call can be handled by a separate thread thus the server need never be deaf to a call. Whether the server is configured with a fixed number of such threads when it is defined or creates them dynamically for each request is probably unimportant.

Such a server could be produced in the existing workstations by implementing a simple multi-tasking environment for a UNIX process; an approach adopted by the ISIS team at Cornell [*Birman 86*]. Alternatively use could be made of one of the other operating systems that already support such threads of control, such as Amoeba [*Tanenbaum and Mullender 81*] or Mach [*Jones and Rashid 86*] However, a much more attractive approach may be possible since the Computing Laboratory has recently acquired a shared memory multiprocessor (an Encore Multimax™) which also supports such threads in addition to providing true parallelism by virtue of having multiple central processors. It remains to be seen what effects such a change in the server model would have on the orphan detection capabilities of the underlying remote procedure call system.

It might appear that by adopting such an approach the servers are becoming similar to Argus guardians. There are, however, important differences. For example, in Argus, guardians are always active and are restarted automatically whenever necessary. In the Arjuna system a server for an object need not exist until the object is actually activated, and once all the actions using the object have terminated the server may be destroyed. All that is required is that once a server is created it is willing to serve all actions that wish to make use of the object, regardless of their relationships to each other.

™Multimax is a Trademark of Encore Computer Corporation.

Also possible is the combination of several of the concurrency control techniques into a single object. This would enable a programmer for example to choose type-specific locking for some operations and an optimistic approach for others depending upon the concurrency required and the conflicts likely to occur.

Such an approach requires that the implementation language supports multiple inheritance so that a user-defined object may be derived from both *LockCC* and *OptCC*. The version of C++ available at this time does not currently support this feature. However, a research version with this capability has been produced and will probably become available in the future.

In the short term the lock-based approaches could also be combined such that some operations used type-specific locking while others used the simple read write locking of the basic system. Such an approach requires that the concurrency controller keeps each type of lock separate since it does not in general make sense to compare a type-specific lock and an ordinary lock for conflict unless the programmer has made the meanings of such comparisons explicitly defined. This point is analagous to the one made in chapter four regarding why the mode of a lock was made part of the state of the *Lock* type, rather than deriving new lock types immediately. Alternatively, the programmer must take additional care in defining the conflict operation such that comparison of type-specific locks and basic read-write locks had some meaning.

This thesis has claimed that the type-inheritance based approach is flexible and is not tied to any particular language or system. Testing this in reality requires moving the system to another operating system base (a port to Amoeba is being considered), and perhaps more radically, re-implementing the system in another object-oriented language such as Trellis/Owl [*Schaffert et al. 86*] or Smalltalk-80 [*Goldberg and Robson 83*].

Finally, although the aim throughout this thesis has been not to modify a language or operating system, integration of parts of the system into an operating system would inevitably bring improvements, particularly in the areas of performance and memory utilisation, which may be important.

In conclusion it can be said that using type inheritance in the way illustrated in this thesis has allowed the production of a highly flexible system in which a variety of concurrency control techniques may be implemented. Treating the system as a collection of communicating objects has allowed the problems caused by distribution to be ignored. However, as has been shown, concurrency control cannot be considered in isolation.

As increased levels of concurrency are required then the recovery system and the underlying system model all play their part. In some respects this echoes the thoughts of Reed [*Reed 78*] and Weihl [*Weihl 84*] who both treated concurrency control and recovery within an integrated framework in order to improve concurrency. Certainly the relationship between the two concepts is worth exploring more fully.

References

Ada 80

U.S. Department of Defense, *Reference Manual for the ADA Programming Language*, 1980.

Agrawal and DeWitt 85

Agrawal, R., and D.J. DeWitt, "Integrated Concurrency Control and Recovery Mechanisms: Design and Performance Evaluation," *ACM Transactions on Database Systems*, Vol. 10, No. 4, pp. 529-564, December 1985.

Allchin 83

Allchin, J.E., "An Architecture for Reliable Decentralized Systems," Ph.D Thesis, Technical Report GIT-ICS-82/83, Georgia Institute of Technology, September 1983.

Allchin and McKendry 83

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

Anyanwu 84

Anyanwu, J.A., "Robust Data Storage in a Network Of Computer Systems," Ph.D Thesis, Computing Laboratory, University of Newcastle upon Tyne, November 1984.

Arens 81

Arens, G.C., "Recovery of the SWALLOW Repository," M.Sc Dissertation, Technical Report MIT/LCS/TR-252, MIT Laboratory for Computer Science, Cambridge, January 1981.

