Technical Report

UCAM-CL-TR-338
ISSN 1476-2986

Number 338

UNIVERSITY OF CAMBRIDGE
Computer Laboratory

A new approach to implementing atomic data types

Zhixue Wu

May 1994

15 JJ Thomson Avenue
Cambridge CB3 0FD
United Kingdom
phone +44 1223 763500

https://www.cl.cam.ac.uk/

This technical report is based on a dissertation submitted October 1993 by the author for the degree of Doctor of Philosophy to the University of Cambridge, Trinity College.

ISSN 1476-2986

Abstract

Many researchers have suggested the atomic data type approach to maintaining data consistency in a system. In this approach, atomicity is ensured by the data objects that are shared by concurrent activities. By using the semantics of the operations of the shared objects, greater concurrency among activities can be permitted. In addition, by encapsulating synchronisation and recovery in the implementation of the shared objects, modularity can be enhanced. Existing systems support user-defined atomic data types in an explicit approach. They either permit limited semantics to be presented thus providing less concurrency, or permit a high level of semantics to be presented but in an encapsulated way, thus resulting in a complicated implementation. This research was done to make the implementation of user-defined atomic data types simple, efficient, while still permitting great concurrency.

The research aims to lessen the programmer's burden by supporting an implicit approach for implementing atomic data types. It permits a high level of semantics to be specified in a declarative way, which makes the implementation of user-defined atomic data types as simple as in a sequential environment. A special concurrency control mechanism is implemented by the system. By using type inheritance, user-defined atomic data types can use the mechanism directly to provide local atomicity for their objects. A language has been developed for specifying the conflicts between object operations. Since the concurrency control mechanism can take operation semantics into account, the approach permits great concurrency.

To support the implicit approach, an appropriate concurrency control protocol must be proposed which can take advantage of operation semantics to increase concurrency and which can be implemented independently from user-defined atomic data types. Such a protocol, called the *dual-level validation* method, is presented and verified in this thesis. The method can make use of the parameters and results of object operations to achieve great concurrency. In addition, it also provides great internal concurrency by permitting operations to take place on an object concurrently.

The prototyping of the implicit approach in a persistent programming language called PC++ is described. The feasibility of the approach is shown by an application, namely a naming database for an active badge system. Some related issues are also addressed in the thesis, such as remote object invocation, distributed transaction commitment and data persistence.

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 many applications are totally dependent on the correct functioning of their computer systems. In these applications the manipulation and preservation of long-lived, on-line data is of primary importance. Examples of such applications are banking systems, airline reservation systems, database systems, and office automation systems. A major issue in such systems is preserving the consistency of on-line data in the presence of concurrency and failures. This thesis is concerned with how to define and implement data objects that help to provide needed consistency.

To support consistency it is useful to make concurrent activities that use and manipulate data objects *atomic*. Atomic activities are referred to as *transactions* or *actions*, which are first identified in work on database systems [Dav73, Dav78, EGLT76]. Transactions are characterised informally by the following properties: recoverability, serialisability, isolation and durability. *Recoverability* means that either all or none of the transaction's operations are performed. *Serialisability* means that if several transactions are executed concurrently, the results must be the same as if they were executed serially in some order. Serialisability together with recoverability are called *atomicity*. *Isolation* means that a transaction cannot reveal its results to other transactions before its commitment. *Durability* means that once a transaction has committed the system must guarantee that the results of its operations will never be lost, regardless of subsequent failures.

Transactions simplify the problem of maintaining consistency by decreasing the number of cases that need to be considered. The properties of transactions allow one to guarantee that the data objects remain consistent by ensuring that each transaction, when run alone and to completion, preserves consistency. Given that each transaction preserves consistency, any serial execution of transactions without failures also preserves consistency. In a transaction system, a concurrent execution with failures is equivalent to some serial execution without failures; thus, a concurrent execution with failures also preserves consistency.

Atomicity can be achieved by implementing applications in terms of *atomic data types*: data types whose objects, *atomic data objects*, provide serialisability and recoverability for transactions using them. Atomicity of transactions is guaranteed when all objects shared by transactions are atomic objects [WL85]. By encapsulating the synchronisation and recovery needed to support atomicity in the implementation of the shared objects, one can enhance modularity; in addition, by using the information about the specification of the shared objects, one can increase concurrency among transactions while still ensuring atomicity.

Implementing atomic data types is a difficult task, as shown in previous work [Wei84, WL85, Wei90]. Increasing concurrency would lead to complicated implementations. The research results at present are not satisfying; further work is needed in designing and evaluating alternative methods. The subject of this thesis is to explore a new approach to implementing atomic data types, which permits a high level of concurrency and leads to a much easier implementation.

1.1 Properties of Transactions

The *transaction* concept has emerged as an abstraction which allows programmers to group a sequence of operations into a logical execution unit. If executed atomically, a transaction transforms a consistent state of data objects into a new consistent state. A transaction is characterised by the following properties [CP84]:

Recoverability Either all or none of the transaction's operations are performed. Recoverability requires that if a transaction is interrupted by a failure, its partial results are undone.

There are two typical reasons why a transaction is not completed: **transaction aborts** and **system crashes**. The abort of a transaction can be requested by the transaction itself (or by its user) because some of its input is wrong or because some conditions are recognised that make transaction completion inappropriate or useless. A transaction abort can also be forced by the system for system-dependent reasons, typically system overloads and deadlocks. The activity of ensuring recoverability in the presence of transaction aborts is called **transaction recovery**, and the activity of ensuring recoverability in the presence of system crashes is called **crash recovery**.

Serialisability If several transactions are executed concurrently, the results must be the same as if they were executed serially in some order. The activity of guaranteeing transactions' serialisability is called **concurrency control**.

Durability Once a transaction has committed, the system must guarantee that the results of its operations will never be lost, regardless of subsequent failures. The activity of providing the transaction's durability is called **database recovery**.

Isolation A transaction cannot reveal its results to other transactions before its commitment. This property is needed in order to avoid the problem of **cascading aborts**, *i.e.,* the necessity to abort all the transactions which have observed the partial results of a transaction which was later aborted.

Note that these properties relate to the definition of transactions and do not imply particular methods of implementation. The effect on system state of running transactions is constrained by these properties.

1.2 Concurrency Control Techniques

Many concurrency control techniques have been developed for synchronising concurrent activities to ensure serialisability. A good survey of these techniques can be found in [Koh81, BG81, BHG87]. Concurrency control techniques can be broadly classified into two distinct classes: *pessimistic* and *optimistic.* Pessimistic schedulers prevent potentially conflicting operations from occurring. In doing so they must always assume the worst possible case in that if two operations might conflict, they assume that the conflict will happen. Optimistic schedulers, on the other hand, allow free access to the data objects and then attempt to determine if any conflict has occurred at some later point, usually when a transaction commits. Thus, they assume that conflict will not occur, and only take action if it actually does.

1.2.1 Pessimistic Concurrency Control

Pessimistic approaches to concurrency control prevent potentially conflicting operations from occurring. Such techniques are pessimistic because they always assume the worst possible case. Consequently pessimistic approaches tend to restrict concurrency somewhat more than necessary. Most pessimistic techniques are variations or hybrids of two simple techniques: *two-phase locking* (**2PL**) [EGLT76, BG81] and *timestamp ordering* (**TSO**) [Ree78]. These two techniques and a hybrid of them are discussed below.

Two-Phase Locking

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

One of the earliest locking protocols developed for concurrency control is 2PL. It works as follows: Before reading an object, a transaction must acquire a read lock on the object. Similarly, before writing an object, a transaction must acquire a write lock on the object. A transaction can acquire a lock on an object only if no concurrent transaction holds a conflicting lock on the object. In addition, once a transaction releases one lock, it is not allowed to acquire any additional locks.

The requirement that a transaction not acquire any more locks after it releases a lock divides the acquisition and release of locks into two distinct phases. During the first phase (called the *growing* phase) locks can only be acquired. During the second phase (called the *shrinking* phase) locks may only be released. In the paper [EGLT76], it was proved that by following this approach then serialisability was guaranteed.

A variant of 2PL, called *strict 2PL*, is more suitable. Under strict 2PL, transactions hold all locks until they commit or abort. This avoids cascading aborts, a problem with non-strict 2PL.

Timestamp-Based Protocols

The serialisation order of transactions maintained by 2PL is determined *dynamically* by the order in which activities lock objects. In contrast, timestamp-based protocols determine the serialisation order *statically* by selecting timestamps for transactions when they start, and then force the execution of transactions to obey this order.

Timestamps are values drawn from a totally ordered domain [BHG87]. Each transaction T_i is assigned a timestamp, denoted TS_i, such that if $T_i \neq T_j$ then either $TS_i < TS_j$ or $TS_j < TS_i$.

The rules of timestamp-based concurrency control state that conflicting operations must be carried out in timestamp order, thus if any request arrives out of order it must be rejected. *Basic timestamping* concurrency schedulers are thus aggressive in nature since operations are performed strictly first in, first out.

Basic timestamping could abort a large number of requests if the order in which requests are processed by the scheduler differs badly from the timestamp order. A *conservative timestamping* concurrency scheduler attempts to avoid this problem by queueing requests for a while to see if any requests with earlier timestamps will arrive [BRGP78]. Obviously the longer the delay imposed, the smaller the number of rejections that should be generated. However this will slow the processing rate.

Multi-version timestamping was introduced by Reed [Ree83]. The protocol works as follows: A transaction is assigned a unique timestamp when it begins execution. When a transaction wants to modify an object, it creates a new version of the object. A version of an object has two timestamps associated with it: a *write* timestamp, which is the timestamp of the transaction that created the version; and a *read* timestamp, which is the maximum of the timestamp of transactions that have read the version. When a transaction with timestamp t wants to read an object, it selects the version with the largest write timestamp less than t and changes the read timestamp of the version to the maximum of its current value and t.

In a case where a transaction with timestamp t wants to write an object, and a version v of the object already exists with a write timestamp less than t and a read timestamp greater than t, then the write operation must be refused in order to ensure the serialisability. To avoid cascading aborts, read operations sometimes must be delayed: if a transaction with timestamp t wants to read an object and the version selected was written by a transaction that has not yet committed or aborted, the read operation must wait until that transaction completes. Otherwise, if the activity that created the version later aborts, the reader must also be aborted.

Hybrid Protocols

Locking and timestamp-based protocols can be combined to yield *hybrid protocols* that achieve greater concurrency [BG81, BG83, Wei87]. Transactions are divided into two classes: *read-only transactions*, which never modify objects; and *update transactions*.

The *mixed method* in [BG83] is a hybrid protocol. It uses multiversion timestamping to process read-only transactions, and multiversion locking to process update transactions. Update transactions set locks on objects as in strict 2PL, but two locks conflict only if one is a read lock and the other is a write lock; a write lock no longer conflicts with another write lock. As with Reed's protocol, timestamps for read-only transactions are chosen when they begin execution, and each write operation creates a separate version and each version has two timestamps. Rather than choosing timestamps for update transactions when they begin executing, however, it waits until they attempt to commit. Then, using a Lamport clock [Lam78], it ensures that the timestamps chosen for updates give a serialisation order consistent with the order induced by the locks. The order of conflicting write operations is sorted out using the timestamps.

Update transactions that invoke read operations always read the version with the largest timestamp. Read-only transactions, however, may read older versions, permitting them to run without interfering with update transactions. Timestamps for read-only transactions are chosen when they start executing. When a read-only transaction wants to read an object, it simply selects the version with the largest write timestamp less than the transaction's timestamp.

1.2.2 Optimistic Concurrency Control

Optimistic concurrency control is based upon the premise that it is sometimes easier to apologise than to ask permission [Her90]. That is, whereas pessimistic approaches always obtain permission to use an object before they actually do so, optimistic approaches use an 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 cheaper than preventing conflicts from occurring in the first place. Any optimistic scheme, however clever, is cost effective only if validation succeeds sufficiently often.

Optimistic approaches divide transaction execution into three stages:

- *Read Phase.* During this phase transactions read objects but only write to local copies that are not visible to others and subsequently read from these.
- *Validation Phase.* Prior to making objects they have written visible to others, transactions 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.

Härder [Ḧ84] has distinguished between *backward validation*, in which each transaction checks that its own results have not been invalidated by concurrent transactions, and *forward validation*, in which each transaction checks that its own effects will not invalidate any concurrent transaction's results.

Herlihy [Her87a, Her90] has distinguished between *conflict-based validation*, a simple validation technique based on predefined conflicts, and *state-based validation*, a more complex scheme that validates additional interleaving by exploiting knowledge about the object's state.

1.3 Atomic Data Types

An atomic data type, like an ordinary abstract data type, provides a set of objects and a set of operations; and the operations provided are the only means to access or manipulate objects of the type. Unlike ordinary abstract data types, however, an atomic data type provides serialisability and recoverability for transactions that use objects of the type.

1.3.1 Specifications

An atomic data type is defined by two specifications: a *serial specification*, which describes the object's permissible behaviour in the absence of concurrency and failures, and a *behavioural specification*, which describes how the object responds to concurrency and failures [WL85]. The serial specification can be used by users to reason about the partial correctness of an individual transaction without considering the other transactions that might be sharing objects with it. The behavioural specification describes the kinds of concurrent executions permitted by the type and how the type handles failures. For the type to be atomic, these executions must be constrained so that transactions using objects of the type are serialisable and recoverable.

1.3.2 Global Atomicity and Local Atomicity

Atomicity of transactions is called *global atomicity* because it is a property of all of the transactions in a system; and atomicity for a type is called *local atomicity* because it deals only with the events (invocations and returns of operations and commits and aborts of transactions) involving the particular type [Wei84]. Such locality is essential if atomic types are to be specified and implemented independently of each other and of the transactions that use them.

Weihl [Wei84, Wei89] has identified three local atomicity properties that result in global atomicity. The three properties characterise the behaviour of three common classes of concurrency control protocols. The *dynamic atomicity* characterises the behaviour of a class of types that ensures serialisability dynamically based on the order in which transactions execute operations provided by the type. Dynamic atomic types can be implemented with protocols like two-phase locking. The *static atomicity* characterises the behaviour of a class of types that ensures serialisability statically based on some predetermined order of transactions. Static atomic types can be implemented with protocols like Reed's multi-version timestamp protocol. The *hybrid atomicity* characterises the behaviour of a class of types that ensures serialisability based on the order in which transactions commit. Hybrid atomic types can be implemented with protocols like the multi-version scheme proposed in [BG83].

Each of these three local atomicity properties is optimal: no strictly weaker local property suffices to ensure atomicity; and each of these three properties is adequate to ensure global atomicity of transactions. For example, if all types shared by transactions are static atomic, transactions are guaranteed to be serialisable and recoverable. However, different local properties may be incompatible. For example, if some types shared by transactions are dynamic atomic and others are static atomic, non-serialisable executions of the transactions can result. Thus in any system, it is necessary to choose a particular local atomicity property that will be used for all shared types in the system.

1.4 Implementations of Atomic Data Types

To some extent the issues involved in implementing an atomic data type are similar to those that arise in implementing other abstract types. The implementation must define a representation for the atomic objects, and an implementation for each operation of the type in terms of that representation. However, there are two other essential problems that must be solved by the implementation of atomic data types, namely: inter-transaction synchronisation, and inter-operation synchronisation.

Inter-transaction synchronisation is a mechanism for managing *transaction concurrency* and failures, *i.e.* ensuring appropriate synchronisation and recovery for transactions using objects of the type. On the other hand, *inter-operation synchronisation* is a mechanism for managing *process concurrency* [WL85] and operation failures. That is, making operations defined on the objects behave like an elementary operation, *i.e.* to appear serial and isolated from each other.

There are a number of systems which support transactions by utilising atomic objects: Argus [LS83, WL85, LCJS87],Clouds [AM83, LW85], Arjuna [SDP91, Dix88, Par88], TABS [SBD$^+$85, SDD$^+$85], and Camelot [SBD$^+$85, STP$^+$87]. Although they differ in detail, their methods to implement user-defined atomic data types can be classified into two kinds: the Argus approach and the extended two-phase locking approach. In this section, we briefly describe these two approaches, analyse them and illustrate their problems. For each of them, we consider how the two synchronisations are implemented, how the semantics of object operations is represented and used, and how a user-defined atomic data type is represented.

1.4.1 The Argus Approach

The Argus approach is designed to support implementation of (*pessimistic*) *dynamic atomic types.* It is an explicit approach in that both *inter-transaction synchronisation* and *inter-operation synchronisation* need to be implemented by atomic type designers. To help designers to implement inter-operation synchronisation, Argus provides the built-in type generator **mutex** and the **seize** statement. A mutex object is essentially a container for another object and provides mutual exclusion. Designers can use **mutex** and **seize** to ensure mutual exclusion among regions of code executed by concurrently executing operations; thus, for example, implementations can prevent interference among concurrently executing operations by forcing them to run serially.

The Argus approach ensures that an operation is elementary by using exclusive locks to prevent interference among concurrently executing operations. Since locks are only held during an operation, measures need to be taken to prevent the partial result of a transaction being seen by others. Argus provides a special type called **atomic_variant** for this purpose. Whenever a data item is created, it should be associated with an atomic variant that is used to indicate the status of the data item: only when the corresponding transaction has committed can that data item be seen by other transactions. However, implementations of user-defined atomic types are not informed about *commit* and *abort* events, but must instead find out about them after the fact through the use of built-in atomic types. This approach makes atomic data type implementation complicated and likely to have bugs, it also results in poor performance. For example, in the semiqueue example given by [Wei84], besides an array defined to record data items, an **atomic_variant** object is created for each newly entered data item; this atomic variant will have a tag "enqueued" if the calling transaction commits later, and a tag "dequeued" if it aborts. Therefore, to find the next data item to dequeue, the *dequeue* operation has to scan the array and check whether the atomic variant associated with a data item has a tag "enqueued". The *dequeue* operation is inefficient, since in the worst case it takes time proportional to the size of the representation of the semiqueue.

In the Argus approach, transaction synchronisation is realised in the implementation of object operations. Thus, the semantics of object operations can be used directly for synchronising transactions without representing explicitly. The advantage of this approach is that more information such as the object state, local variables, operation's parameters and operation's results can be used for making synchronisation decisions, hence greater concurrency may be achieved. However, the drawback of this approach is obvious. First, this approach makes the atomic data types become difficult to implement. Second, there exists potential dangers, since careless programming could lead to chaos. Third, changing the concurrency control method will result in the re-implementation of atomic objects.

The representation of a user-defined atomic type in the Argus approach is a combination of atomic and non-atomic objects, with the non-atomic objects used to hold the application information that can be accessed by concurrent transactions, and the atomic objects containing the synchronisation information that allows the non-atomic data to be interpreted properly.

1.4.2 The Extended Two-Phase Locking Approach

The extended two-phase locking (E2PL) approach is also an explicit approach. Like in Argus, exclusive locks are used for providing inter-operation synchronisation. An extended two-phase locking protocol [BG81, Kor81], which permits type-specific locks to increase concurrency, is used for providing inter-transaction synchronisation. However, unlike Argus, the synchronisation information and the application information are represented independently and the semantics of object operations is specified in a declarative way through a lock compatibility matrix.

The E2PL approach partitions the set of object operations into classes, and uses a different lock mode for each class. A lock mode for one class is compatible with a lock mode for another class if all operations in the first class commute with all operations in the second class. Two lock modes conflict if they are not compatible. The execution of an operation must first acquire a lock in the mode defined for its class. A lock can be acquired if no concurrent activity holds a conflicting lock. Locks are released when activities complete. If an operation is unable to acquire its lock, it waits until conflicting activities complete. Implementing inter-transaction synchronisation is much easier in this approach than in the Argus approach, because it is not implemented in an *ad hoc* way, but in the typical locking pattern. However, this approach provides less concurrency than the Argus approach in that it uses less semantics of object operations when making synchronisation decisions. For example, it does not take operation's parameters and results into account.

Another problem of the E2PL approach is that only the operation-based recovery can be used, and that recovery operations need to be provided by the programmer. This is caused by the short duration of locks set for inter-operation synchronisation(see Chapter 2 for detail).

The representation of a user-defined atomic data type in the E2PL approach consists of data items and locks. Data items are used for representing application information, locks for making synchronisation decisions.

1.4.3 Conclusions

In summary, the Argus approach makes the system flexible, for the designer can choose variant protocols freely in a certain scope (*e.g.* within dynamic protocols). It also permits high concurrency, for it allows great amount of semantics to be used for making synchronisation decisions. However, it is clear from the example implementations presented in the related papers [LS83, WL85, LCJS87] that implementing user-defined atomic data types in Argus is a difficult task, although some linguistic support has been provided. It requires considerable sophistication on the part of the programmer to implement atomic data types, since the programmer has to provide both inter-operation and inter-transaction synchronisation, and to encapsulate the semantics of object operations in their implementations. It may result in a complicated object representation and operation implementation, and consequently, poor performance. It also makes the implementation difficult to verify and likely to have bugs.

On the other hand, the E2PL approach makes atomic data types become less difficult to implement. The inter-transaction synchronisation can be done in the typical locking pattern and the semantics of object operations can be specified in a declarative way. The E2PL approach, however, permits less concurrency than the Argus approach because less semantical information can be taken into account when making synchronisation decisions.

Atomic data types are clearly useful in many applications for supporting atomicity. However, it is not clear how often the programmer will need to take on the job of implementing synchronisation and recovery. It may be that the performance demands of most applications can be satisfied without basing synchronisations on the specifications of objects, or that only a few types in a system need to provide this extra concurrency. It is clear that this extra concurrency will be provided rarely as long as it is so difficult to construct an implementation that provides it [Wei84]. Therefore, the implementation of atomic data types needs to be further explored in an attempt to develop new approaches which lead to a simple and efficient implementation, and still permit great concurrency.

1.5 Research Statement

The aim of this research is to investigate whether it is feasible to provide a mechanism that makes the implementation of user-defined atomic data types simple, efficient, while still permitting great concurrency.

An implicit approach is proposed in this thesis. This approach has the following characteristics:

1. It permits the designer to represent the semantics of object operations in a declarative way instead of by encapsulating it into the implementation of object operations. Moreover, it can represent a great amount of semantics.
2. It releases the designer from the burden of implementing inter-transaction and inter-operation synchronisation by making the system do the work according to the semantics provided by the designer.
3. It increases internal concurrency by permitting operations on an object to be executed concurrently.
4. It limits the linguistic requirement for the language to a minimum by using the preprocessing method and type-inheritance facility to provide appropriate support.

The thesis explores how the targets envisaged above may be reached. First, a concurrency control protocol, called the *dual-level validation* (**DLV**) method, is presented, formalised and verified; then an implicit approach by which user-defined atomic data types can be implemented is explained; and then the language used for specifying the semantics of object operations is described. The thesis also describes briefly how distributed transactions are constructed, how fault-tolerance is supported, how data persistence is achieved, and how remote object invocation transparency is provided.

The thesis describes how such an implicit approach can be provided by taking a prototype implementation of a persistent extension to C++ as an example, and also presents an application illustrating how user defined atomic data types can be implemented and used.

1.6 Structure of the Thesis

The general problem of concurrency control has been discussed in this chapter. However, some special issues on concurrency control arise when implementing atomic data types, and these are examined in some detail in Chapter 2.

Chapter 3 and Chapter 4 are devoted to the concurrency control method **DLV**. It is introduced and specified in Chapter 3, and formalised and verified in Chapter 4.

Chapter 5 focusses on global atomicity in both the absence and the presence of failures. First, a model in a centralised system is established and then extended to be usable in distributed systems.

In Chapter 6, the *type-inheritance* concept of object-orientation is introduced and the *type-inheritance approach* used to implement user defined atomic data types and to construct transactions is described. The language developed for specifying operation semantics is also discussed.

Chapter 7 presents a prototyping implementation of the proposed approach.

Chapter 8 considers how to make a programming language provide data persistence. The related problems such as object naming, binding and type checking, data abstraction and protection, and referential integrity are also discussed and methods to solve these problems are described.

In Chapter 9, an example, namely to maintain the database for an active badge system, is presented as an application of the prototype implementation to show its feasibility. Experience gained in implementing this application is also given.

Comparisons with other systems addressing the same problems are given in Chapter 10. Chapter 11 concludes the thesis by providing a summary of work done and suggestions for future work.

Chapter 2

Issues on Providing Local Atomicity

An atomic data type, like an ordinary abstract data type, provides a set of objects and a set of operations. As with ordinary abstract types, the operations provided by an atomic data type are the only way for users to access the objects of the type. Unlike regular types, however, an atomic data type needs to provide local atomicity (serialisability and recoverability) for transactions that use objects of the type. In this chapter we discuss some issues arising when providing local atomicity for atomic data types.

2.1 Type-Specific Concurrency Control

The concurrency control protocols discussed in Chapter 1 were developed for simple data types, such as files or relations, with only read and write operations. To support atomic data types that can be defined by users, they must be extended to cope with arbitrary operations. An important aim of implementing atomic data types is to achieve a high degree of concurrency: by taking into consideration the semantics of the operations provided by types, we can allow concurrent executions that would otherwise be forbidden if operations are simply characterised as *reads* and *writes.*

Consider, for example, a bank account abstract data type, that has an associated set of operations: *credit* money to an account, *debit* money from an account, and *check* the balance of an account. These three operations can be partitioned into three classes instead of only *reads* and *writes.* The commutativity between them is shown in Table 2.1. Informally, we say that two operations U and V commute if for all states s, U(V(s)) = V(U(s)). Now

	credit	debit	check
credit		X	X
debit	X	X	X
check	X	X	

Table 2.1: The Commutativity for *Account*

consider the following concurrent execution of two transactions, T_1 and T_2:

T_1 : *credit*(£1000);
T_2 : *credit*(£2000);
T_2 : Commits.
T_1 : Commits.

Since the transaction T_1 and T_2 both issue *credit* operation to increment the balance of the account and since the order of *credit* is insignificant (because of commutativity), the execution above is serialisable. It is not permitted, however, by any of the protocols discussed in Chapter 1, because T_1 and T_2 both update the account (reading the current balance, and then writing a new balance), and so cannot access the account concurrently.

The example above shows a general phenomenon: by describing a system in terms of abstract objects (rather than primitive objects with read and write operations), we can permit greater concurrency than would otherwise be possible. This additional concurrency may be essential for achieving adequate performance in an application. Particularly, in a distributed system, transactions may take a relatively long time to complete; by permitting more concurrent access to objects, we may be able to avoid creating bottlenecks in the system.

2.2 Representation of Semantics

In the last section, the defined commutativity of object operations must be true for all possible parameters, all possible results and all possible object states. However, in many cases, more commutativity can be specified by taking into account operation parameters, operation results and/or object states. Consider the bank account example again. If we take the result of a *debit* operation into account, then an unsuccessful *debit* and a *check* are commutative, and two unsuccessful *debits* are also commutative. The new commutativity is shown in Table 2.2. Now consider the following concurrent execution of two transactions,

	credit	debit(OK)	debit(over)	check
credit		X	X	X
debit(OK)	X	X	X	X
debit(over)	X	X		
check	X	X		

Table 2.2: The New Commutativity for *Account*

T_1 and T_2, supposing the account balance is £500 initially:

T_2 : *debit*(£700) / Overdraw;
T_1 : *debit*(£800) / Overdraw;
T_2 : *check*( ) / £500;
T_2 : Commits.
T_1 : Commits.

The execution above is serialisable according to the new commutativity. It is not serialisable, however, according to the one specified in the last section.

Generally speaking, the more semantics of object operations is considered for specifying commutativity, the more concurrency can be achieved. However, representing and using operation semantics, especially the high level of semantics, is not easy.

Traditionally, the semantics of object operations is either represented in the implementation code of the operations, or represented in a lock compatibility matrix. The former permits a high level of semantics to be specified, but unfortunately it also makes the object operations complicated and difficult to implement. The latter is simple but only permits limited semantics to be specified, thus providing less concurrency.

To lessen the programmer's burden of implementing atomic data types but still permitting great concurrency, it is desirable that programmers can represent the semantics of object operations in a declarative way and that the system makes use of the semantics provided to synchronise concurrent activities.

2.3 Transaction Concurrency and Process Concurrency

As indicated by the previous examples, greater concurrency can be achieved by using operation semantics. However, this requires a more complex implementation. In the examples, the concurrent execution seems to be legal with respect to the standard notion of conflict-based serialisability. In classical concurrency control theory, however, operations are supposed to be elementary, that is, to appear serial and isolated from each other. This basic assumption naturally holds for system operations, that is, *reads* and *writes*, but, may be violated for abstract object operations unless additional measures are provided. In the examples, *credit* is composed of a *read* and a *write*, so it is not elementary. However, two *credits* in the example are allowed to execute concurrently, and thus could interfere with each other on lower level, giving rise to all sorts of anomalies. For example, consider the following synchronisation at the low level:

T_1 : *balance*$_1$ ←*read*( );
T_2 : *balance*$_2$ ←*read*( );
T_1 : *write*(*balance*$_1$ + £1000);
T_2 : *write*(*balance*$_2$ + £2000);
T_2 : Commits.
T_1 : Commits.

The credit of transaction T_1 is lost in the execution above.

Therefore, some form of concurrency control needs to be applied to the low level, so as to ensure that high-level operations can be considered elementary. Thus, the implementation of atomic data types needs to deal with two levels of concurrency: *transaction concurrency* and *process concurrency* [WL85]. The transaction concurrency control mechanism provides appropriate synchronisation and recovery for transactions using objects of the type; while the process concurrency control mechanism copes with internal concurrency and failures.

2.4 Recoverability

Most concurrency control protocols rely on the capability to roll back transactions when a non-serialisable situation is detected, for example, when a deadlock occurs or, in an optimistic approach, when a validation fails. For that reason, it is necessary to consider transaction aborts for a comprehensive treatment of concurrency, even if all transactions involved and the underlying system never fail. Recovery may be provided by a number of techniques[Ver78]. This section investigates to what extent the execution of aborts may influence serialisability in an atomic data type environment.

When a transaction aborts, all the effect of it must be removed. This capability is supported by the property that is termed *recoverability*. Techniques that provide abstraction of recovery generally take one of two forms: either *state* or *operation* based. A state based recovery technique takes a copy of the state of an object before the object is modified. During recovery, the current object state is replaced by the old state. An operation based recovery technique records the operations invoked on an object, enabling the state to be recovered by invoking the inverse operation of each operation recorded.

Both recovery techniques require recovery information, which consists of the recovery data required by the transaction, to be created during operations that modify the state of an object. Each time an object is modified within a new transaction, recovery information must be created and added to the transaction. Some objects are inherently more recoverable than others and more suitable for a particular recovery technique. For example, objects that provide an assignment operation may be recovered using a state based approach that involves saving a copy of the object and reassigning this copy during recovery of the object. An object that provides *increment* and *decrement* operations, however, may be better suited to an operation based approach, as record of the *increment* and *decrement* operations (and parameters) could be used to recover the object by invoking the inverse operation of each operation recorded in the recovery data.

For abstract data types, however, further difficulties turn up when providing recoverability. Again, consider the account example. Since two *credit* operations commute with each other, thus, if *credit* is elementary, there is no reason why two *credit* operations should not be allowed to proceed in parallel with each other. In order to make *credit* become elementary, we suppose that an exclusive lock is held on the object during the execution of *credit*.

Unfortunately this has a disastrous interaction with the recovery system. Consider some object A that initially has the value £2000. If two concurrent transactions T_1 and T_2 both attempt a *credit*(£1000) operation on A, then providing both transactions commit the result would be that A has the value of £4000.

However, consider the following sequences of events. Transaction T_1 sets a *credit* lock on A and changes the value to £3000, in doing so it records the old value as £2000. Similarly, transaction T_2 sets a *credit* lock and sets the value of A to £4000, recording the old value of £3000 (since the inter-operation synchronisation mechanism has allowed the two transactions simultaneous access to the object). T_2 then commits producing £4000 as the final value for A, while T_1 aborts, and thus restores the prior value of A to what it believes it should be, in this case £2000. This also backs out T_2's modification on account A, hence, causes the T_2 to be ineffective, in spite of T_2 being committed.

The reason for this problem is quite obvious: the execution of the transactions is scheduled by two levels of concurrency synchronisation mechanisms. The exclusive locks set by the inter-operation synchronisation mechanism at the lower level are released at the end of each operation, thus if the inter-transaction synchronisation allows non-strict conflicting operations to be executed concurrently, the *isolation* property of transaction is violated. In the example above, the result of T_1 has been seen at the lower level by T_2, hence we cannot simply undo T_1 by means of restoring the prior state recorded by T_1. Neither can we take no action at all; otherwise, T_2 has to abort.

However, it is the intention of atomic data types to allow access to uncommitted low-level data in order to provide greater concurrency. Therefore, we should not resolve the problem by restricting the high-level synchronisation to prevent violating the property of isolation, otherwise, we cannot achieve the aim of greater concurrency by considering operation semantics; instead, we should find other methods to provide recoverability.

The *operation based* technique does not have this problem. We assume that every operation has an *inverse* operation and that a record of invocations is kept with the object. When a transaction aborts, each of its invocations must be undone. For a given object, if there have been no conflicting invocations since the one that is to be undone then we simply apply the inverse operation to the current state. In the example above, the inverse of *credit* is *debit*. When T_1 aborts we can simply *debit*(£1000) and remove the record of the original invocation from the object.

Unfortunately, not all operations have natural inverses. For example, it is unlikely to define an *inverse* for *assign* operation because we cannot deduce the old value of the object from the *assign* operation. The *insert* operation on a set object is another example.

Therefore, suitable methods need to be proposed to provide recoverability for atomic data types that can undo a transaction's effect even if its partial results have been seen by some other transactions; or a more suitable concurrency control method should be proposed that can take advantage of operation semantics to provide greater concurrency but does not violate the *isolation* property.

2.5 Durability

Atomicity alone is not sufficient to provide consistency, because it only concerns running transactions. In addition, the objects must be *resilient*: the probability of loss of data due to a hardware failure such as a node or media crash must be acceptably small. Resilient objects are needed to ensure the durability of transactions, that is, to guarantee that effects of committed transactions are not lost in crashes that occur later.

Various methods of achieving resilience for traditional transactions have been proposed [Koh81, Dat86]. There are two universal rules to be obeyed by any crash recovery mechanism, independently of particular implementation techniques such as logging or shadow versions:

1. Application transactions are guaranteed to appear atomic. Therefore, sufficient UNDO information has to be written to a stable place before modifications are propagated to the permanent database.
2. Application transactions are guaranteed to appear persistent. Therefore, sufficient REDO information has to be written to a stable place before a transaction is definitely committed.

The first rule is often called the "write-ahead-log principle", having its origin in log-based implementation techniques; and the second one may be known as the "committed rule". As a consequence of these rules, the warm-start after a crash requires UNDO operations to be executed for all incomplete transactions, and REDO operations to be executed for all committed transactions, unless the permanent database already reflects the desired state.

In a system that uses atomic data types to provide data consistency, it is atomic objects themselves that are responsible for ensuring data resilience. Therefore, for user defined atomic data types, it is the programmer, not the system, who provides crash recovery. In order to make the work easy, usually the system provides persistent transparency to users by taking a *persistent programming language* [RC89, ACC81, ABC+83] approach. A persistent programming language generally provides data persistence by employing an object store[Low87, EM90]. Objects entrusted to an object store are extremely unlikely to be lost. To provide resilience, objects must be saved in the object store that will not be corrupted by system failures. To move objects to and from the object store requires a mechanism for mapping the volatile state into, and out of, the form expected by the object store system. In a programming system supporting *persistence*, automatic movement of objects to and from object store, and the mapping mechanisms, are provided for objects of each type.

Taking a persistent programming approach to provide data resilience introduces new issues. These will be addressed in Chapter 8.

2.6 Summary

In this chapter, we have shown that although greater concurrency can be achieved by taking account of operation semantics when doing concurrency control, some problems also arise by doing so. First, this makes object operations become more complicated, and consequently, more difficult to implement and more likely to have bugs. Therefore, research should be done on providing suitable methods to permit programmers to represent operation semantics in a declarative way, and leaving the system to synchronise concurrent activities based on the semantics provided. Second, concurrency control needs to be applied at two levels, namely, the transaction level and the process level. The concurrency control methods applied at each level not only need to be correct in themselves but also need to be compatible with each other to guarantee the correctness of the whole protocol. Third, the *isolation* property of transactions is usually violated in such cases to achieve greater concurrency. Therefore, a suitable method needs to be designed to provide recoverability that can undo a transaction's effect even if its partial results have been seen by some other transactions; or a concurrency control method should be proposed that can take advantage of operation semantics to provide greater concurrency but without violating the *isolation* property. Fourth, a system that supports data consistency through atomic data types usually uses a persistent programming language approach to provide data resilience, and new issues arising in such an approach need to be resolved.

Chapter 3

Dual-Level Synchronisation

An atomic data type must provide local atomicity (serialisability and recoverability) for transactions that use objects of the type. In the last chapter, we addressed some issues that arise when implementing atomic data types. In this chapter, we introduce a new concurrency control method that can be used to provide local atomicity for atomic data types.

3.1 Internal Concurrency Control

As we pointed out in the last chapter, when implementing atomic data types, some form of concurrency control needs to be applied to low level objects, so as to ensure that operations on high level objects can be considered elementary. A simple way to ensure an object operation to be elementary is by using *locks* to ensure mutual exclusion among regions of code executed by concurrent operations; thus, for example, implementations can prevent interference among concurrently executing operations by forcing them to run serially.

A problem with the above approach is that it limits concurrency. To ensure the elementary property of object operations, at any one time only one operation is allowed to access the object state by applying mutual exclusive locks. Although a lock only needs to be held during the execution of an operation (much shorter than the locks set for synchronising transactions in the strict two-phase locking protocol, which need to be held until the end of the transaction), it still limits object concurrency considerably if the execution takes a long time.