Avizienis and Chen 77

Avizienis, A. and L. Chen, "On the Implementation of N-Version Programming for Software Fault-Tolerance During Execution," *Proceedings of COMPSAC 77*, pp. 149-155, November 1977.

Banâtre et al. 83

Banâtre, J.-P., M. Banâtre, and F. Ployette, "Construction of a Distributed System Supporting Atomic Transactions," *IEEE Proceedings of the 3rd Symposium on Reliability in Distributed Software and Database Systems*, pp. 95-99, October 1983.

Bayer and Schkolnick 77

Bayer, R., and M. Schkolnick, "Concurrency of Operations on B-Trees," *Acta Information*, Vol. 9, pp. 1-21, 1977.

Bernstein and Goodman 81

Bernstein, P.A., and N. Goodman, "Concurrency Control in Distributed Database Systems," *ACM Computing Surveys*, Vol. 13, No. 2, pp. 185-221, June 1981.

Bernstein et al. 78

Bernstein, P.A., J.B. Rothnie, N. Goodman, and C.H. Papadimitriou, "The Concurrency Control Mechanism of SDD-1: A System for Distributed Databases (The Fully Redundant Case)," *IEEE Transactions on Software Engineering*, Vol. SE-4, No. 3, pp. 154-168, May 1978.

Bernstein et al. 87

Bernstein, P.A., V. Hadzilacos, and N. Goodman, *Concurrency Control and Recovery in Database Systems*, Addison-Wesley, 1987.

Birman 86

Birman, K.P.,"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

Birman, K.P., 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

Birrell, A., and B.J. Nelson, "Implementing remote procedure calls," *ACM Transactions on Computer Systems*, Vol. 2, No. 1, pp. 39-59, February 1985.

Birtwhistle et al. 73

Birtwhistle, G.M., O.-J. Dahl, B. Myhrhaug, and K. Nygaard, *Simula Begin*, Academic Press, 1973.

Bobrow et al. 86

Bobrow, D.G., K. Kahn, G. Kiczales, L. Masinter, M. Stefik, and F. Zdybel, "CommonLoops: Merging Lisp and Object-Oriented Programming," *OOPSLA '86 Conference Proceedings*, pp.17-29, September 1986.

Buckley and Silberschatz 84

Buckley, G.N. and A. Silberschatz, "Concurrency Control in Graph Protocols Using Edge Locks," *Proceedings of the 3rd ACM SIGACT-SIGMOD Symposium on Principles of Database Systems*, Waterloo, Ontario, pp. 45-50, April 1984.

Campbell and Randell 86

Campbell, R.H. and B. Randell, "Error Recovery in Asynchronous Systems," *IEEE Transactions on Software Engineering*, Vol. SE-12, No. 8, pp. 811-826, August 1986.

Carey 87

Carey, M.J., "Improving the Performance of an Optimistic Concurrency Control Algorithm Through Timestamps and Versions," *IEEE Transactions on Software Engineering*, Vol. SE-13, No. 6, pp. 746-751, June 1987.

Ceri and Owicki 82

Ceri, S., and S. Owicki, "On The Use of Optimistic Methods for Concurrency Control in Distributed Databases," *Proceedings of the Sixth Berkeley Workshop on Distributed Data Management and Computer Networks*, February 1982.

Cox 86

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

Dasgupta et al. 85

Dasgupta, P., R.J. LeBlanc Jr., and E. Spafford, "The Clouds Project: Designing and Implementing a Fault Tolerant, Distributed Operating System," Technical Report GIT-ICS-85/29, Georgia Institute of Technology, 1985.

Davies 73

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

Dixon et al. 87

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

Dixon 88

Dixon, G.N., "Object Management for Persistence and Recoverability," Ph.D Thesis, Computing Laboratory, University of Newcastle upon Tyne, in preparation.

Enslow 78

Enslow, P.H., "What is a "Distributed" Data Processing System?," *IEEE Computer*, Vol. 11, No. 1, pp. 13-21, January 1978.

Eswaran et al. 76

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

Franaszeck and Robinson 85

Franaszeck, P., and J.T. Robinson, "Limitations on Concurrency in Transaction Processing,", *ACM Transactions on Database Systems*, Vol. 10, No. 1, pp. 1-28, March 1985.

Fridrich and Older 81

Fridrich, M., and W. Older, "The FELIX File Server," *Proceedings of the 8th Symposium on Operating System Principles*, pp. 37-44, December 1981.