Another problem with the above approach concerns isolation of the result of a transaction.

Because a lock is held only during the execution of an operation, a transaction's partial results may be seen by other transactions; therefore a special method needs to be followed for preventing *cascading aborts*. For the same reason, a special recovery technique needs to be used for providing recoverability, as indicated in Chapter 2.

This thesis takes a multi-level transaction management [Wei91] approach to provide object atomicity by utilising layer-specific semantics. We view an atomic object as a two-layered architecture. The high layer, called the *logical level*, is the object's interface to users: a set of operations defined on the object, which are the only means for users to access the object. The low layer, called the *physical level*, is the interface between the supporting system and the objects: a set of operations provided by the system to manage primitive objects.

Usually an object is composed of several primitive objects, and operations provided by an object are *composite operations* [Bac93], *i.e.,* an operation consisting of a number of lower level operations, here, the operations provided by the supporting system. Consider the bank account example. At the logical level, an object interface is composed of a set of operations: *create, credit, debit* and *check*. At the physical level, an account object is composed of several primitive objects: a number for the current balance, a string for the account name, and an integer for the account number. These primitive objects can be accessed by system operations: *read* and *write*. System operations are elementary, *i.e.*, the execution of a system operation never appears to overlap (or contain) the execution of any other system operations, even when the operations are run concurrently (serialisability), and the overall effect of a system operation is *all-or-nothing* (recoverability). This assumption is reasonable, for most systems support this property; and even if a system does not support this property, certain methods can be used to provide it [Bac93].

In our approach, an atomic object provides local atomicity by providing two levels of synchronisation: *logical level synchronisation* and *physical level synchronisation*. The logical level synchronisation mechanism schedules transactions that access the object concurrently according to the semantics of object operations. It considers object operations as elementary, *i.e.* to appear serial and isolated from each other. The implementation of object operations is transparent to the logical synchronisation mechanism. It is the responsibility of the physical synchronisation mechanism to ensure the elementary property of object operations. The physical level synchronisation mechanism, in turn, considers the system operations as elementary ones. It classifies them into four kinds: *create, delete, read* and *write*, and synchronises them based on this classification. We call this approach *dual-level synchronisation*.

3.2 Introduction to Dual-Level Synchronisation

Several concurrency control techniques have been discussed in Chapter 1, any of which can be used to implement single level synchronisation. However, as indicated in Chapter 2, these techniques cannot be used directly to provide dual-level synchronisation. Some coordination between the two levels is needed in order to provide object atomicity. In this section, we introduce a concurrency control method, called the *dual-level validation* (DLV) method, which can be used to provide dual-level synchronisation.

3.2.1 Overview

In the DLV method, the synchronisation mechanism at both levels uses an optimistic concurrency control technique, *i.e.*, it allows transactions to execute without synchronisation, relying on commit-time validation to ensure serialisability. Both levels use the *conflict-based validation* technique [Her90], *i.e.* based on predefined conflicts between pairs of operations, for validating transactions.

Our objects are tree structured with primitive objects as leaves. We assume that objects may be large. Our approach is to take a shadow copy only of the subobject of the (tree structured) physical object that is required for a given invocation. A copy of a subobject is taken on the first invocation that updates it and all subsequent invocations on that subobject, for read or update, are performed on that shadow. A later invocation by the transaction may cause a shadow of a different subobject to be taken and this shadow may contain the committed updates of concurrent transactions.

An execution of a transaction consists of two, three or four phases: a *read phase*, a *validation phase*, and possibly a *pending phase* and a *write phase* (See Figure 3.1).

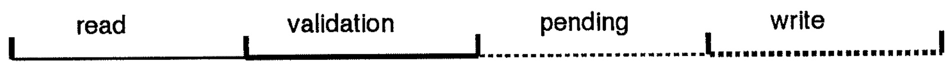


Figure 3.1: The Four Phases of a Transaction

A transaction usually encloses operations on several objects. The sequence of operations of a transaction on a particular object forms the *component* of the transaction at that object.

During the read phase, the transaction manager passes each operation enclosed in a transaction to the appropriate object. The object arranges immediate execution of the operation and records details of the invocation. If the invocation involves an update, this takes place at a local *shadow copy* of the physical subobject as described above. Each object therefore has a record of which object operations have been performed by each transaction, and which physical (sub)objects, together with their version numbers, have been read or written by each transaction. A *performed-operations table* and an *accessed-objects table* are used to record the relevant information.

The validation phase begins when the execution of a transaction reaches the end, *i.e.*, a *Commit* is met[1]. During the validation, the transaction manager first assigns a timestamp to the transaction, and then communicates with every object involved, passing it the transaction identifier and the timestamp. Each object validates its component of the transaction and indicates *accepted* or *rejected.* The aim is to establish whether any of the invocations of the transaction have been invalidated by the invocations of concurrent transactions. This stage is called *logical validation.*

Each accepted component of the transaction enters the *pending* phase with a "waiting" status, while each rejected component is aborted. Note that aborting simply involves discarding the shadow subobjects. The transaction manager then asks every object involved whether the component of the transaction handled at that object is accepted. If all are accepted, the transaction as a whole is committed, otherwise the transaction as a whole is aborted. The transaction manager informs every object involved of the result. If the result is commit, then the component at each object remains in the pending phase but with a new status "commit"; otherwise the component at each object is aborted and removed from the pending queue.

A component of a transaction in the pending phase will not enter the write phase, if the transaction as a whole cannot be accepted. Even if the transaction as a whole is accepted, a component of the transaction does not necessarily enter the write phase immediately after the object gets the final result from the transaction manager. This is because there may be several pending components in an object, and they must enter the write phase in the order defined by their timestamps. Only the component that has the smallest timestamp and is in "commit" status can enter the write phase, and at any time there is at most one component that can be in the write phase in an object.

After entering the write phase, a component of a transaction is validated again by the object to check whether it can be accepted at the physical level. This stage is called *physical validation.* The purpose of this validation is to check whether the values read by the component are still up to date. If they are, the transaction is committed by merging

[1]If a transaction is ended by an *Abort*, then the transaction manager simply asks each object involved to abort the transaction locally.

its shadow copies into the permanent state. Otherwise the shadow copies are discarded and the operations of the component are re-executed. During the re-execution, any update to a physical object takes place in a shadow copy of the object as in the read phase. After the re-execution, shadow copies are merged into the permanent state.

Physical validation can be done by using the information recorded in the *accessed-objects table* based on read/write conflict. The operations of a component can be found in the *performed-operations table.*

3.2.2 Validation Algorithms

From the overview, we can see that validation algorithms play a very important role in the DLV method. In this section, we introduce some concepts related to validation.

Transaction Timestamp

The purpose of logical validation in DLV is to ensure that the concurrent execution of a set of transactions is equivalent to executing these transactions serially in some order. Therefore, an order, say π, must be found, so that the validation algorithm can validate transactions based on π. This can be handled by explicitly assigning each transaction T_i a unique number t_i, called the *timestamp* of the transaction, during the course of its execution. The validation algorithm then ensures that there exists a serially equivalent schedule in which transaction T_i comes before transaction T_j whenever $t_i < t_j$.

When should transactions be assigned timestamps? It depends on the validation algorithm being used. Generally speaking, timestamps should be assigned to transactions as late as possible. For example, suppose that we use a validation algorithm that requires transactions to be validated in the order defined by their timestamps. If we assign timestamps at the beginning of the read phase, this might make a short transaction have to wait for a long transaction unnecessarily. Consider the case of two transactions T_1 and T_2 starting at roughly the same time, assigned timestamps t_1 and t_2, respectively. Then even if T_2 completes its read phase much earlier than T_1, before being validated T_2 must wait for the completion of the read phase of T_1. Instead, we can assign timestamps at the end of the read phase, so that the above problem will not arise.

Validation Conditions

We have pointed out that the aim of a validation algorithm is to ensure that: there must exist a serial schedule in which transaction T_i comes before transaction T_j whenever $t_i < t_j$. This can be guaranteed by the following validation condition [KR81, Pap79]. For each transaction T_j with transaction number t_j, and for all T_i with $t_i < t_j$, one of the following three conditions must hold (see Figure 3.2):

1. T_i completes its write phase before T_j starts its read phase.
2. The operation set of T_i does not invalidate the operation set of T_j, and T_i completes its write phase before T_j starts its write phase.
3. Neither the operation set of T_i invalidates the operation set of T_j nor the operation set of T_j invalidates the operation set of T_i, and T_i completes its read phase before T_j completes its read phase.

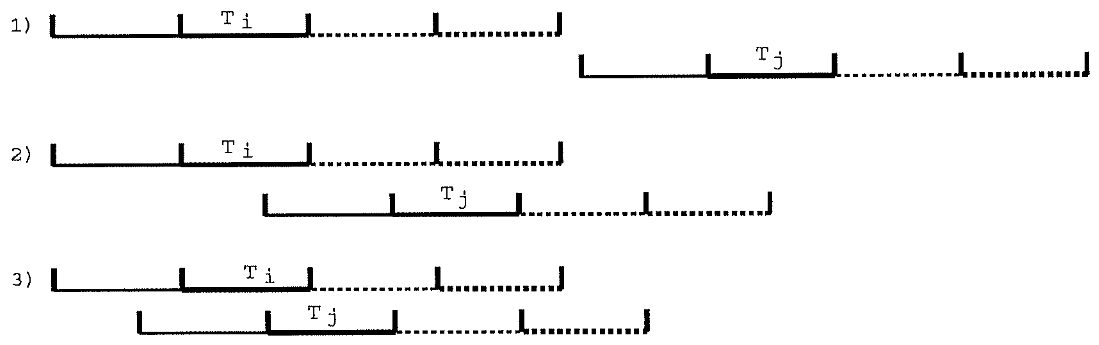


Figure 3.2: Possible Interleaving of Two Transactions

Here, we say an operation set OP_i invalidates an operation set OP_j if and only if there exist at least one operation p in OP_i and one operation q in OP_j such that p invalidates q. Informally, "an operation p invalidates an operation q" means that the result of executing q on an object d might not be the same as the result of executing q after executing p on d. The formal definition will be given in Chapter 4. Note that in our definition "p invalidates q" does not imply that "q invalidates p". For example, a *write* operation invalidates a *read* operation, but a *read* does not invalidate a *write*.

Condition 1 states that T_i actually completes before T_j starts. Condition 2 states that T_i's operations do not affect the read phase of T_j and T_i finishes writing before T_j starts writing, hence does not overwrite T_j (also, note that T_j cannot affect the read phase of T_i). Finally, condition 3 is similar to condition 2 but does not require that T_i finishes writing before T_j starts writing; it simply requires that T_i not affect the read phase or the write phase of T_j (again note that T_j cannot affect the read phase of T_i, by the last part of the condition).

In [KR81], the algorithms that are an implementation of validation conditions 1 and 2 are called *serial validation* algorithms, for the write phases must be serial in such a case. The algorithms that are an implementation of all three conditions are called *parallel validation* algorithms, for then the write phases may take place in parallel. The parallel validation algorithm allows more concurrency, but requires more complicated checking. Note that a serial validation algorithm does not mean that *validation* must be done serially, concurrent validation is allowable.

3.3 Specification of the DLV Method

From the last section, we know that the DLV method involves two types of participant: the transaction manager and atomic objects. They provide data atomicity cooperatively according to a protocol. In this section we specify what functionality should be provided by atomic objects and how it is provided.

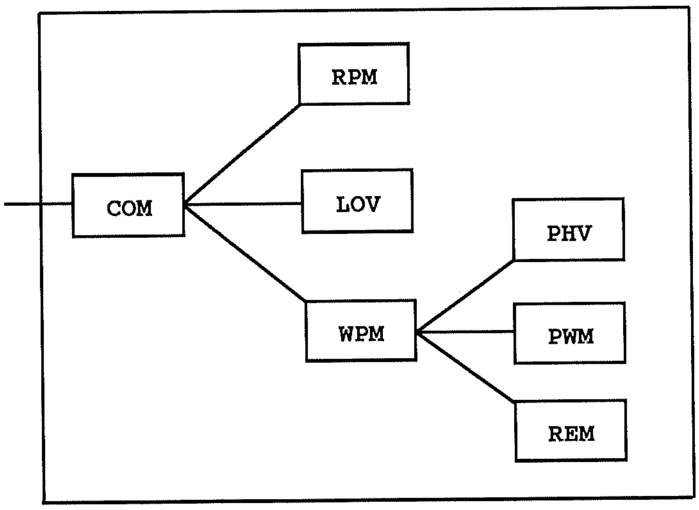


Figure 3.3: The Architecture of Atomic Objects

An atomic object is composed of seven modules: a *COoperation Manager* (COM), a *Read Phase Manager* (RPM), a *LOgical Validator* (LOV), a *Write Phase Manager* (WPM), a *PHysical Validator* (PHV), a *Physical Writing Manager* (PWM), and a Re-Execution Manager (REM) (see Figure 3.3). They work together to ensure the local atomicity of the object.

On the object's side, the communication protocol between the transaction manager and atomic objects proceeds as follows in the absence of failures:

1. *During the read phase*: the COM receives operations from the transaction manager, and then asks the RPM to execute them.

2. *During the validation phase*: after receiving a validation request from the transaction manager, the COM asks the LOV to validate the relevant transaction component, and then returns the result to the transaction manager. If the result is *accepted* the component is put in the pending queue with a "waiting" status, otherwise the component is aborted.

3. *During the pending phase*: after receiving the transaction outcome from the transaction manager, the COM sends an ACK message to it, and then either aborts the component if the outcome is *abort*, or changes the component's status from "waiting" to "commit" if the outcome is *commit*.

4. *During the write phase*: when a component meets the write condition (stated in Section 3.3.3), the COM asks the WPM to write it out. Then the WPM asks the PHV to do a physical validation to the component. If the validation result is *accept*, it asks the PWM to write the component result to the object; otherwise it asks the REM to re-execute the component before asking the PWM to write the result.

It is worth pointing out that all the steps in the protocol are independent actions, there is no requirement for making any two or more of them become an atomic action. This property is important for achieving great object concurrency and availability, especially in a distributed environment (see Section 5.3.2 for details).

3.3.1 The Read Phase Manager

The RPM of an atomic object executes object operations, but any update to a physical object is done on a local shadow copy. For each transaction that shares the object, the RPM creates a *performed-operations table* (POT) to record the operations done by the transaction, together with their parameters and results; and an *accessed-objects table* (AOT), consisting of a *read set* and a *write set*, to record the physical objects read or written by the transaction, together with their version numbers.

After receiving an operation from a transaction through the COM, the RPM records the operation with its parameters into the POT, and then executes the operation. An operation usually calls some system operations to read or write some physical objects. When writing a physical object, the RPM records its OId (Object Identifier) and version number to the write set of the AOT. Whenever the first write to a given physical object is requested, a shadow copy of it is made, and all subsequent operations from that transaction to that object are directed to the shadow copy. A shadow copy of a physical object is local to the transaction for which it is created; it is inaccessible to other transactions. When reading a physical object, the RPM records its OId and version number in the read set of the AOT. If there is a local shadow copy for that physical object, the operation will read from the shadow copy, otherwise the physical object will be read.

3.3.2 The Logical Validator

The LOV of an atomic object is responsible for validating transaction components handled by the object, *i.e.* checking whether committing a component would violate object atomicity.

The validation algorithm used by the LOV is a serial validation algorithm, thus write phases must be serial. A property of the algorithm is that it does not require transaction components to be validated in the order defined by the transaction's timestamps. This property is important for distributed transactions (see Chapter 5 for details).

A serial validation algorithm only uses the first two validation conditions of Section 3.2.2. Since the first validation condition can be checked easily by remembering the latest committed transaction's timestamp when a transaction starts, the main work of the validation algorithm is to check whether validation condition 2 holds, which is done by the following three checks (suppose transaction T_j is under validation):

- *check 1*: For every transaction T_i that is *older* than T_j and had not committed when T_j began, check whether the operation set of T_i invalidates the operation set of T_j; if it does, the validation fails.
- *check 2*: For every transaction T_k that is in its pending phase and is *younger* than T_j, check whether the operation set of T_j invalidates the operation set of T_k; if it does, the validation fails.
- *check 3*: check whether any committed transaction T_k is *younger* than T_j; if any T_k is, the validation fails.

Check 1 and *check 2* ensure that the first part of validation condition 2 holds, *i.e.*, the operation set of an older transaction does not invalidate the operation set of a younger

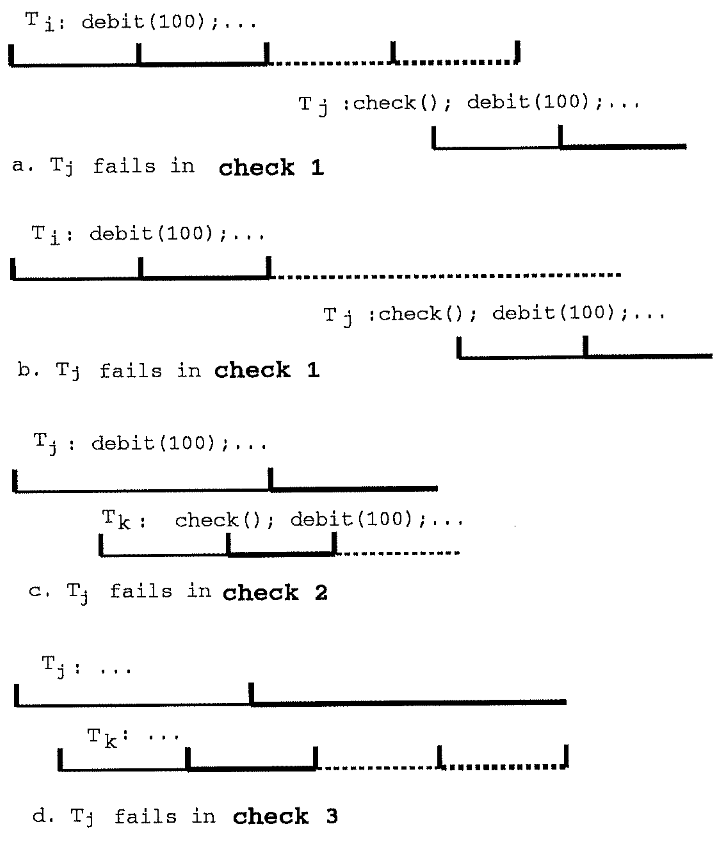


Figure 3.4: Possible Validation Failures ($t_i < t_j < t_k$)

transaction. *Check 3* ensures that the second part of validation condition 2 holds, *i.e.*, transactions are committed in the order defined by their timestamps.

A transaction T_j will have seen the results of transactions that completed their write phase before it began. T_j, however, might not see the whole result of transactions that completed their write phase after it began, thus they may invalidate T_j(see Figure 3.4 a). Furthermore, since T_j cannot see the results of any pending transactions, they may also invalidate T_j if they are older than T_j (see Figure 3.4 b). It is the responsibility of *check 1* to check whether the above situations have happened.

Because the validation algorithm does not require transactions to be validated in their timestamp order, it is possible that a transaction T_k finishes its validation earlier than a transaction T_j, where T_j is older than T_k (see Figure 3.4 c). Therefore, when validating T_j, the algorithm needs to check whether T_j invalidates T_k in order to ensure that the first part of validation condition 2 holds, which is the responsibility of *check 2*.

For the same reason as above, it is possible that before a transaction T_j finishes its validation, a younger transaction T_k has committed (see Figure 3.4 d); thus T_j has to be rejected to ensure that the last part of validation condition 2 holds. This is the responsibility of *check 3*.

Check 1 and *check 2* are done by using the information recorded in the *performed-operations tables*. *Check 3* is done by comparing the timestamp of the transaction being validated with the timestamp of the latest committed transaction.

A transaction that fails its validation is aborted, while a transaction that succeeds is put in the *pending queue* waiting for completion. The pending queue is organised in the order defined by the timestamps of transactions.

3.3.3 The Write Phase Manager

The WPM of an atomic object is responsible for ensuring that transactions are committed in the order defined by their timestamps. The WPM guarantees this property by ensuring that a transaction component can enter the write phase only when the following write conditions hold:

1. the component is at the head of the pending queue, thus no older transactions are waiting for committing;
2. the component is in "commit" status, *i.e.*, the transaction manager has decided to commit the transaction;
3. there is no other transaction in the write phase at this particular object.

The WPM commits a transaction component by coordinating the PHV, the PWM and the REM. At first, it asks the PHV to validate the transaction component to see whether the component can be accepted at the physical level. Then, if the validation result is *accept*, it asks the PWM to do the writing. Otherwise, *i.e.* if the result is *reject*, it asks the REM to re-execute the component and then asks the PWM to do the writing. The operations of the component can be found in the *performed-operations table* for the transaction.

The Physical Validator

The DLV method permits only one transaction at any time to be in the *write phase* in a particular object, and transactions enter the write phase in their timestamp order. Therefore, to validate a transaction T_j, the PHV only needs to check whether any transaction T_i, that committed after T_j began, invalidates T_j.

As we have seen, only four kinds of operation are recognised at the physical level: *create*, *delete*, *read* and *write*. Among them only a *write* operation invalidates a *read* operation in such a case. Therefore, to check whether T_i invalidates T_j, the PHV only needs to check whether the write set of T_i intersects with the read set of T_j.

The physical validation can also be done by using the version number technique. Each physical object is associated with a version number, which starts from zero and increases whenever the physical object is updated. When a transaction reads a physical object during the read phase, the RPM records the object together with its version number in the read set. To validate a transaction T_j, the PHV compares, for each physical object in the read set, its version number in the read set with its current version number to see whether the object has been updated since the read. If any physical object has been updated, the validation fails; otherwise, it succeeds.

The Physical Write Manager

The PWM of an object is responsible for writing the results of a transaction component into the object atomically. It writes the results by overwriting the original physical objects with the "shadow copies" created for the transaction component. The "shadow copies" are then discarded.

The overwriting operation must be done atomically even in the presence of system crashes, otherwise the object may be brought into an inconsistent state, *i.e.*, some of the "shadow copies" may be written out but others not.

The Re-Execution Manager

The REM of an object is responsible for re-executing a transaction component. The operations of a transaction component and also their parameters are recorded in the *performed-operation table*; they can therefore be re-executed.

Like the RPM, when re-executing a transaction component, the REM also causes any update to a physical object to be done on a "shadow copy". Unlike the RPM, however, the REM does not record any information in the *performed-operations table* or *accessed-objects table.*

3.4 Summary

This section concludes the chapter by evaluating the DLV method and comparing it with a related method [Her90].

3.4.1 Evaluation

An important issue when implementing atomic data types is how to deal with low-level concurrency to guarantee that operation executions appear serial and isolated from each other. The transaction scheduler can then use the semantics of high-level operations for synchronising transactions to achieve great concurrency. The DLV method resolves the issue by applying synchronisation at two levels. A logical level synchronisation mechanism schedules transactions according to the semantics of operations, while a physical level synchronisation mechanism ensures the elementary property of operations by scheduling its accesses to physical objects based on the *read/write* conflict.

The DLV method separates conflicts between transactions into two levels: logical level and physical level. Whether two transactions conflict at the logical level depends on the semantics of operations enclosed in the transactions. Whether two transactions conflict at the physical level depends on what physical objects they have accessed. Generally speaking, transactions that conflict at the logical level must conflict at the physical level; however, transactions that conflict at the physical level may not conflict at the logical level. Therefore, if we can solve physical level conflicts at that level, fewer transactions need to be aborted. The REM is provided in the DLV method to ensure that the physical level conflicts can be solved locally, without causing transaction aborts. This saves much system resource and improves system performance when a transaction involves several objects, especially when some of them are remote ones.

The recoverability problem mentioned in Chapter 2 is automatically solved in the DLV method, for the result of a transaction is isolated from other transactions until it is committed.

3.4.2 Comparison

The DLV method is very similar to the method proposed by Herlihy [Her90] in that both of them utilise the semantics of operations to validate more interleavings and both of them can be used to implement atomic data types. However, there is a large difference between the two approaches. Herlihy's method represents the partial results of a transaction component by a snapshot of the object's permanent state plus an intentions-list, and commits a transaction component by applying the intentions-list to the current permanent state serially. The DLV method, on the other hand, represents the partial results of a transaction component by a group of "shadow copies" of physical objects, and commits a transaction component by writing the shadow copies back.

A drawback of Herlihy's method is that when taking a snapshot of an object's state for a transaction, it needs to create a whole copy of the object state, even if the transaction only accesses a small part of it, thus wasting system resource. The DLV method can create shadow copies at any level of an object's structure, so keeping shadow copies as small as possible. Another drawback of Herlihy's method is that it takes more time and uses more system resources than the DLV method when committing a transaction component, for it needs to re-execute the intentions-list, while the DLV method usually needs only to write the shadow copies back. Furthermore, since applying the intentions-list needs to be done serially, if there are several transaction components waiting to commit, some of them may need to wait for a long time.

An important issue in the DLV method is that transactions cannot be committed by writing shadow copies back without checking, for transactions may conflict at the physical level. Thus, a physical validation phase is required in the DLV method. Transactions that pass the physical validation are committed directly by writing the shadow copies back. Transactions that fail the physical validation need to be re-executed locally before being committed.

How much performance can be gained by the DLV method depends on the success rate of physical validation. The more the transactions that succeed in it, the greater the gain in performance. Since the DLV method permits shadow copies to be created at any level of an object's structure, it reduces the possibility of physical confliction greatly, thus making more transactions pass the physical validation.

Chapter 4

Formalisation and Correctness

In the last chapter, we introduced a concurrency control technique for atomic data types, the DLV method. In this chapter, we formalise this method, and verify that it is atomic, *i.e.*, every schedule produced by it is atomic. The formal method developed by Weihl [Wei89] and the *serial dependency relation* introduced by Herlihy [Her90] are used in this formalisation and verification process.

As we pointed out in the last chapter, there is a large difference between the DLV method and Herlihy's method. Herlihy's method uses abstract objects as the granularity of object access. A snapshot of an (abstract) object state is taken for a transaction at the first time it accesses the object, and all subsequent operations of the transaction on the object are performed on the snapshot. Therefore, the operations of a transaction on an object are performed based on the same permanent state of the object. However, the DLV method uses physical objects as the granularity of object access. A shadow copy of a physical object is created for a transaction at the first time that it intends to update the physical object. Its operations on the physical object are then performed on the shadow copy. Since shadow copies of different physical objects are created at different times, it is possible that they do not come from the same permanent state. Therefore, different operations of a transaction on the same abstract object may be performed based on different permanent states of it. This difference between the two methods makes the DLV method permit more internal concurrency and take less commitment time than Herlihy's method, but also makes it more difficult to formalise and verify; thus new concepts need to be introduced.

4.1 Assumptions and Definitions

Each object has a type which defines a *state* and a set of *operations*. The operations are the only means available for users to create and manipulate objects of that type. An *event* is a pair consisting of an operation invocation and a response.

In the absence of failure and concurrency, an object's state is modelled by a sequence of events called a *history*. For example, the following sequence of events h is a history for an Account object defined in Chapter 1.
credit(£1000) / OK
credit(£1000) / OK
debit(£1500) / OK
debit(£700) / Overdrawn

A specification for an object is the set of permissible histories for that object. For example, the specification for an Account object should contain the following event sequence:
credit(£1000) / OK
debit(£700) / OK

but should not contain the following event sequence:
check() / £2000
debit(£1500) / overdrawn

A *legal history* is one that is included in the object's specification. The history h above is a legal history. A *subhistory* of a history h is a subsequence of events of h. Histories are denoted by lower-case letters in this thesis.

In the presence of failure and concurrency, an object's state is given by a *schedule*, which is a sequence of operation executions (*events*), *transaction commits*, and *transaction aborts*. To keep track of interleaving, a transaction identifier is associated with each step in a schedule. For example, the following is a schedule for an Account object:
T_1: credit(£800) / OK
T_2: credit(£1000) / OK
T_1: commit
T_2: debit(£1500) / OK
T_2: commit

Here, T_1 and T_2 are transaction identifiers. The ordering of operations in a schedule reflects the order in which the object returned responses, not necessarily the order in which it received invocations.

(Serial) histories and (concurrent) schedules are related by the notion of *atomicity.* Let $\ll$ denote a total order on committed and active transactions, and let H be a schedule (schedules are denoted by upper-case letters in this thesis). The *serialisation* of H in the order $\ll$ is the history h constructed by reordering the events in H so that if $T_1 \ll T_2$ then the subsequence of events associated with T_1 precedes the subsequence of events associated with T_2. H is *serialisable in order* $\ll$ if h is legal. The schedule in the example above is serialisable in order $T_1 \ll T_2$, for the following history is legal:
credit(£800) / OK
credit(£1000) / OK
debit(£1500) / OK

However, it is not serialisable in order $T_2 \ll T_1$, for the following history is illegal (suppose the account balance is zero initially):
credit(£1000) / OK
debit(£1500) / OK
credit(£800) / OK

H is *serialisable* if it is serialisable in some order. H is *atomic* if the subschedule associated with *committed transactions* is serialisable. It is obvious that the schedule above is atomic. An object is atomic if it only produces atomic schedules.

A transaction usually encloses operations on several objects. The sequence of operations of a transaction on a particular object forms the *component* of the transaction at that object.

4.2 Serial Dependency

We wish to take account of all events that might, directly or indirectly, have influenced e. Let $<_d$ be a relation between pairs of events, and let h be a history. A subhistory (*i.e.* a subsequence) g of h is *closed* under $<_d$ if whenever it contains an event e it also contains every event e' of h such that $e' <_d e$. A subhistory g is a *view* of h for e under $<_d$ if g is closed under $<_d$, and if g contains every e' of h such that $e' <_d e$.

Again, consider the Account example, suppose we have the following relation $<_d$:
$debit/OK <_d debit/OK$
$credit/OK <_d debit/Over$
$debit/OK <_d check$
$credit/OK <_d check$

and a history h:
credit(£1000) / OK
debit(£500) / OK
check() / £500
debit(£100) /OK
credit(£200) /OK
check() /£600

Then, the following subhistory g is closed under $<_d$:
credit(£1000) / OK
debit(£500) / OK
debit(£100) /OK
credit(£200) /OK
check() /£600

so is the following subhistory:
debit(£500) / OK
debit(£100) /OK
credit(£200) /OK

although it is an illegal subhistory; while the following subhistory is not:
credit(£1000) / OK
debit(£100) /OK
credit(£200) /OK

for it does not contain debit(£500) which is required by debit(£100), for debit(£500) $<_d$ debit(£100).

Suppose there is an event e:
debit(£300) / OK

Then, the subhistory g above is a *view* of h for e under $<_d$, for g is closed under $<_d$ and contains every *debit* event of h.

Informally, $<_d$ is a *serial dependency relation* if whenever an event is legal for a view, it is legal for the complete history. More precisely, let " $\bullet$ " denote concatenation:

Definition 1 A relation $<_d$ is a *serial dependency relation* if $g \bullet e$ is legal implies that $h \bullet e$ is legal, for all events e and all legal histories g and h, such that g is a view of h for e under $<_d$.

It is not easy to check whether a given relation is a serial dependency relation according to the definition. Fortunately, Herlihy [HW88] has proved that an *invalid-by* relation must be a serial dependency relation, and it is much easier to check whether a given relation is an *invalid-by* relation.

Informally, we say that an operation p invalidates an operation q means that the result of executing q on an object d might not be the same as the result of executing q after executing p on d. More precisely,

Definition 2 *Invalid-by relation* $<_c$ is a relation between pairs of events. Let h be a legal history, e and e' be events, we say that $e <_c e'$ (e invalidates e') if and only if that $h \bullet e$ and $h \bullet e'$ are legal, but $h \bullet e \bullet e'$ might be illegal.

The legal histories for an object is related to the semantics of the object, and the invalid-by relation for an object depends on its legal histories. Therefore, changing the semantics of an object would affect its invalid-by relation.

In the Account example above, the relation $<_d$ is an *invalid-by* relation, thus a *serial dependency relation.* Where successful, *debits* do not depend on prior *credits*, because the *debit* cannot be invalidated by increasing the balance. Attempted overdrafts, however, do depend on prior credits, because the *overdraft* exception can be invalidated by increasing the balance. Successful *debits* do depend on prior successful *debits*, because the *debit* can be invalidated by decreasing the balance. Attempted overdrafts, however, do not depend on prior *debits*, because the *overdraft* exception cannot be invalidated by decreasing the balance. *Checks* depend on both *credits* and successful *debits*, because the result of *check* can be invalidated by any change in the balance.

We make extensive use of the following lemma introduced and proved in [Her90] when reasoning about serial dependency relations. It states that any sequence of events can be inserted into the middle of a history provided no later event depends on any inserted events.

Lemma 1 *If $<_d$ is a serial dependency relation, f, g and h histories such that $f \bullet g$ and $f \bullet h$ are legal, and there is no e in g and e' in h such that $e <_d e'$, then $f \bullet g \bullet h$ is legal.*

Proof: The proof is by induction on the length of h. If h is empty, the result is immediate. Otherwise, let $h = h' \bullet e'$. By assumption, $f \bullet h'$ is a view of $f \bullet g \bullet h'$ for e'. Moreover, $f \bullet g \bullet h'$ is legal by the inductive hypothesis and $f \bullet h'$ is legal because $f \bullet h$ is legal by assumption and an initial subsequence of a legal history is legal. Because $f \bullet g \bullet h'$ is legal and $<_d$ is a serial dependency relation, $f \bullet g \bullet h' \bullet e' = f \bullet g \bullet h$ is legal by Definition 1.□

4.3 Views of Transactions

Internally, an object is implemented by two components: a *permanent state* that records the effect of committed transactions, and a set of *local versions* (shadow copies) that record each active transaction's tentative changes. When a transaction commits, its local version is merged into the permanent state. At any time, there is at most one transaction that is in the write phase at a particular object. However, there may be several transactions that are in the read phase at a time. A transaction in its read phase writes to its local version of physical objects, and reads either from its local version or from the permanent state. The permanent state of an object can only be changed by transaction commits. A transaction may see different permanent states of an object in its read phase, for some transactions may commit and change the permanent state in this period.

4.3.1 Translation Function

We formalise an operation on an object as a function that reads from some primitive physical objects (maybe none), and based on the results of reading and its parameters writes to some primitive physical objects (maybe none). We use $o(R, W)$ to denote an operation, which reads from a set of primitive physical objects $R = [r_0, \ldots, r_m]$ and writes to a set of primitive physical objects $W = [w_0, \ldots, w_n]$.

Let $T = [T_0, \ldots, T_l]$ be a set of transactions, d_u be a primitive physical object. We use d_u^i to denote a specific version i (created for transaction T_i) of d_u, and d_u^p to denote its permanent state. To process operations from T_i, an object must translate an operation of T_i on a (single version) primitive physical object into an operation on a specific version of that physical object. This translation is formalised by a function *tr*.

Definition 3 Let $o(R, W)$ be an operation from transaction T_i, $R = [r_0, \ldots, r_m]$, $W = [w_0, \ldots, w_n]$, $R' = [r_0^{j_0}, \ldots, r_m^{j_m}]$, $W' = [w_0^{k_0}, \ldots, w_n^{k_n}]$. Then $tr(o(R, W)) = o(R', W')$, where

1. $k_u = i$ for $0 \leq u \leq n$;

2. $j_u = i$ for $0 \leq u \leq m$, if an operation of T_i has written to w_u;

3. $j_u = p$ for $0 \leq u \leq m$, if no operation of T_i has written to w_u.

Rule 1 states that a transaction can only write to its own version of physical objects. Rule 2 states that if a version of a physical object has been created for a transaction, then it must read from its version. Rule 3 states that if a transaction has not written to a physical object, it must read from the current permanent state of the object.

4.3.2 Views of a Transaction

The permanent state of an object may be changed at any time, therefore different operations of a transaction may view different permanent states from each other. Since a transaction can only affect the permanent state of an object during the write phase in the DLV method, the change in the permanent state between the execution of two operations of a transaction can be modelled by the sequence of events caused by the transactions that commit during this period of time. Therefore, the permanent state of an object when executing an operation can be represented by the initial state of the object followed by a sequence of changes. More precisely,

Definition 4 Let $C = o_1 \bullet o_2 \cdots \bullet o_n$ be the component of transaction T_i at an object and h_j denote the change in the permanent state of the object between the execution of o_j and o_{j+1}, then the permanent state of the object when executing o_j $(1 \leq j < n)$ is $ps_j(T_i) = h_0 \bullet \cdots \bullet h_{j-1}$, where h_0 is the initial state of the object before executing o_1.

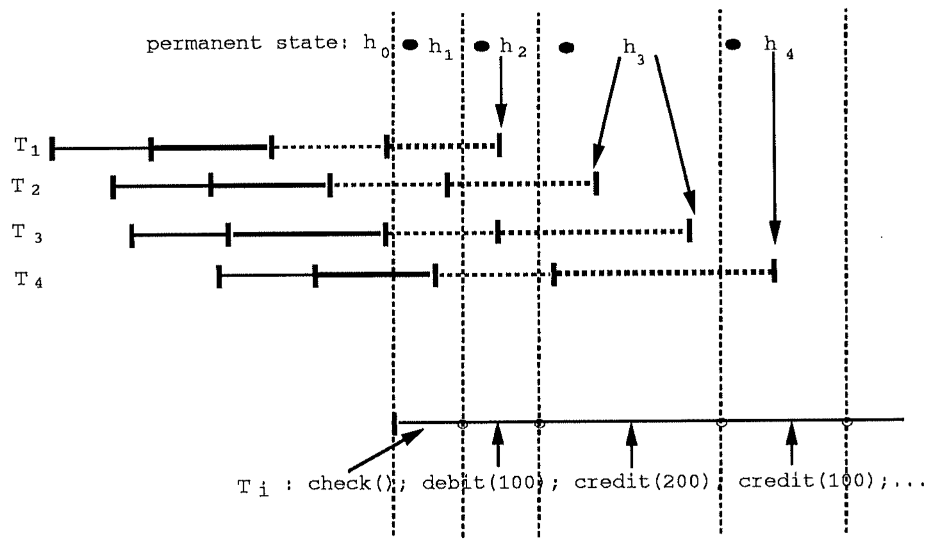

permanent state: h0 ● h1 ● h2 ● h3 ● h4
T1
T2
T3
T4
Ti : check(); debit(100); credit(200); credit(100); . . .