Garcia-Molina 83

Garcia-Molina, H., "Using Semantic Knowledge for Transaction Processing in a Distributed Database," *ACM Transactions on Database Systems*, Vol. 8, No. 2, pp. 186-213, June 1983.

Goldberg and Robson 83

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

Goodenough 75

Goodenough, J.B., "Exception Handling: Issues and a Proposed Notation," *Communications of the ACM*, Vol. 18, No. 12, pp. 683-696, December 1975.

Goodman and Shasha 85

Goodman, N., and D. Shasha, "Semantically-based Concurrency Control for Search Structures," *Proceedings of the 4th ACM SIGACT-SIGMOD Symposium on Principles of Database Systems*, Portland, March 1985.

Gray et al. 75

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

Gray 78

Gray, J.N., "Notes on Data Base Operating Systems," in Operating Systems: An Advanced Course, eds. R. Bayer, R.M. Graham, and G. Seegmueller, pp. 393-481, Springer, 1978.

Hedayati 88

Hedayati, F, "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 86

Herlihy, M.P., "Optimistic Concurrency Control for Abstract Data Types," *Proceedings of the Fifth Annual ACM Symposium on Principles of Distributed Computing*, Calgary, Alberta, pp. 206-216, August 1986.

Herlihy and Wing 87

Herlihy, M.P., and J.M. Wing, "Avalon: Language Support for Reliable Distributed Systems," *Digest of Papers FTCS-17, Seventeenth Annual Symposium on Fault-Tolerant Computing*, Pittsburgh, pp. 89-94, July 1987.

Herlihy 87

Herlihy, M.P., "Extending Multi-Version Time-Stamping Protocols to Exploit Type Information," *IEEE Transactions on Computers,* Vol. SE-36, No. 4, pp. 443-448, April 1987.

Herlihy and Weihl 88

Herlihy, M.P., and W.E. Weihl, "Hybrid Concurrency Control for Abstract Data Types," *Proceedings of the Seventh Annual ACM SIGACT-SIGART Symposium on Principles of Database Systems*, Austin, Texas, March 1988.

Horning et al. 74

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

Hughes 86

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

Jefferson 85

Jefferson, D.R., "Virtual Time," *ACM Transactions on Programming Languages and Systems*, Vol. 7, No. 3, pp. 404-425, July 1985.

Jones 78

Jones, A.K., "The Object Model : A Conceptual Tool for Structuring Software," in *Lecture Notes in Computer Science 60*, eds. R. Bayer, R.M. Graham, and G. Seegmueller, pp. 8-16, Springer, 1978.

Jones et al. 85

Jones, M.B., R.F. Rashid, and M.R. Thompson, "Matchmaker: An Interface Specification Language for Distributed Processing," *Proceedings of the 12th Annual ACM Symposium on Principles of Programming Languages*, pp. 225-235, January 1985.

Jones and Rashid 86

Jones, M.B., and R.F. Rashid, "Mach and Matchmaker: Kernel and Language Support for Object-Oriented Distributed Systems," *OOPSLA '86 Conference Proceedings*, pp.67-77, September 1986.

Kedem and Silberschatz 81

Kedem, Z.M., and A. Silberschatz, "A Characterisation of Database Graphs Admitting a Simple Locking Protocol," *Acta Information*, Vol. 16, pp. 1-13, 1981.

Kenley 86

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

Kernighan and Ritchie 78

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

Kohler 81

Kohler, W.H., "A Survey of Techniques for Synchronisation and Recovery in Decentralised Computer Systems," *ACM Computing Surveys*, Vol. 13, No. 2, pp. 149-183, June 1981.

Kung and Papadimitriou 79

Kung, H.T., and C.H. Papadimitriou, "An Optimality Theory of Concurrency Control for Databases," *Proceedings of the ACM SIGMOD Conference*, pp. 116-126, May 1979.

Kung and Robinson 81

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

Lampson and Sturgis 79

Lampson, B.W., and H.E. Sturgis, "Crash Recovery in a Distributed Data Storage System," Unpublished internal report, Xerox PARC, April 1979.

Lausen 82

Lausen, G., "Concurrency Control in Database Systems : A Step Towards The Integration of Optimistic Methods and Locking," *Proceedings of the ACM 82 Conference*, pp. 64-68, October 1982.

LeBlanc and Wilkes 85

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

Lee et al. 80

Lee, P.A., N. Ghani, and K. Heron, "A Recovery Cache for the PDP-11," *IEEE Transactions on Computers*, Vol. C-29, No. 6, pp. 546-549, 1980.