Figure 4.1: Permanent States When Executing Operations

Consider the example in Figure 4.1, the permanent state is h_0 (the initial state) when executing *check*(), is $h_0 \bullet h_1$ (h_1 is empty) when executing *debit*(100), is $h_0 \bullet h_1 \bullet h_2$ (h_2 is composed of the events of T_1) when executing *credit*(200), is $h_0 \bullet h_1 \bullet h_2 \bullet h_3$ (h_3 is composed of the events of T_2 and T_3) when executing *credit*(100).

An object state consists of a set of primitive physical objects, the leaves in the tree structure. A change on an object D made by an operation can be represented by corresponding changes on *D's* primitive physical objects. A read from D issued by an operation can be represented by corresponding reads from *D's* primitive physical objects. If we denote an object D consisting of a set of physical objects $d_1 \cdots d_m$ as:

$$D = \begin{pmatrix} d_1 \\ \vdots \\ d_m \end{pmatrix},$$

then an operation o on D can be represented by a group of operations on the primitive physical objects:

$$o = \begin{pmatrix} o^1 \\ \vdots \\ o^m \end{pmatrix},$$

where o^i is an operation on physical object d_i. We call o^i a suboperation of o on physical object d_i.

An event e on D can be represented by a group of events:

$$e = \begin{pmatrix} e^1 \\ \vdots \\ e^m \end{pmatrix},$$

where e^i is a subevent of e on d_i. Furthermore, a history h for D can be denoted as:

$$h = e_1 \bullet \cdots \bullet e_n = \begin{pmatrix} e_1^1 \bullet \cdots \bullet e_n^1 \\ \vdots \\ e_1^m \bullet \cdots \bullet e_n^m \end{pmatrix} = \begin{pmatrix} h^1 \\ \vdots \\ h^m \end{pmatrix}.$$

We call $e_1^i \bullet \cdots \bullet e_n^i$ the subhistory of h on physical object d_i, denoted by h^i.

The history for a physical object d_i, $h^i = e_1^i \bullet \cdots \bullet e_n^i$, is *locally legal* if it is the same as executing the corresponding operations on d_i in a sequential environment in the same order.

Lemma 2 *Suppose object D consists of a set of primitive physical objects $d_1 \cdots d_m$, $h = e_1 \bullet \cdots \bullet e_n$ is the history for D. Then h is legal if and only if every subhistory of h for each primitive physical object is locally legal.*

Proof: At first we prove that if h is legal, then every subhistory of h for a primitive physical object is locally legal. Since $h = e_1 \bullet \cdots \bullet e_n$ is legal, h must be a permissible history for the object. That is, h must be the same as executing the corresponding operations on D in a sequential environment. Hence, every subhistory h^j of h must be the same as executing the corresponding suboperations on d_j in a sequential environment. Therefore, h^j is locally legal by its definition.

Now let's prove that if every subhistory of h for a primitive physical object is locally legal, then h is legal. Since every subhistory, $h^j = e_1^j \bullet \cdots \bullet e_n^j$, is locally legal, it is the same as executing the corresponding suboperations on d_j in a sequential environment in the same order. Moreover, e_i is the composite of $e_i^1 \cdots e_i^m$. Therefore, h must be the same as executing the corresponding operations on D in a sequential environment. Hence, h must be legal.□

A *view* of a transaction T_i for an (abstract) object, denoted by View(T_i), is the value of the object that T_i observes at some moment. It is a composite of the values of primitive physical objects. From Definition 3, we know that the value of a primitive physical object seen by a transaction is either from its local version, or from the permanent state of the object. More precisely,

Definition 5 Let $C = o_1 \bullet \cdots \bullet o_n$ be the component of transaction T_i at an object and e_j be an event of executing operation o_j $(1 \leq j \leq n)$, suppose that the object state consists of primitive physical objects $d_1 \cdots d_m$ and we have $ps_j(T_i) = h_0 \bullet \cdots \bullet h_{j-1}$. Then after executing operation $o_j(1 \leq j \leq n)$, we have

$$\text{View}(T_i) = \begin{pmatrix} v^1 \\ \vdots \\ v^m \end{pmatrix} = \begin{pmatrix} h_0^1 \bullet \cdots \bullet h_{k_1-1}^1 \bullet e_{k_1}^1 \bullet \cdots \bullet e_j^1 \\ \vdots \\ h_0^m \bullet \cdots \bullet h_{k_m-1}^m \bullet e_{k_m}^m \bullet \cdots \bullet e_j^m \end{pmatrix}.$$

where e_u^l is the subevent of e_u on physical object d_l, $(1 \leq l \leq m, k_l \leq u \leq j)$. Here, $e_{k_l}^l$ $(1 \leq l \leq m)$ is the first subevent that writes physical object d_l. We call v^l the subview of View(T_i) on physical object d_l.

Note that a view of a transaction for an object is different from the permanent state of that object, because a view of a transaction is a mixture of local (physical) objects and permanent physical objects. Consider the example in Figure 4.2: T_i's view is composed of, at t_1, one local object d_1' and two permanent objects d_2 and d_3; at t_2, two local objects d_1' and d_3' and one permanent object d_2.

Lemma 3 *Suppose* View*(T_i) is a view of transaction T_i for an object. Then every subview of it is locally legal, if the permanent state of the object is legal.*

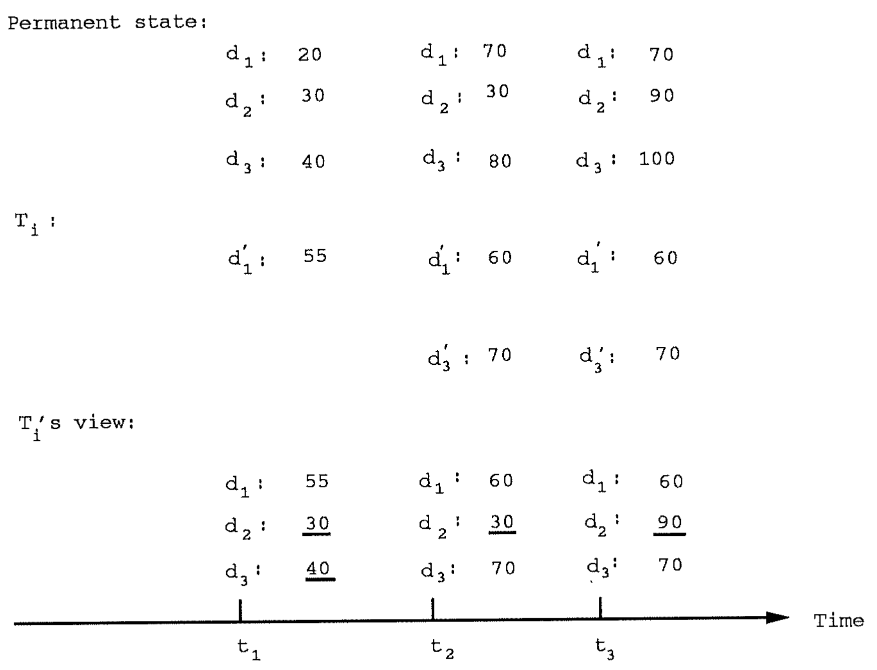


Figure 4.2: Views of a Transaction

Proof: Let's consider the subview for d_l, $v^l = h_0^l \bullet \cdots \bullet h_{k_l-1}^l \bullet e_{k_l}^l \bullet \cdots \bullet e_j^l$. Since the permanent state of an object is legal by assumption, $h_0^l \bullet \cdots \bullet h_{k_l-1}^l$ is locally legal. By Definition 3, we know that when executing operation o_{k_l}, the object creates a local version of d_l, which has the value $h_0^l \bullet \cdots \bullet h_{k_l-1}^l$, and all the subsequent operations of T_i on d_l are done on this local version. Since the local version can only be accessed by T_i, the event sequence $e_{k_l}^l \bullet \cdots \bullet e_j^l$ happened in a sequential environment. Therefore, $v^l = h_0^l \bullet \cdots \bullet h_{k_l-1}^l \bullet e_{k_l}^l \bullet \cdots \bullet e_j^l$ is locally legal.□

4.3.3 An Important Lemma

Now we can get the lemma that is important to the proof of the DLV method.

Lemma 4 *Let $<_d$ be a serial dependency relation, $C = o_1 \bullet \cdots \bullet o_n$ be the component of transaction T_i at an object, e_j be the execution of o_j, h_0 be the object state before e_1, h_j $(1 \leq j < n)$ be the change that happened at the object between e_j and e_{j+1}, h_n be the change that happened after e_n. Then $h_0 \bullet \cdots \bullet h_n \bullet e_1 \bullet \cdots \bullet e_n$ is legal, if $h_0 \bullet \cdots \bullet h_n$ is legal and if there is no e in $h_1 \bullet \cdots \bullet h_n$ and e' in $e_1 \bullet \cdots \bullet e_n$ such that $e <_d e'$.*

Proof: The proof is by induction on the length of C, that is, the number of its operations. If the length $n = 1$, by Definition 3, $h_0 \bullet e_1$ is legal. By assumption, $h_0 \bullet h_1$ is legal and there is no event e in h_1 such that $e <_d e_1$. That is, h_0 is a view of $h_0 \bullet h_1$ for e_1. Therefore, $h_0 \bullet h_1 \bullet e_1$ is legal by Definition 1.

If the length $n > 1$, then

1. (a) by assumption, $\underbrace{h_0 \bullet \cdots \bullet h_{n-1}}_{f} \bullet \underbrace{h_n}_{g}$ is legal,

   (b) by the inductive hypothesis, $\underbrace{h_0 \bullet \cdots \bullet h_{n-1}}_{f} \bullet \underbrace{e_1 \bullet \cdots \bullet e_{n-1}}_{h}$ is legal, and

   (c) by assumption, there is no event e in $g = h_n$ and e' in $h = e_1 \bullet \cdots \bullet e_{n-1}$ such that $e <_d e'$.

   Therefore, $p = \underbrace{h_0 \bullet \cdots \bullet h_{n-1}}_{f} \bullet \underbrace{h_n}_{g} \bullet \underbrace{e_1 \bullet \cdots \bullet e_{n-1}}_{h}$ is legal by Lemma 1. Consequently, every $p^j = h_0^j \bullet \cdots \bullet h_n^j \bullet e_1^j \bullet \cdots \bullet e_{n-1}^j$ is locally legal by Lemma 2.

2. (a) By Definition 5, there is no event e^j in $e_1^j \bullet \cdots \bullet e_{k_j-1}^j$ that writes d_j. Therefore, there is no event e^j in $e_1^j \bullet \cdots \bullet e_{k_j-1}^j$ such that $e^j <_d e_n^j$. This is because in the physical level there is only one serial dependency relation: *write* $<_d$ *read.*

   (b) By assumption, there is no e in $h_{k_j} \bullet \cdots \bullet h_n$ such that $e <_d e_n$. Notice that if nonsense can arise at a physical level, the user must declare the potential nonsense in the abstract semantics. Hence we know that there is no e^j in $h_{k_j}^j \bullet \cdots \bullet h_n^j$ such that $e^j <_d e_n^j$.

   Therefore, $q^j = h_0^j \bullet \cdots \bullet h_{k_j-1}^j \bullet e_{k_j}^j \bullet \cdots \bullet e_{n-1}^j$ is a view of p^j for e_n^j.

3. Since $h_0 \bullet \cdots \bullet h_n$ is legal by assumption, every subview of View(T_i) is locally legal by Lemma 3. Hence $v^j = q^j \bullet e_n^j = h_0^j \bullet \cdots \bullet h_{k_j-1}^j \bullet e_{k_j}^j \bullet \cdots e_n^j$ is locally legal.

Therefore, $p^j \bullet e_n^j = h_0^j \bullet \cdots \bullet h_{n-1}^j \bullet h_n^j \bullet e_1^j \bullet \cdots \bullet e_{n-1}^j \bullet e_n^j$ for $1 \leq j \leq m$ is locally legal by Definition 1. Hence $h_0 \bullet \cdots \bullet h_n \bullet e_1 \bullet \cdots \bullet e_n$ is legal by Lemma 2. □

4.4 The Dual Level Validation Automaton

Formally, each object is modelled by an automaton that accepts certain schedules. The automaton's state is defined by using the following primitive domains: TRANS is the set of transaction identifiers, DIDS is the set of physical object identifiers, EVENTS is the set of events, and TIMESTAMP is a totally ordered set of timestamps. The derived domain HISTORY is the set of sequences of events. A *dual-level validation automaton* has the following state components:

Perm:	HISTORY
View:	TRANS $\rightarrow$ HISTORY
Intentions:	TRANS $\rightarrow$ HISTORY
ReadSet:	TRANS $\rightarrow$ DIDS
ReadVersion:	(TRANS, DIDS) $\rightarrow$ TIMESTAMP
WriteSet:	TRANS $\rightarrow$ DIDS
WriteVersion:	(TRANS, DIDS) $\rightarrow$ TIMESTAMP
TimeStamp:	TRANS $\rightarrow$ TIMESTAMP
BeginTime:	TRANS $\rightarrow$ TIMESTAMP
CommitTime:	TRANS $\rightarrow$ TIMESTAMP
Version:	DIDS $\rightarrow$ TIMESTAMP
LastCommitTime:	TIMESTAMP
Clock:	TIMESTAMP
Committed:	2^{TRANS}
Aborted:	2^{TRANS}

Perm is the history that represents the object's permanent state, initially empty. *View*(T_i) is the view of transaction T_i for the object. *Intentions*(T_i) is the history of operations executed by transaction T_i. $ReadSet(T_i)$ is a set of primitive physical objects that are read by transaction T_i, and $ReadVersion(T_i, d)$ indicates the version from which physical object d is read. $WriteSet(T_i)$ is a set of primitive physical objects that transaction T_i intends to write, and $WriteVersion(T_i, d)$ indicates the version from which the shadow copy of physical object d is created. $Version(d)$ is the version number of physical object d in the permanent state. *TimeStamp*(T_i) is the transaction timestamp assigned by the transaction manager to transaction T_i. *LastCommitTime* is the transaction timestamp of the latest committed transaction.

The DLV automaton enforces the atomicity of schedules generated at the object. Its behaviour is specified by giving the transitions during the read phase and the write phase. Each transition has a precondition and a postcondition. In postconditions, primed component names denote new values, and unprimed names denote old values.

At the read phase:

For a transaction T_i to execute operation *o(R, W)* at an object, where $R = [r_1, \cdots, r_m]$, $W = [w_1, \cdots, w_n]$:

Pre: $T_i \notin$ Committed $\cup$ Aborted.
e is the event of executing $o(R, W)$.

Post:

$$\text{View}'(T_i) = \text{View}(T_i) \uplus o(R, W)$$
$$\text{Intentions}'(T_i) = \text{Intentions}(T_i) \bullet e$$
$$\text{ReadSet}'(T_i) = \text{ReadSet}(T_i) \cup [r_1, \cdots, r_m]$$
$$\text{WriteSet}'(T_i) = \text{WriteSet}(T_i) \cup [w_1, \cdots, w_n]$$
$$\text{BeginTime}'(T_i) = \min(\text{BeginTime}(T_i), \text{Clock})$$
$$\text{ReadVersion}'(T_i, r_j) = \begin{cases} \text{ReadVersion}(T_i, r_j) & \text{if } r_j \in \text{ReadSet}(T_i) \\ \text{WriteVersion}(T_i, r_j) & \text{if } r_j \in \text{WriteSet}(T_i) \\ \text{Version}(r_j) & \text{otherwise} \end{cases}$$
$$\text{WriteVersion}'(T_i, w_j) = \begin{cases} \text{WriteVersion}(T_i, w_j) & \text{if } w_j \in \text{WriteSet}(T_i) \\ \text{Version}(w_j) & \text{otherwise} \end{cases}$$

where "$\uplus$" denotes executing an operation according to Definition 3.

The DLV automaton does not undergo any transition during the validation phase. The result of logical validation is reported to the transaction manager, which returns a decision *commit* or *abort* for the transaction. Committed transactions will subsequently enter the write phase. Logical validation at an object is governed by a conflict relation $<_c$ defined at the object.

Definition 6 A transaction T_i is logically valid for relation $<_c$ on an object, if the following three conditions hold:

- For each transaction T_j such that TimeStamp(T_j) < TimeStamp(T_i) and BeginTime(T_i) < CommitTime(T_j), there is no e in Intentions(T_i) and no e' in Intentions(T_j) such that $e' <_c e$.
- For each transaction T_k such that TimeStamp(T_i) < TimeStamp(T_k) and T_k is in its pending phase, there is no e in Intentions(T_i) and no e' in Intentions(T_k) such that $e <_c e'$.
- TimeStamp(T_i) > LastCommitTime.

When a committed transaction proceeds to the write phase we perform physical validation to check whether we can avoid re-executing the operations. If so, then the transaction is committed by using its view for the object to replace the permanent state of the object; otherwise we must apply Intentions(T_i) to the permanent state of the object.

Physical validation is done by checking whether the *version number* of each physical object in the *read set* of a transaction is still current. More precisely,

Definition 7 A transaction T_i is physically valid at an object if there is no physical object d in ReadSet(T_i) such that Version(d) > ReadVersion(T_i, d). That is, the value of d read by T_i is still current.

At the write phase:

Depending on the result of physical validation transaction commitment is defined by the following transition of the DLV automaton.

If physical validation succeeds:

Pre:

$T_i \notin$ Committed $\cup$ Aborted.

T_i is logically valid.

T_i is physically valid.

TimeStamp(T_i) > LastCommitTime.

Post:

$$\text{Perm}' = \text{View}(T_i)$$
$$\text{Clock}' > \text{Clock}$$
$$\text{Version}'(d_j) = \begin{cases} \text{Clock} & \text{if } d_j \in \text{WriteSet}(T_i) \\ \text{Version}(d_j) & \text{otherwisw} \end{cases}$$
$$\text{LastCommitTime}' = \text{TimeStamp}(T_i)$$

If physical validation fails:

Pre:

$T_i \notin$ Committed $\cup$ Aborted.

T_i is logically valid.

T_i is not physically valid.

TimeStamp(T_i) > LastCommitTime.

Post:

$$\text{Perm}' = \text{Perm} \bullet \text{Intentions}(T_i)$$
$$\text{Clock}' > \text{Clock}$$
$$\text{Version}'(d_j) = \begin{cases} \text{Clock} & \text{if } d_j \in \text{WriteSet}(T_i) \\ \text{Version}(d_j) & \text{otherwisw} \end{cases}$$
$$\text{LastCommitTime}' = \text{TimeStamp}(T_i)$$

4.5 Atomicity of the DLV Method

To verify the dual-level validation method, a new concept needs to be introduced. We want to define equivalence so that two histories of an object are equivalent if they have the same effects on the object. The effects of a history on an object are the values produced by *write* operations in the history.

Definition 8 Two histories of an object are *view equivalent* if they produce the same object value, denoted by " $\equiv$".

Recall that in Section 4.3.2, we model an (abstract) object as a set of primitive physical objects. As a primitive object, its value cannot be further divided, *i.e.*, an operation can read or write either the whole value or none of it. Therefore, two histories of a primitive physical object are equivalent if the *last* write to the object in two histories are of the same value.

Recall that in Section 4.3.1, we formalise an operation as a function that reads some values from and writes some values produced by it to an object. We do not know anything about the computation that each operation performs, we do not know much about the value written by an operation. All we know is that if an operation reads the same values in two histories, then the values it writes will be the same in both histories.

Lemma 5 *If View(T_i) is the view of T_i for an object when it entered the write phase, Perm is the permanent state of the object, then View(T_i) $\equiv$ Perm $\bullet$ Intentions(T_i) for any physically valid transaction T_i.*

Proof: Suppose the view of T_i for the object when it entered the write phase is

$$\text{View}(T_i) = \begin{pmatrix} h^1 \\ \vdots \\ h^m \end{pmatrix} = \begin{pmatrix} h_0^1 \bullet \cdots \bullet h_{k_1-1}^1 \bullet e_{k_1}^1 \bullet \cdots \bullet e_n^1 \\ \vdots \\ h_0^m \bullet \cdots \bullet h_{k_m-1}^m \bullet e_{k_m}^m \bullet \cdots \bullet e_n^m \end{pmatrix}.$$

Since T_i is physically valid, for any d_j either it does not belong to the read set of T_i or it has not been changed after T_i read it. That is, there is no e^j in $h_{k_j}^j \bullet \cdots \bullet h_n^j$ and $e^{j'}$ in $e_1^j \bullet \cdots \bullet e_n^j$ such that e^j writes d_j and $e^{j'}$ reads d_j. Moreover, by Definition 3, $e_1^j, \cdots, e_{k_j-1}^j$ are each either a read only event or an empty event, otherwise a shadow copy would have been created. Thus, $h^j \equiv h_0^j \bullet \cdots \bullet h_{k_j-1}^j \bullet h_{k_j}^j \bullet \cdots \bullet h_n^j \bullet e_1^j \bullet \cdots \bullet e_{k_j-1}^j \bullet e_{k_j}^j \bullet \cdots \bullet e_n^j$, because $e_{k_j}^j \bullet \cdots \bullet e_n^j$ reads the same values, thus writes the same values in both histories. Therefore, View(T_i) $\equiv h_0 \bullet \cdots \bullet h_n \bullet e_1 \bullet \cdots \bullet e_n =$ Perm $\bullet$ Intentions(T_i). □

Notice that in the DLV method, at any time there is at most one transaction in the write phase at a particular object. Therefore, the view of a transaction for an object will be the same throughout the write phase.

Lemma 6 *For any dual-level validation automaton whose logical validation relation $<_c$ is a serial dependency relation, Perm $\bullet$ Intentions(T_i) is legal for any logically valid T_i.*

Proof: Suppose $C = o_1 \bullet \cdots \bullet o_n$ is the component of T_i at the object. We may now write Perm $= h_0 \bullet$ Intentions(T_1) $\bullet \cdots \bullet$ Intentions(T_{i-1}) $= h_0 \bullet \cdots \bullet h_n$, and Intentions($T_i$) $= e_1 \bullet \cdots \bullet e_n$, where $T_1, \cdots, T_{i-1}$ have been committed with timestamp earlier than that of T_i. The proof is by induction on the number of transactions that have entered the write phase before T_i.

When $i = 1$, Perm $\bullet$ Intentions(T_1) $= h_0 \bullet$ Intentions(T_1): it is legal by Lemma 4 because $h_1 \bullet \cdots \bullet h_n$ is empty.

When $i > 1$, by the inductive hypothesis, $h_0 \bullet$ Intentions(T_1) $\bullet \cdots$ Intentions(T_{i-1}) is legal. Since T_i is logically valid by assumption, there is no e in Intentions(T_1)$\bullet \cdots$ Intentions(T_{i-1}) and e' in $e_1 \bullet \cdots \bullet e_n$ such that $e <_c e'$. Moreover, $<_c$ is a serial dependency relation. Therefore, $h_0 \bullet$ Intentions(T_1) $\bullet \cdots$ Intentions(T_{i-1})$\bullet$ Intentions(T_i) is legal by Lemma 4. □

The following lemma is a direct consequence of Lemma 5 and Lemma 6:

Lemma 7 *The dual-level validation method is atomic, if the conflict relation used by the logical validator is a serial dependency relation.*

Proof: From the *dual-level validation automaton*, we know that the permanent state of an object is the serialisation in timestamp order of the schedule accepted by the automaton. Moreover, Lemma 5 and Lemma 6 imply that each commit carries the permanent state from one legal history to another. Therefore, the schedule is atomic. Since its automaton only accepts atomic schedules, the DLV method is atomic.□

Since the DLV method is atomic, an object which uses the DLV method to control concurrent accesses to it will be atomic. Atomicity for an object is called *local atomicity* because it deals only with the events involving the particular object. In a system composing many objects, the fact that all component objects are locally atomic cannot guarantee that the system is atomic — in addition there must be at least one serialisation order for committed transactions on which all objects can agree. Therefore, a property should be found such that if each object in the system satisfies it then the system is atomic. This kind of property is called a local atomicity property [Wei89].

Weihl [Wei89] has identified three local atomicity properties: dynamic atomicity, static atomicity and hybrid atomicity. The dynamic atomicity characterises the behaviour of a class of types that ensures serialisability dynamically based on the order in which transactions execute operations. Static atomicity characterises the behaviour of a class of types that ensures serialisability statically based on some predetermined order of transactions. Hybrid atomicity characterises the behaviour of a class of types that ensures serialisability based on the order in which transactions commit.

Lemma 8 *The dual-level validation method is hybrid atomic, if the conflict relation used by the logical validator is a serial dependency relation.*

Proof: From Lemma 7 we know that the DLV method is atomic. Furthermore, from the *dual-level validation automaton* we know that any schedule accepted by the DLV automaton is serialisable in the order in which transactions commit. Therefore, the DLV method is hybrid atomic according to Weihl's definition.□

Since the DLV method is hybrid atomic and hybrid atomicity is a local atomicity property, a system will be atomic if all objects in the system use the DLV method to provide local atomicity.

4.6 Summary

In this chapter, we have proved the DLV method is hybrid atomic, hence can be used to provide local atomicity for atomic data types. The proof is conducted in three steps. We first prove the correctness of an important lemma, the Lemma 4, then verify the DLV method by using the lemma, and finally certify that the DLV method is hybrid atomic.

To prove Lemma 4, we first decompose an abstract object into a set of primitive physical objects, and formalise an operation on an abstract object as a function that reads some values from and write some values produced by it to some primitive physical objects. Then we apply the formal method developed by Weihl [Wei89] and the *serial dependency relation* introduced by Herlihy [Her90] to reason the local legal property of each individual primitive physical object. Finally, we reason the relationship between the legal property of an abstract object and the local legal property of its primitive physical objects.

Chapter 5

Global Atomicity and Resilience

In the last two chapters we focussed on local atomicity of atomic data types. However, local atomicity does not automatically guarantee atomicity of a system. In this chapter, we discuss how to ensure the global atomicity of a system when objects use the DLV method for providing local atomicity, in both the absence and the presence of failures. First a model in a centralised system is established, then it is extended to be usable in distributed systems.

5.1 Atomic Commitment

In a system which takes the atomic object approach for providing data consistency, commitment of a transaction must be atomic over all objects involved in the transaction to ensure global atomicity. First the objects involved agree on whether to commit or abort a transaction, then all the objects must implement the agreement whatever happened.

To ensure that all the objects involved in a transaction make a common decision on a commit request, a transaction manager is introduced into the system to be responsible for making the final commit or abort decision. The transaction manager and objects cooperate according to a protocol, called the *local 2-phase-commitment* (L2PC) protocol, for its similarity to the 2-phase-commitment (2PC) protocol developed for distributed transactions[Gra79]. Transactions are controlled by and interact with objects through the transaction manager. The transaction manager may simultaneously control multiple independent transactions.

To process transactions, the transaction manager forwards object operations enclosed in transactions to corresponding objects. Objects are responsible for executing object operations requested by the transaction manager. Operations on an object from the same transaction form a *transaction component.* When a commit request comes from a transaction, the transaction manager asks objects involved in the transaction, called *participant objects*, to vote whether the transaction should be committed. Then, it makes the final decision based on the replies: *commit*, if all the replies are "commit"; or *abort*, otherwise. Finally, it propagates the final decision to all participant objects. A participant object implements the decision accordingly after receiving it.

Like 2PC, the basic idea of L2PC is to determine a unique decision for all participant objects with respect to committing or aborting a transaction. If an object is unable to commit the transaction, then all participant objects must abort the transaction. The L2PC protocol consists of two phases. The goal of the first phase is to reach a common decision; the goal of the second phase is to implement this decision.

The L2PC protocol on the transaction manager side can be described as follows:

1. *During the read phase*: when receiving an object operation from a transaction, the transaction manager forwards it to the corresponding object.

2. *Asking for votes*: when receiving a commit request from a transaction, the transaction manager generates a timestamp for the transaction, sends an *asking-for-vote* command with the timestamp to each participant object, then waits for replies.

3. *Making decision*: the transaction manager decides to commit the transaction if all replies are "commit", to abort the transaction otherwise.

4. *Propagating decision*: the transaction manager propagates the final decision to participant objects.

5. *Waiting for acknowledgement*: the transaction manager waits for ACK messages from participant objects.

The protocol on the object side has been given in Section 3.3. Briefly speaking, an object performs any operations forwarded to it, decides its vote based on the result of logical validation, and commits transactions through the Write Phase Manager.

5.2 Recovery

In the last section, we described the protocol for atomic commitment in the absence of failures. However, the possibility of failure always exists; recovery methods therefore must be provided so that the system can be restored to a consistent point when a failure occurs.

5.2.1 Three Kinds of Recovery

To ensure data consistency a system needs to provide three kinds of recovery [CP84]. The activity of ensuring a transaction's atomicity in the presence of *transaction aborts* is called **transaction recovery**. The activity of ensuring a transaction's atomicity in the presence of *system crashes*, in which only *volatile storage* is lost, is called **crash recovery**. The activity of providing a transaction's durability in the presence of *media failures*, in which *nonvolatile storage* is lost, is called **database recovery**.

In Chapter 2 we pointed out that a transaction's *isolation* property may be violated in an implementation of transactions based on atomic data types. This results in the failure of traditional *state-based* recovery. The DLV method uses optimistic concurrency control in which a transaction performs update operations on local copies of objects during its read phase. Transaction abort is therefore achieved by discarding these shadow copies. Persistent object values are not affected until the write phase of a transaction.

Database recovery is independent of the concurrency control method used by a transaction system. Various methods such as *stable storage* [Lam81] can be used for providing database recovery. The method is typically implemented by replicating the same information on several disks that have independent failure modes and using a so-called *careful replacement strategy*[Koh81]. At every update operation, first one copy of the information is updated, then the correctness of the update is verified, and finally the second copy is updated.

5.2.2 Crash Recovery

Our crash recovery method is log-based [Gra79]. A *log* usually contains information for undoing or redoing all actions which are performed by transactions; whenever a transaction performs an action on an object, a log record is written in the log file in the form:

(*Transaction id, object id, old value, new value*)

Moreover, when a transaction is started, committed, or aborted, a *begin_transaction, commit*, or *abort* record is written in the log.

In a system using the DLV method for providing local atomicity, however, there is no need to write a log record for every action. This is because, in this method, updates to an object can only be made on its shadow copies, before a transaction commits.

A log record is recorded on stable storage when each phase of a transaction starts, and when a transaction completes (*aborts* or *commits*). Hence, during a recovery procedure after a system crash, the status of a transaction can be determined.

If a transaction was in its read phase when the system crashed, it will be aborted when the system restarts. No special recovery operation needs to be done, since the transaction neither made any change to a persistent object, nor made any promise. Any local copy of objects it has created will be collected by the garbage collector.

Before entering the validation phase, the *performed-operations table* (POT) and the *accessed-objects table* (AOT) of the transaction must be recorded on stable storage. If a transaction was in its validation phase when the system crashed, the object will do the validation again at restart. The information necessary for the validation, i.e. the POT, has been recorded on stable storage.

If a transaction was in its pending phase when the system crashed, it will remain in this phase at restart. No special action needs to be taken. However the *pending queue* of an object which records all the transactions in their pending phase needs to be refreshed to stable storage whenever a change is made to it.

The write phase of a transaction is separated into two or three steps: a physical validation step, possibly a re-execution step, and a merging step. A log record is necessary to indicate the end of a step. Moreover, if a re-execution step is required, the new POT and AOT produced by the re-execution need to be recorded on stable storage before the merging step.

If a transaction was in its physical validation step when the system crashed, at restart the object will perform physical validation again for that transaction. The validation can be performed because the AOT with the required information has been written to stable storage. If a transaction was in the merging step, at restart the object will redo the merging operation. This can be done because all the shadow copies as well as the AOT have been written to stable storage. Notice that a merging operation is *idempotent*. If a transaction was in the re-execution step, at restart the object will re-execute the operations on the object of the transaction. The re-execution can be done since the POT which recorded the operations of the transaction has been written to stable storage.

5.2.3 Log Records

We describe in detail the log protocol used by the DLV method in the following two subsections. We discuss when and in what form a log record is needed in this subsection, and explain how to restore a system to a consistent state in the next subsection.

Every object and the transaction manager have their own log file. The log protocol can be described as follows:

• Before asking for votes from participant objects, the transaction manager writes an "*asking_for_votes*" record in its log.

• After finishing a logical validation but before casting its vote to a transaction, an object records the related information in its log.

1. If voting for "commit", it should ensure that it is able to commit the transaction even if failure occurs afterwards. In practice, this requires two things to be recorded on the log of the object:
   (a) All the information which is required for locally committing the transaction. This means the *performed-operation table* (POT) and the *accessed-objects table* (AOT) of the transaction have to be recorded on the log of the object.
   (b) The fact that this object has voted for "commit" for the transaction. This means that a log record of a type, called a "*voted_for_commit*" log record, must be recorded on the log in the form:
   (*object id, transaction id, timestamp, "voted_for_commit"*).
2. If voting for abort, it only needs to write a "*voted_for_abort*" record on its log in the form: (*object id, transaction id, timestamp, "voted_for_abort"*)

• After deciding whether to commit or abort a transaction, the transaction manager must record its decision on its log. This corresponds to writing a "*transaction_commit*" or "*transaction_abort*" record in its log. The fact that the transaction manager records its decision on its log means that the transaction will eventually be committed or aborted, in spite of failures.

• After receiving a decision about a transaction's outcome from the transaction manager, an object writes an "*object_commit*" or "*object_abort*" record in its log accordingly.

• When the transaction manager has received an ACK message from every participant object, it writes an "*end_transaction*" record in its log. From this moment, the transaction can forget the outcome of the transaction.

• When an object commits or aborts a transaction locally, recovery measures are still necessary.

* It writes an "*object_abort*" record in its log before performing an abort, and writes an "*end_object_abort*" record afterwards.
* Performing commit is more complex.
  - It writes an "*object_commit*" record in its log before starting the physical validation.
  - If the validation result is "accept", it writes a "*to_write*" record in its log before merging shadow copies into the object permanent state, and writes an "*end_write*" record afterwards.
  - If the validation result is "reject", then it writes a "*to_redo*" record in its log before performing the re-execution, and writes the new AOT and a "*redo_write*" record afterwards. Then it performs the merging action, following by writing an "*end_redo_write*" and an "*end_redo*" record in its log.
  - An "*end_commit*" and an "*end_subtransaction*" record are written in the log.

In summary, for a completed transaction T_i, the transaction manager should have written the following log records in its log:

begin_transaction T_i
  asking_for_votes T_i
  transaction_commit or *transaction_abort* T_i
end_transaction T_i

The log records written in the log of an object for a committed transaction T_i are as follows:

begin_subtransaction T_i
  AOT and *POT* T_i
  voted_for_commit T_i
  to_commit T_i
  (physical validation)

(succeed)	**(failed)**
to_write T_i	*to_redo* T_i
	AOT T_i
	redo_write T_i
	end_redo_write T_i
end_write T_i	*end_redo* T_i

```
        end_commit $T_i$
    end_subtransaction $T_i$
```

and for an aborted transaction T_i:

```
begin_subtransaction $T_i$
    voted_for_commit or voted_for_abort $T_i$
    object_abort $T_i$
    end_object_abort $T_i$
end_subtransaction $T_i$
```

5.2.4 Recovery Procedure

We have described when and what kinds of log records should be written in logs in the last subsection. In this subsection, we discuss how these logs are used to restore a system to a consistent state when system crashes occur, *i.e.*, we analyse the behaviour of the protocol when failures occur at different times.

We suppose that the transaction manager and all atomic objects run in the system as threads in a single memory address space. Therefore, if a failure with loss of volatile storage occurs, then the whole system stops running, *i.e.,* a fail-stop model is assumed.

- The transaction manager:

  1. *A failure occurs before writing the* transaction_commit *or* transaction_abort *record.* In this case, the transaction manager simply sends an "abort_transaction" command to every participant object to ask it to abort the transaction locally. Since the transaction manager has not already propagated its decision, no object can have updated the object permanent state. Therefore, it is safe to abort the transaction.

  2. *A failure occurs after writing the* transaction_commit *record, but before writing the* end_transaction *record.* In this case, the transaction must be committed in all participant objects, for some objects may have already made changes to object permanent states. The transaction manager must send the final decision to all participant objects again, since some or all of them may not have received the decision. An object that has received the decision must recognise that the new decision message is a repetition of a previous one.

  3. *A failure occurs after writing the* transaction_abort *record, but before writing the* end_transaction *record.* In this case, the transaction manager must send the "abort"

command to all participant objects again, for some or all of them may not have received it. As before, participant objects must not be affected by receiving the command twice.

- At an object:

1. *A failure occurs before writing the* voted_for_commit *or* voted_for_abort *record.* In this case, at restart the object can abort the transaction locally without delay, for the transaction manager must be in case 1, thus "abort" must be the final decision. When it receives the "abort" command from the transaction manager later, the object simply sends an ACK message to the transaction manager.