Lee and Anderson 85

Lee, P.A., and T. Anderson, "Design Fault Tolerance," in *Resilient Computing Systems*, ed. T. Anderson, pp. 64-77, Collins, 1985.

Lindsay et al. 84

Lindsay, B.G., L.M. Haas, C. Mohan, P.F. Wilms, and R.A. Yost, "Computation and Communication in R*: A Distributed Database Manager," *ACM Transactions on Computer Systems*, Vol. 2, No. 1, pp. 24-38, February 1984

Liskov et al. 79

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

Liskov and Scheifler 83

Liskov, B., and R. Scheifler, "Guardians and Actions: Linguistic Support for Robust, Distributed Programs," *ACM Transactions on Programming Languages and Systems*, Vol. 5, No, 3, pp. 381-404, July 1983.

Liskov 84

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

Liskov 88

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

Lomet 77

Lomet, D.B., "Process Structuring, Synchronisation and Recovery Using Atomic Actions," *Proceedings of ACM Conference on Language Design for Reliable Software*, pp. 128-137, March 1977. (Also ACM SIGPLAN Notices, Vol. 12, No. 3).

Lorie 77

Lorie, R.A., "Physical Integrity in a Large Segmented Database," *ACM Transactions on Database Systems.*, Vol. 2, No. 1, pp. 91-104, March 1977.

Melliar-Smith and Randell 77

Melliar-Smith, P.M. and B. Randell, "Software Reliability: the Role of Programmed Exception Handling," *Proceedings of ACM Conference on Language Design for Reliable Software*, pp. 95-100, March 1977. (Also ACM SIGPLAN Notices, Vol. 12, No. 3).

Metcalfe and Boggs 76

Metcalfe, R.M., and D.R. Boggs, "Ethernet: Distributed Packet Switching for Local Computer Networks," *Communications of the ACM*, Vol. 19, No. 7, pp. 395-404, July 1976.

Mohan et al. 83

Mohan, C., and B. Lindsay, "Efficient Commit Protocols for the Tree of Processes Model of Distributed Transactions," *Proceedings of the 2nd Annual ACM Symposium on Principles of Distributed Computing*, pp. 76-88, August 1983.

Mohan et al. 86

Mohan, C., B. Lindsay, and R. Obermarck, "Transaction Management in the R* Distributed Database Management System," *ACM Transactions on Database Systems*, Vol. 11, No. 4, pp. 378-396, December 1986.

Moon 86

Moon, D.A. "Object-Oriented Programming with Flavors," *OOPSLA '86 Conference Proceedings*, pp. 1-8, September 1986.

Moore 82

Moore, J.D., "Simple Nested Transactions in LOCUS: A Distributed Operating System," M.Sc Dissertation, University of California at Los Angeles, 1982.

Moss 81

Moss, J.E.B., "Nested Transactions : An Approach to Reliable Distributed Computing," Ph.D Thesis, MIT/LCS/TR-260, MIT Laboratory for Computer Science, Cambridge, Mass., April 1981.

Mueller 83

Mueller, E.T., "Implementation of Nested Transactions in a Distributed System," M.Sc Dissertation, University of California at Los Angeles, 1983.

Mueller et al. 83

Mueller, E.T., J.D. Moore, and G.J. Popek, "A Nested Transaction Mechanism for LOCUS," *Proceedings of the 9th ACM Symposium on Operating System Principles*, pp. 71-89, October 1983.

Mullender and Tanenbaum 85

Mullender, S.J., and A.S. Tanenbaum, "A Distributed File Service Based on Optimistic Concurrency Control," *Proceedings of the 10th ACM Symposium on Operating System Principles*, pp. 51-62, December 1985.

Nelson 81

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

Nett et al. 85

Nett, E., 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, 1985.

Panzieri and Shrivastava 88

Panzieri, F., 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.

Papadimitriou 79

Papadimitriou, C.H., "Serializability of Concurrent Updates," *Journal of the ACM,* Vol. 26, No. 4, pp. 631-653, October 1979.

Parrington and Shrivastava 88

Parrington, G.D., 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 (to be published.)

Randell 75

Randell, B., "System Structure for Software Fault Tolerance," *IEEE Transactions on Software Engineering*, Vol. SE-1, No. 2, pp. 220-232, June 1975.