2. *A failure occurs after writing the* object_abort *record, but before writing an* end_object_abort *record.* In this case, the object simply performs the aborting operation again. Notice that the aborting operation should be *idempotent*, *i.e.*, performing it several times is equivalent to performing it once.

3. *A failure occurs after writing the* voted_for_commit *record, but before writing a* to_commit *or* to_abort *record.* In this case, the object must wait for further messages from the transaction manager.

4. *A failure occurs after writing the* to_commit *record, but before writing any other record.* In this case, the object at restart resumes by doing physical validation.

5. *A failure occurs after writing the* to_write *record, but before writing an* end_write *record.* In this case, the object at restart resumes by doing merging operation; the merging operation can be performed because all the shadow copies as well as the AOT have been written on stable storage. Notice that the merging operation is *idempotent.*

6. *A failure occurs after writing the* to_redo *record, but before writing a* redo_write *record.* In this case, the object simply deletes the AOT and then performs the re-execution action. Operations of the transaction component can be found from the POT that has been written in the log.

7. *A failure occurs after writing a* redo_write *record, but before writing an* end_redo_write *record.* In this case, the object at restart continues working from doing the merging operation. All the shadow copies as well as the AOT can be found from the log.

It can be seen from the above description that a log must be very large. Processing a large log might be tolerable on media failure but transaction abort must be efficient and recovery after a crash should be reasonably fast [Bac93]. A *checkpoint* technique [Koh81] therefore is introduced to reduce the log by deleting the records related to completed transactions. We are not going to discuss this technique in this thesis, for it has been well developed already.

5.3 Distributed Transactions

In this section, we discuss global atomicity in a distributed environment. First we establish a distributed transaction system model, and then describe an atomic commitment protocol which ensures the commitment of a transaction is atomic over all participants.

5.3.1 The Computation Model

We consider an environment consisting of one or more *virtual sites*, and connected by a *virtual communication network*. Every site can communicate with any other. Messages may be delayed, or lost, but not duplicated, nor created by the communication network, and they are always delivered in order.

The architecture of the distributed transaction system can be described by Figure 5.1. The four basic components are Transactions, Distributed Transaction Managers (DTMs), (Local) Transaction Managers (LTMs), and Atomic Objects. Each transaction is controlled by and interacts with atomic objects through a single DTM. The DTMs may simultaneously control multiple independent transactions. The DTM in charge of a transaction forwards object operations to the LTM local to the object. The LTMs are responsible for managing their own objects. Objects are responsible for completing object operations on behalf of transactions.

In the distributed transaction system model, atomic objects work exactly the same as in the centralised model, and the communication protocol between the LTM and atomic objects remains the same. However, the LTM accepts operations from DTMs, instead of from transactions.

A distributed transaction T usually accesses several atomic objects resident at different sites, but it has a "home site" — the site where it originated. T submits its operations to the DTM at its home site, which is then known as the *coordinator* for that transaction. This DTM subsequently forwards the operations to the LTMs in the appropriate sites, which are then known as *participants* in the transaction. After an LTM becomes a participant in the transaction, the coordinator establishes a connection with it. Thereafter, whenever

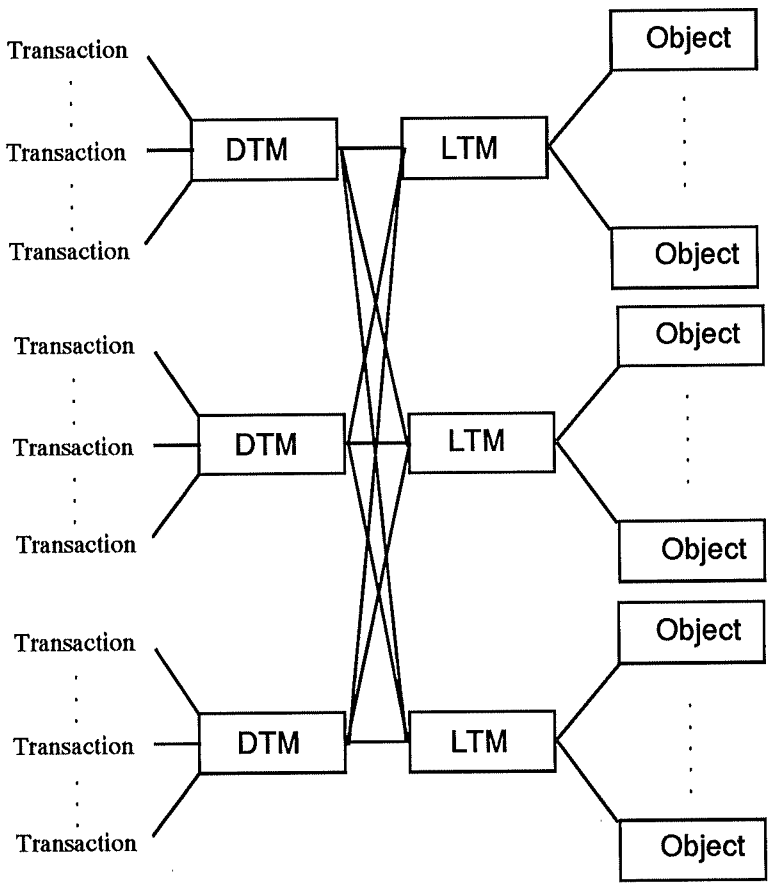


Figure 5.1: The Architecture of the Distributed Transaction System

there is access to an object at that site, the coordinator forwards the operation to the participant LTM; the participant then forwards it to the appropriate object where the operation will be performed. However, when an *end_transaction* request comes from a transaction, a protocol is needed between the coordinator and participants to ensure that the commit of a transaction is atomic over all participants. The general idea of *2-phase commitment* (2PC) has been developed to meet this requirement. Other such protocols have been defined which vary with respect to the failures they can tolerate, the number of communications that are needed, etc.[DS83]. In the rest of this section, we describe how to make the 2PC protocol work together with the L2PC protocol to ensure atomic commitment of distributed transactions.

5.3.2 Atomic Commitment Protocol

In the previous sections, we presented an L2PC protocol to ensure atomic commitment of a transaction over all participant objects. In this subsection, we describe a protocol that ensures atomic commitment of transactions over all participant LTMs in the distributed transaction model. The protocol, referred as the *distributed 2-phase-commitment* (D2PC) protocol, is described as a modification to the basic 2PC protocol.

To commit a transaction T:

1. The coordinator generates a timestamp ts(T) from its logical clock.
2. The coordinator sends a *prepare-to-commit* command with the timestamp and the transaction identifier to each participant LTM.
3. After receiving the *prepare-to-commit* command, each participant LTM sends an *asking-for-vote* command with the timestamp ts(T) to each participant object at its site.
4. If all the votes received from participant objects are "commit", the participant LTM returns "success" to the coordinator; otherwise, it returns "failure".
5. The coordinator receives responses and determines what action to take. If all replies are "success", it decides to commit the transaction; otherwise, it decides to abort the transaction.
6. The coordinator propagates the decision to all participant LTMs.
7. Each participant LTM, after receiving the decision, sends an ACK to the coordinator and forwards the decision to every participant object at its site.
8. The coordinator waits for ACK messages from participant LTMs.

Notice that the actions taken by a participant LTM are almost the same as the transaction manager in the L2PC protocol, except:

1. The timestamp is forwarded from the coordinator, not generated by the LTM itself.
2. The final decision to commit or abort a transaction is made by the coordinator. After making its local decision based on the replies from participant objects, instead of propagating that decision to objects, an LTM reports its local decision to the coordinator, and waits for the final decision from the coordinator. When it gets that decision, it forwards it to every participant object at its site.

The actions taken by an object are not affected by this extension.

It is worth pointing out that the validation algorithm applied by the DLV method does not require that transactions do their logical validation in timestamp order. This feature is important for the above protocol. Since in a distributed system, it is very likely that a transaction with a smaller timestamp requests to validate later than a transaction with a larger timestamp. Without this feature, such kind of requests would be rejected automatically.

It is also worth pointing out that performing logical validation, waiting for the final decision, and applying updates are three independent actions in the D2PC protocol, they need not to be a single atomic one. An object can begin validating a new transaction as soon as it has finished an earlier one. This is an important feature of the D2PC protocol. If an object cannot begin validating a new transaction until the earlier one finishes its write phase or gets the final decision, object concurrency and availability will be reduced greatly. This is because getting the final decision involves communication between different sites.

5.3.3 Other Commitment Protocols

Herlihy [Her90] proposed two commitment protocols that are also based on the 2PC protocol. The first one, like the D2PC protocol, assigns timestamps to transactions at the start of the first phase. However, unlike the D2PC protocol, its validation method requires transactions to be validated in timestamp order; it therefore aborts transactions that try to validate out of order automatically, thus causing unnecessary transaction aborts. The second protocol has the advantage that it does not abort transactions with out-of-order validation requests, because it chooses the timestamp for a transaction in the second phase, *i.e.*, after the coordinator has decided to commit the transaction. However, it can cause incipient deadlocks, which are broken by timeout.

The major drawback of these two protocols is that they permit only one transaction at a time to validate at any particular object, and an object is held by a transaction during the whole process of its commitment. This reduces both object concurrency and object availability greatly. The problem is caused by the concurrency control methods on which the protocols are based. First, they require that transactions are validated in timestamp order. Second, they require that the *validation phase* and *write phase* of a transaction are atomic. The D2PC protocol does not have the problem since the DLV method does not have such requirements.

Bacon [Bac93] proposed a two-phase-validation method that aims to provide greater object availability by separating out the *write phase* of a transaction from the two phase commitment protocol. An *update manager* is introduced to be responsible for the queue of transactions that have been validated for update. If a transaction is validated successfully in the first phase, the participants do not need to apply the intention updates during the second phase. Instead the validation manager applies to the *update manager* for a timestamp for the transaction, and then informs each participant object of the decision. Once a transaction has its updates guaranteed, these can be applied asynchronously at participant objects. Objects are therefore held only during the validation, unlike the case of Herlihy's protocols that need to hold objects during the whole commitment process. However, it still requires performing local validation and waiting for the final decision to be an atomic action, thus reducing object concurrency in the validation phase.

5.3.4 Timestamp Generation

In the D2PC protocol, DTMs need to generate timestamps for transactions. Two things need to be taken into account when designing a timestamp generation algorithm. First, timestamps must be system-wide unique. Second, timestamps generated by different DTMs at a time should not vary widely. Otherwise, transactions submitted to some DTMs will have little chance to pass the logical validation because their timestamps are too small.

Lamport's method [Lam78] can be used to generate timestamps for transactions. Each DTM is given a unique site number which is always used as the second part of timestamps generated by the DTM. Each DTM also maintains a number counter, called *timestamp counter*. To generate a timestamp, a DTM simply increases the timestamp counter, and takes the resulting value as the first part of the timestamp. To compare two timestamps, the first parts are compared; only if the first parts are equal, the second parts are compared. In such a way, timestamps are ensured to be system-wide unique.

5.3.5 Recovery of Distributed Transactions

Recovery mechanisms for distributed transactions require understanding not only the failures which can occur at each site but also the failures which may occur in the communications between sites. There is a great variety of possible failures with communications. However, modern communication networks are capable of eliminating most of them. In a distributed system, it is usually assumed that there are two basic types of possible communication error: lost messages and network partitions.

Dealing with network partitions is a harder problem than dealing with site crashes or lost messages. Fortunately, in many computer networks partitions are much less frequent than site crashes. Therefore, algorithms of several levels of reliability can be designed, which are capable of dealing with the following failures, in order of increasing difficulty:

- class 1: site failures only.
- class 2: site failures and lost messages.
- class 3: site failures, lost messages, and network partitions.

The D2PC protocol is a modification of the basic 2PC protocol, hence they have the same capability for dealing with failures. They can deal with failures in class 3, but may cause objects to be *blocked* when the coordinator site crashes or communication failure occurs. We are not going to discuss this issue further, for it is beyond the research of this thesis. Ceri [CP84] gives a good description about how the 2PC protocol deals with failures.

5.4 Summary

In this chapter, the issue of global atomicity has been addressed, *i.e.*, how to ensure that transactions commit atomically over all participant objects. In the first section, an L2PC protocol is described which ensures a transaction's global atomicity in a centralised system. In the second section it is extended to provide recovery after failure with loss of volatile storage.

In the third section, a distributed transaction system model is established, and an atomic commitment protocol for distributed transactions, the D2PC, is described and compared with some other commitment protocols.

Both the L2PC and the D2PC protocols are designed based on the assumption that objects use the DLV method for providing local atomicity.

Chapter 6

Constructing Atomic Objects and Transactions

In the previous chapters, a concurrency control protocol, the DLV method, was presented and verified, and a distributed transaction model was described. In this chapter, the approach taken in this thesis for implementing the user-defined atomic data types and the method for constructing transactions are described.

6.1 Atomic Data Types

An atomic data type, like an ordinary abstract data type, provides a set of objects and a set of operations. As with ordinary abstract data types, the operations provided by an atomic data type are the only means for users to access or manipulate objects of the type. Unlike regular types, however, an atomic data type provides serialisability and recoverability for transactions that use objects of the type. That is, a regular type defines the behaviour of objects in a sequential and reliable environment, while an atomic data type defines the behaviour of objects in a concurrent and unreliable environment. Therefore, more information needs to be maintained and more operations need to be defined in atomic data types.

Definition of an atomic data type needs to represent application information, synchronisation information and recovery information; to implement synchronisation operation, recovery operation and object operations in terms of their representations; and to specify the semantics of object operations. Application information is the data an application intends to maintain; other kinds of information are used for ensuring the consistency of application information in a concurrent and unreliable environment. For example, for a bank account object, a variable *balance* is used to represent the current balance of the account. If a two-phase locking protocol is used for providing concurrency control, then data needs to be maintained to indicate the current locks set on the object as well as their modes and their holders. Data also needs to be maintained for transaction recovery, for example, through recording the object state before a transaction starts so that its effect, if required, can be removed by restoring this state.

Obviously, the concurrency control method used by an object affects what kind of synchronisation and recovery information needs to be kept. Using a two-phase locking protocol, an object needs to keep locks set on it; using an optimistic method, an object needs to keep shadow copies and accessing sets for every transaction. The concurrency control method can also affect the representation of application information, consequently object operations. For example, in order that results of operations can be used for scheduling transactions, an object in Argus is implemented as a history (log). The *balance* of a bank account is represented by a number and a history (log), with the number representing the balance at some previous stage and the history representing the operations executed since then.

6.2 Approaches

Approaches taken for defining atomic data types can be divided into three classes: implicit approach, explicit approach and hybrid approach, according to whether the system or programmers are responsible for implementing the synchronisation and recovery operations. In an implicit approach, programmers need only describe the semantics of object operations and implement object operations based on the assumption that there is no concurrency and no failure. It is the system which is responsible for implementing the synchronisation and recovery operations by using the semantics of objects.

In an explicit approach, as well as object operations, programmers need also to implement the synchronisation and recovery operations themselves. The system only provides some support to make the implementation easier.

In a hybrid approach, the work of implementing the synchronisation and recovery operations is shared by the system and programmers. For example, transaction synchronisation in Argus is done by programmers, but committing or aborting a transaction is done by the system. That is, programmers are responsible for deciding whether an operation can be executed immediately or has to be retried, while the system is responsible for making the outcome of a committed transaction become permanent and removing the effect of an aborted transaction. A special kind of variable is provided by which the system and programmers communicate with each other according to a predefined protocol.

Each approach has its advantages and disadvantages. The implicit approach makes the definition and implementation of user-defined atomic data types simple, but it is not flexible. The strategy for synchronisation and recovery is decided by the system, which may not be suitable for all applications. The explicit approach is more flexible but requires more work on the part of programmers. Using an explicit approach programmers have control over what level of concurrency a type supports, and can choose the synchronisation and recovery method within a certain range. However, this flexibility has its potential penalties, since careless programming could lead to chaos as objects are manipulated without being supervised by a concurrency controller. How much flexibility a hybrid approach can provide and how much work on the part of programmers it requires depend on how the work of implementing synchronisation and recovery is shared by the system and programmers.

6.3 An Implicit Approach

The aim of this research is to lessen the programmer's burden in implementing atomic data types, thus an implicit approach is taken for their definitions.

To support an implicit approach, the system must provide the synchronisation and recovery operations for atomic objects. This requires that it should be possible to implement the concurrency control method used by the system independently from atomic objects. Furthermore, the concurrency control method should be able to use the semantics of object operations to achieve greater concurrency. The DLV method meets these requirements. It can be implemented without knowledge of the individual objects which will make use of it, and can synchronise transactions according to the semantics of object operations.

Supporting type-specific concurrency in an implicit approach requires that a mechanism should be provided for programmers to specify the semantics of object operations.

The system designed in this thesis provides four kinds of mechanisms to support the implicit approach. First, a special type, called *Scheduler*, is presented for providing the synchronisation and recovery operations. It is an implementation of the DLV method. User-defined atomic data types can be derived from *Scheduler* by using the type-inheritance facility available in object-oriented languages. In such a way, programmers can define atomic data types without implementing the synchronisation and recovery operations themselves. Second, a small language is provided for programmers to specify the semantics of object operations in a declarative way. Third, a preprocessor is implemented. A difficulty particular to an implicit approach is how the system can get the information necessary for synchronisation and recovery, because programmers should not be asked to provide it. By preprocessing atomic type definitions, the preprocessor adds some code in object operations so that the information necessary for synchronisation and recovery can be collected automatically when object operations are invoked by transactions. Fourth, a transaction mechanism is provided by which user transactions can be constructed by programmers and controlled by the system. The transaction mechanism starts, commits or aborts transactions according to the D2PC protocol.

In the rest of this chapter, we first describe the language developed for specifying the semantics of object operations. Then following an introduction to the concepts of object orientation, we discuss how type inheritance can be used to construct user-defined atomic data types. Finally we discuss the design of the transaction mechanism and the method for constructing transactions through that mechanism.

6.4 Representing the Semantics of Operations

A major aim of atomic data types is to provide great concurrency by making use of the semantics of object operations. Unfortunately, introducing application semantics may make the implementation of atomic data types very complicated if an unsuitable method is used to represent the semantics. This section discusses this issue and introduces the method used in this thesis for representing the semantics of object operations.

6.4.1 Levels of Semantics

When specifying how object operations may interact, the amount of concurrency that can be permitted depends in part on how much detailed knowledge is available concerning the semantics of operations. One can take advantage of increased knowledge about operations being performed in order to achieve greater concurrency while still providing serialisability. The semantic knowledge used by concurrency control methods can be classified into five levels; each level may include more semantics, and hence may permit more concurrency.

1. *Read/Write:* Only the semantics of read and write is used in this level.

2. *Type-Specified:* Type-specific operation compatibilities may be specified in this level. The compatibilities specify the commutativity of object operations which must be true for all possible parameters and all possible object states.

3. *Object Item:* In many cases, the particular entities which must be controlled are discrete and can be determined by the parameters of object operations. At this level, compatibilities can be defined by taking into account the object item upon which the operation will be performed.