Rashid and Robertson 81

Rashid, R., and G. Robertson, "Accent: A Communication Oriented Network Operating System Kernel," *Proceedings of the 8th ACM Symposium on Operating System Principles*, pp. 64-75, December 1981.

Reed 78

Reed, D.P., "Naming and Synchronisation in a Decentralized Computer System," Ph.D Thesis, MIT/LCS/TR-205, MIT Laboratory for Computer Science, Cambridge, Mass., September 1978.

Reed 83

Reed, D.P., "Implementing Atomic Actions on Decentralized Data," *ACM Transactions on Computer Systems*, Vol. 1, No. 1, pp. 3-23, February 1983.

Robinson 82

Robinson, J.T., "Design of Concurrency Controls for Transaction Processing Systems," Ph.D Thesis, CMU-CS-82-114, Department of Computer Science, Carnegie-Mellon University, April 1982.

Rosenkrantz et al. 78

Rosenkrantz, D.J., R.E. Stearns, and P.M. Lewis II, "System Level Concurrency Control for Distributed Database Systems," ACM Transactions on Database Systems, Vol. 3, No. 2, pp. 178-198, June 1978.

Schaffert et al. 86

Schaffert, C., T. Cooper, B. Bullis, M. Kilian, and C. Wilpolt, "An Introduction to Trellis/Owl," *OOPSLA '86 Conference Proceedings*, pp. 9-16, September 1986.

Schlicting and Schneider 83

Schlicting, R.D., and F.B. Schneider, "Fail-Stop Processors: An Approach to Designing Fault-Tolerant Computing Systems," *ACM Transactions on Computer Systems*, Vol. 1, No. 3, pp. 222-238, August 1983.

Schwarz and Spector 82

Schwarz, P.M., and A.Z. Spector, "Synchronizing Shared Abstract Types," Technical Report CMU-CS-82-128, Department of Computer Science, Carnegie-Mellon University, September 1982.

Schwarz 84

Schwarz, P.M., "Transactions on Typed Objects," Ph.D Thesis, Technical Report CMU-CS-84-166, Department of Computer Science, Carnegie-Mellon University, December 1984.

Severance and Lohman 76

Severance, D.G., and G.M. Lohman, "Differential Files: Their Application to the Maintainance of Large Databases," ACM Transactions on Database Systems, Vol. 1, No. 3, pp.256-267, September 1976.

Sha et al. 83

Sha, L., E.D. Jensen, R.F. Rashid, and J.D. Northcutt, "Distributed Co-operating Processes and Transactions," in *Distributed Computing Systems*, eds. Y. Paker and J. P. Verjus, pp. 23-50, Academic Press, 1983.

Sha 85

Sha, L., "Modular Concurrency Control and Failure Recovery --- Consistency, Correctness and Optimality," Ph.D Thesis, Technical Report CMU-CS-85-114, Department of Computer Science, Carnegie-Mellon University, March 1985.

Sha et al. 88

Sha, L., J.P. Lehoczky, and E.D. Jensen, "Modular Concurrency Control and Failure Recovery," *IEEE Transactions on Computers*, Vol. 37, No. 2, pp. 146-159, February 1988.

Shrivastava 82

Shrivastava, S.K., "A Dependency, Commitment and Recovery Model for Atomic Actions," *Proceedings of 2nd IEEE Symposium on Reliability in Distributed Software and Database Systems*, Pittsburgh, pp. 112-119, 1982.

Shrivastava 85

Shrivastava, S.K., (ed) *Reliable Computer Systems: Collected Papers of the Newcastle Reliability Project*, Springer-Verlag, 1985.

Shrivastava 86

Shrivastava, S.K., "An Introduction to Arjuna: A System for Reliable Distributed Programming," Internal Report SRM/439, Computing Laboratory, University of Newcastle upon Tyne, August 1986.

Shrivastava et al. 88

Shrivastava, S.K., G.N. Dixon, F. Hedayati, G.D. Parrington, and S.M. Wheater, "A Technical Overview of Arjuna: A System for Reliable Distributed Computing," Proceedings of the Alvey Conference, July 1988. (to be published)

Skeen 81

Skeen, D., "Nonblocking Commit Protocols," *Proceedings of the ACM International Conference on the Management of Data*, pp. 133-142, April 1981.

Sloman 87

Sloman, M., and J. Kramer, *Distributed Systems and Computer Networks*, Prentice-Hall, 1987.

Snyder 86

Snyder, A., "Encapsulation and Inheritance in Object-Oriented Programming Languages," *OOPSLA '86 Conference Proceedings*, pp.38-45, September 1986.

Spector et al. 85a

Spector, A.Z., 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. 85b

Spector, A.Z., D.S. Daniels, D.J. Duchamp, J.L. Eppinger, and R. Pausch, "Distributed Transactions for Reliable Systems," *Proceedings of the 10th ACM Symposium on Operating Systems Principles*, pp. 127-146, December 1985.

Spector 87

Spector, A.Z., "Distributed Transaction Processing and The Camelot System," Technical Report CMU-CS-87-100, Department of Computer Science, Carnegie-Mellon University, January 1987.

Spector et al. 87

Spector, A.Z., 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.

Stearns and Rosenkrantz 81

Stearns, R.E., and D. J. Rosenkrantz, "Distributed Database Concurrency Control Using Before-Values," *Proceedings of the ACM International Conference on the Management of Data*, pp. 74-83, April 1981.

Stonebraker et al. 85

Stonebraker, M., D. DuBourdieux, and W. Edwards, "Problems in Supporting Database Transactions in an Operating Systems Transaction Manager," *ACM Operating Systems Review*, Vol. 19, No. 1, pp.6-14, January 1985.

Stroustrup 86

Stroustrup, B., *The C++ Programming Language*, Addison Wesley, 1986.

Stroustrup 87a

Stroustrup, B. "What is 'Object-Oriented Programming'?," Technical Report, AT&T Bell Laboratories, 1987.

Stroustrup 87b

Stroustrup, B. "Multiple Inheritance for C++," *Proceedings of the EUUG Conference*, Helsinki, May 1987.

Svobodova 80

Svobodova, L., "Management of Object Histories in the SWALLOW Repository," Technical Report MIT/LCS/TR-253, MIT Laboratory for Computer Science, Cambridge, Mass., July 1980.

Svobodova 81

Svobodova, L., "A Reliable Object Oriented Data Repository for a Distributed Computer System," *Proceedings of the 8th ACM Symposium on Operating System Principles*, pp 47-58, December 1981.

Tanenbaum and Mullender 81

Tanenbaum, A.S., and S.J. Mullender, "An Overview of the Amoeba Distributed Operating System," *ACM Operating Systems Review*, pp. 51-64, July 1981.

Tanenbaum and Renesse 85

Tanenbaum, A.S., and R. van Renesse, "Distributed Operating Systems," *ACM Computing Surveys*, Vol. 17, No. 4, pp. 419-470, December 1985.

Tay et al. 85

Tay, Y.C., N. Goodman, and R. Suri, "Locking Performance in Centralized Databases," *ACM Transactions on Database Systems*, Vol. 10, No. 4, pp. 415-462, December 1985.

Thomas 79

Thomas, R.H., "A Majority Concensus Approach to Concurrency Control for Multiple Copy Databases," *ACM Transactions on Database Systems*, Vol. 4, No. 2, pp. 180-209, June 1979.

Walker et al. 83

Walker, B., G.J. Popek, R. English, C. Kline, and G. Thiel, "The LOCUS Distributed Operating System," *Proceedings of the 9th ACM Symposium on Operating System Principles*, pp. 49-70, October 1983.

Walker 85

Walker, B.J., *The Locus Distributed System Architecture*, MIT Press, 1985.

Wegner 86

Wegner, P., "Classification of Object-Oriented Systems," ACM SIGPLAN Notices, Vol. 21, No. 10, pp. 173-182, October 1986.

Wegner 87

Wegner, P., "Dimensions of Object-Based Language Design," *OOPSLA '87 Conference Proceedings*, pp.168-182, October 1987.

Weihl 84

Weihl, W., "Specification and Implementation of Atomic Data Types," Ph.D Thesis, MIT/LCS/TR-314, MIT Laboratory for Computer Science, Cambridge, Mass., March 1984.

Weihl and Liskov 85

Liskov, B., and W. Weihl, "Implementation of Resilient, Atomic Data Types," *ACM Transactions on Programming Languages and Systems,* Vol. 7, No. 2, pp. 244-269, April 1985.

Weinstein et al. 85

Weinstein, M.J., T.W. Page Jr., B.K. Livezey, and G.J. Popek, "Transactions and Synchronization in a Distributed Operating System," *Proceedings of the 10th ACM Symposium on Operating System Principles*, pp. 115-126, December 1985.

Wheater 88

Wheater, S. "Distributed Programming in C++," Technical Report, Computing Laboratory, University of Newcastle upon Tyne (to appear).