4. *Operation Result:* In some cases, distinguishing operations that fail from those that succeed may permit more concurrency than protocols not making this distinction. This distinction can be used for specifying compatibilities at this level.

5. *Object State:* In a state based concurrency control protocol, the object state may be used for providing greater concurrency.

6.4.2 Two Different Ways

Making use of the semantics of operations to provide great concurrency can be implemented in two ways. One is to combine the synchronisation into the implementation of object operations, so that operation semantics can be used directly to make synchronisation decisions by operations without needing to be expressed explicitly. In this way, more information can be used for synchronisation, such as the object state, local variables, operation's parameters and operation's results. Therefore, greater concurrency may be achieved. The disadvantage of this approach however is obvious. Firstly, it makes the implementation of object operations become complicated. Secondly, there exist potential dangers, since careless programming could lead to chaos. Finally, changing the level of semantics used for synchronisation, or changing the synchronisation and recovery scheme would require the re-implementation of object operations.

Another way to make use of the semantics is to let programmers express it explicitly in some convenient way, and for the concurrency controller to use it when performing synchronisation and recovery. In this way, the implementation of object operations will not be affected by changes to the synchronisation or recovery scheme, nor by changes to the level of semantics used for synchronisation. However, a suitable form of expressing semantics must be provided in this case. It should be easy to use and have strong expressibility in order to represent most of the semantics of object operations.

6.4.3 The Language

The concurrency control protocol used by objects for providing local atomicity in this thesis is the DLV method. The amount of concurrency a type permits in this case depends on the *conflict relation* provided by programmers. The smaller the conflict relation is, the more concurrency a type will permit. Therefore, the language for specifying conflict relations should let programmers represent enough semantics so that conflict relations can be specified concisely. In this thesis, a very small language is developed for the purpose. The language's syntax is given in Appendix A. The language can represent the first four levels of semantics given in the last subsection. The fifth level of semantics is not included since the DLV method is a conflict-based protocol which does not use that kind of information for synchronisation.

When specifying the conflict relation of an atomic data type, an operation is represented by its name and result. The result of an operation is simply distinguished as either *failed* or *succeed*, since usually only this distinction makes a significance different to conflict relations. The object item that an operation acts on can also be taken into account if applicable. The relationship between two object items can be classified into: $=$, $<$, $>$, $\leq$, $\geq$, or $\neq$. The default comparison is done according to lexicographical order. However, programmers can provide their own comparison operations.

For example, suppose there are two operations: *oper1* and *oper2* (they are not necessarily different). If *oper1* invalidates *oper2* in all cases, then this conflict can be represented as:

((*oper1, any*); (*oper2, any*); *any*).

If *oper1* invalidates *oper2* only when they act on the same object item, then this conflict can be represented as:

((*oper1, any*); (*oper2, any*); =).

If *oper1* invalidates *oper2* only when they act on the same object item and both of them succeed, then this conflict can be represented as:

((*oper1, succeed*); (*oper2, succeed*); =).

Sometimes, object operations can be partitioned into classes, and conflict relations can be specified in terms of these classes, hence specifications become more tidy and clear. To support this kind of specification, the language permits specifying several invalidations in one expression. For example, suppose there are two sets of operations, say *(oper11, oper12, oper13)* and *(oper21, oper22)*. If an operation in the first set invalidates all operations in the second set when they act on the same object item, then this invalidation can be represented as:

((*oper11, any*)/(*oper12, any*)/(*oper13, any*); (*oper21, any*)/(*oper22, any*); =).

6.4.4 An Example

In this subsection we show the expressive power of the language by using an example. Consider a *directory* data type that is intended to provide a mapping between text strings and capabilities for arbitrary objects. The usual operations are provided:

- Insert(*str*, *capa*) inserts *capa* into the directory with key string *str* and returns *ok* or *duplication.*
- Delete(*str*) deletes the capability stored with key string *str* from the directory and returns *ok* or *unexist.*
- LookUp(*str*) searches for a capability in the directory with key string *str* and returns the capability *capa* or *unexist.*
- Dump() returns a vector of <*str*, *capa*> pairs with the complete contents of the directory.

Suppose that one wishes to specify the *Directory* type so as to permit serialisation of transactions that perform operations on *Directories.* If the language for specifying conflict relations only permits operations to be classified as *read* and *write*, then the conflict relation of the *Directory* can be described as follows:

(Insert, Insert)	(Delete, Insert)
(Insert, Delete)	(Delete, Delete)
(Insert, LookUp)	(Delete, LookUp)
(Insert, Dump)	(Delete, Dump)

The problem with using such limited semantic information is that concurrency is restricted unnecessarily. For example, a transaction performing a *LookUp("John")* operation will be forced to abort, if another transaction performs a *Delete("Guang")* operation concurrently and is validated earlier. The outcome of *LookUp("John")* does not depend in any way on the eventual outcome of *Delete("Guang")*; this abort is unnecessary.

The unnecessary loss of concurrency in this example is caused by the lack of semantic information in the specification of the conflict relation. By using the language proposed in the last subsection, this problem can be alleviated. This is because more knowledge about operations can be taken into account when specifying the conflict relation, for example the object item an operation acts on and the result of an operation. The conflict relation of the *Directory* can be described in the language as follows:

((Insert, succeed); (Insert, succeed); =)	((Delete, succeed); (Insert, failed); =)
((Insert, succeed); (Delete, failed); =)	((Delete, succeed); (Delete, succeed); =)
((Insert, succeed); (LookUp, failed); =)	((Delete, succeed); (LookUp, succeed); =)
((Insert, succeed); (Dump, any); any)	((Delete, succeed); (Dump, any); any)

By using this conflict relation to validate transactions, a system can permit much more concurrency than by using the old one. For example, in the old one, an *Insert* operation always invalidates another *Insert* operation; while in the new one, an *Insert* operation invalidates another *Insert* operation only when both of them intend to insert a *capa* into the directory with the same key string and both of them succeed. Therefore, several concurrent transactions that perform *Insert* operations may all pass their validations.

6.5 Concepts of Object Orientation

Object orientation provides better paradigms and techniques for constructing reusable software components and easily extensible libraries of software modules. The advantages of object orientation are mostly provided by three fundamental concepts: ***abstract data types***, ***inheritance*** and ***object identity***. This section provides a brief introduction to these concepts which will be used within the system to implement the mechanisms for supporting the implicit approach.

6.5.1 Data Abstraction

Data types describe a set of objects with the same representation. There are a number of operations associated with each data type. Abstract data types extend the notion of a data type through "hiding" the implementation of the user-defined operations associated with the data type. An abstract data type specification describes a set of objects not by an implementation, but by the list of *operations* available on the objects, and the formal *properties* of these operations. Abstract data types provide a mechanism whereby a clear separation is made between the interface and implementation of the data type. The implementation of an abstract data type is hidden. Hence, alternative implementations could be used for the same abstract data type without changing its interface.

A data type is thus viewed as a set of operations offered to the outside world. Using abstract data type descriptions, users do not care about what a data type is; what matters is what it has — what it can offer to other software elements, *i.e.* the implementation is hidden from users of the data type. This information hiding capability allows the development of reusable and extensible software components. Abstract data type descriptions preserve each module's autonomy in an environment of constant change, every component must mind its own business. It must only access other components' data structure on the basis of their advertised properties, not the implementation that may have been chosen at a certain point of system evolution.

Languages that support abstract data types provide constructs to directly define data types and the operations used to manipulate instances of the data types. In addition, all manipulations of instances of the data type are done exclusively through operations associated with the data type. *Class* is such a construct most commonly provided in object-oriented programming languages, *e.g.* C++ [Str86], Eiffel [Mey92].

6.5.2 Inheritance

Progress in either reusability or extensibility demands that users take advantage of the strong conceptual relations that hold between classes: a class may be an extension, specification or combination of other classes. Inheritance provides users the support to record and use these relations.

Inheritance achieves software reusability and extensibility. Through inheritance users can build new classes on top of an existing hierarchy of classes. This avoids redesigning and recording everything from scratch. New classes can inherit both the behaviour and the representation from existing classes. Inheriting behaviour enables code sharing (and hence reusability) among classes. Inheriting representation enables structure sharing among data objects. The combination of these two types of inheritance provides a very powerful software modelling and development strategy. Inheritance also provides a very natural mechanism for organising information. It "classifies" objects into well-defined inheritance hierarchies.

If class B inherits from class A, all the features of A are automatically available in B, without any need to further define them. B is free to add new features for its own specific purposes. An extra degree of flexibility is provided by redefinition, which allows B to take its pick in the implementations offered by A: some may be kept as they are, others may be overridden by locally more appropriate ones.

The mechanism that allows a class to inherit from more than one immediate parent is called *multiple inheritance*. With multiple inheritance, users can combine several existing classes to produce combination classes that utilise each of their multiple super classes in a variety of usages and functionalities.

6.5.3 Object Identity

Object identity is a property of an object that distinguishes the object from all other objects in the application. Using object identity, users can dynamically construct arbitrary graph-structured composite or complex objects, objects that are constructed from subobjects.

In conventional programming languages, identity is realised through memory addresses. In databases, identity is realised through identifier keys. User-defined names are used in both languages and databases to give unique names to objects. Each of these strategies compromises identity, and corrupts the computational model of the language.

In a completely object-oriented system, each object will be given an identity that will be permanently associated with the object, immaterial of the object's structural or state transitions. Identity is internal to an object. Its purpose is to provide a way to represent the individuality of an object independently of how it is accessed, what it contains, or where it resides. With object identity users can referentially share objects. Object identity provides the most natural modelling primitive, allowing one object to be a sub-object of multiple parent objects. Without object identity, it would be awkward (if not impossible) to assign autonomous objects as values to instance variables.

6.5.4 Summary

This section provided a brief introduction to object orientation. Object orientation provides better paradigms and techniques for constructing reusable software components and easily extensible libraries of software modules. This enhances the extensibility of programs developed through object-oriented methodologies.

The advantages of object orientation are mostly provided by three fundamental concepts: abstract data types, inheritance, and object identity. An abstract data type describes a collection of objects with the same structure and behaviour. Abstract data types extend the notion of data types through hiding the implementation of the user-defined operations associated with the data type. Through inheritance users can build new software modules on top of an existing hierarchy of modules. Inheriting behaviour enables code sharing (and hence reusability) among software modules. Inheriting representation enables structure sharing among data objects. Object identity is that property of an object that distinguishes each object from all others. With object identity objects can contain or refer to other objects.

6.6 Constructing Atomic Objects

Besides specifying object representation and object operations, in order to construct atomic objects a programmer must implement the functionality of local atomicity. This is a difficult task. In order to lessen the programmer's burden, this thesis proposes an implicit approach for implementing atomic data types. That is, programmers construct objects assumed in a serial and reliable environment and depend on the system to provide synchronisation and recovery. A problem of this approach is how to integrate system-provided synchronisation and recovery into different kinds of atomic objects. That is, how to ensure type-specific concurrency control is still available within such an approach. This is the topic of this section.

6.6.1 The Type-Inheritance Method

One of the key concepts of object orientation is type inheritance, which permits new types to be derived from, and inherit the properties of, old types. By using this concept, a system can provide an implicit approach for implementing atomic objects while still permitting objects to support type-specific concurrency. The method is quite straightforward. A special type is constructed to provide a specific concurrency control protocol. User-defined types can then inherit this underlying concurrency control facility by use of the type-inheritance facility available in object-oriented languages. By allowing particular types to provide their own operation semantics, objects can provide type-specific concurrency if the concurrency control mechanism can make synchronisation decisions based on these semantics. It is obvious that not every concurrency control protocol can be used for the purpose, but only those protocols that can be implemented independently from objects and that can use the semantics of object operations for providing type-specific concurrency.

In the system designed in this thesis, a special type called *Scheduler* is constructed. It is an implementation of the DLV method. To provide local atomicity, user-defined atomic types inherit this method from the *Scheduler* by making use of the type-inheritance facility. The semantics of object operations is specified by users in the form of conflict relations (see Section 6.4). The logical validation of an object is done according to the conflict relation of the object, that is, the semantics of the object's operations.

6.6.2 The Interface of the *Scheduler*

The interface of the *Scheduler* is shown in Figure 6.1. It is assumed that there is only one *Scheduler* type in a system so that atomic objects would provide compatible local atomicity, consequently making it possible to ensure global atomicity.

```
class Scheduler {
    public:
                Scheduler();
    int         create(Tid);
    int         invoke(Tid);
    int         validate(Tid);
    int         object_commit(Tid);
    int         object_abort(Tid);
    };
```

Figure 6.1: The Interface of the Scheduler

The constructor *Scheduler* should be invoked to initialise an object whenever it is declared.

Before any operation can be executed on it, an atomic object should be activated. This can be done in either of the following ways: by calling the *invoke* operation, if the object exists; or by calling the *create* operation, otherwise. After an atomic object is activated, the calling transaction is registered with the object, so that it can call operations defined on the object.

The operation *validate* should be called when the transaction manager intends to ask participant objects to vote for a transaction. The *validate* operation performs logical validation according to the conflict relation of the object. Conflict relations are specific to atomic data types.

When the transaction manager has decided to commit a transaction, it should ask every participant object to commit that transaction locally by invoking the *object_commit* operation. This operation applies the transaction's intention updates on the object state. It first does a physical validation to check whether the shadow copies created for the transaction are still valid. If the validation result is *accept*, it merges the shadow copies into the object state. Otherwise, it re-executes the data operations done by the transaction on the object, and then merges the new created shadow copies into the object state.

The *object_abort* operation is responsible for aborting a transaction locally, *i.e.* undoing all the effects of a transaction on an object. By using the DLV method, aborting a transaction can be done by simply discarding the shadow copies created for the transaction.

In summary, the *validate* operation performs the *logical validator's* work; the *object_commit* and *object_abort* operations perform the *write phase manager's* work; while other operations perform the *cooperation manager's* work (see Section 3.3).

6.6.3 Constructing User-Defined Atomic Data Types

This subsection shows how user-defined atomic data types can be constructed through a simple example, a bank account. Assume that a bank account has an associated set of operations: *credit* money to an account, *debit* money from an account, and *check* the balance of an account. To construct an atomic data type, a programmer needs to define

```
class Account::Scheduler
{
          Money  amount;
public:
                 Account();
          int    credit(Money);
          int    debit(Money);
          Money  check();
conflict relation:
          ((credit, succeed); (check, succeed); =)
          ((debit, succeed); (check, succeed); =)
          ((debit, succeed); (debit, succeed); =)
          ((credit, succeed); (debit, over); =)
};
```

Figure 6.2: The Class Account

the object state and object operations in a serial environment as well as the conflict relation of the type. The functionality required by an account may be represented by the class *account* definition (illustrated in Figure 6.2). Here, the *amount* is used to record the current balance of the account, followed by the operations provided by the type and then the *conflict relation.* Notice that the concurrency control facility need not be defined here

```
        Account::Account()
{
        amount = 0;
};

int     Account::credit(Money money)
{
        amount = amount + money;
        return Success;
};

int     Account::debit(Money money)
{
        if (amount – money ≥0)
        {
          amount = amount – money;
          return Success;
        }
        else
          return Overdraw;
};

Money Account::check()
{
        return amount;
};
```

Figure 6.3: The Implementation of Class Account

for it has been inherited from *Scheduler*.

The conflict relation has four items: a successful *credit* invalidates a successful *check*; a successful *debit* invalidates a successful *check*; a successful *debit* invalidates another successful *debit*; and a successful *credit* invalidates an attempted *overdraft*. The implementation of object operations is illustrated in Figure 6.3[1].

In the account example, it is obvious that little extra work is introduced to implement a user-defined atomic data type compared with implementing it in a serial environment.

6.6.4 Remarks

The type-inheritance facility of object-oriented languages provides a very powerful means of constructing objects that inherit properties of higher level objects. This thesis makes use of this facility to provide local atomicity for user-defined atomic data types. The approach permits local atomicity to be added to objects in a very simple and flexible manner. Programmers can change the concurrency level of a type without re-implementing object operations. Changes in the implementation of the special type *Scheduler* would not affect atomic data types which make use of it. The use of type inheritance has also enabled the design and implementation of a concurrency control scheme that is highly adaptable and flexible without designing a new language or system.

Like our approach, Arjuna [Par88] and Avalon [DHW88] also take the type-inheritance approach for implementing user-defined atomic data types. However, there is a large difference. We use type inheritance to directly provide a synchronisation mechanism for user-defined atomic data types. On the contrary, Arjuna and Avalon use type inheritance to provide facilities by which programmers themselves implement synchronisation mechanism for user-defined atomic data types. Our approach, Avalon and Arjuna are all good examples illustrating that type inheritance provides an effective way to construct atomic data types.

6.6.5 Summary

In this section, we described an approach by which atomic data types can be constructed. Generally speaking, to implement an atomic data type, there are three things that need to be done: implementing the serial specification, *i.e.* the object representation and

[1] In this example, it is supposed that data persistence is supported by the language. The implementation of this property is discussed in Chapter 8.

object operations in a serial environment; implementing the atomicity operations, *i.e.* the operations used for providing local atomicity; and describing the conflict relation.

To enable user-defined atomic data types to be implemented easily, a special type called *Scheduler* is implemented by the system, in which all the atomicity operations are implemented. User-defined atomic data types can inherit these operations from *Scheduler* to provide local atomicity, instead of implementing these operations themselves. Taking such an approach, implementing user-defined atomic data types becomes similar to implementing ordinary data types, except that users need to specify the conflict relation of the type.

Only the interface of the *Scheduler* is described in this section, its implementation will be addressed in Chapter 7.

6.7 Constructing Transactions

In the last section, we showed how user-defined atomic data types can be defined and implemented. In this section, we describe how transactions that access atomic objects can be constructed.

Transactions in this thesis are implemented as objects. Thus a class, called *Transaction*, is constructed, its instances being transactions. Programmers construct a transaction by declaring an instance of *Transaction* and encapsulating computations by the primitive operations provided by *Transaction.*

6.7.1 The Transaction Model

In this thesis, a transaction is assumed to be a passive entity that controls the outcome of the operations it encapsulates. Each operation accesses only atomic objects that provide local atomicity by using the DLV method.

There are three primitive operations which may be used to declare and control a transaction: *begin_transaction, commit_transaction* and *abort_transaction.* The *begin_transaction* operation starts a transaction that may be terminated by either the *commit_transaction* or the *abort_transaction* operation. A transaction terminated by the *abort_transaction* operation will definitely be aborted, its partial results will be removed. A transaction terminated by the *commit_transaction* operation is not guaranteed to be committed successfully, but programmers will be told the outcome of the commitment. During the commitment of a transaction, *i.e.* when performing the *commit_transaction* operation, transaction managers and participant objects cooperate with each other according to the D2PC protocol to ensure the transaction's atomicity over all participant objects. The outcome of the *commit_transaction* operation is either a *success*, when the effects of the transaction become permanent; or a *failure*, when the effects of the transaction are removed.

6.7.2 The *Transaction* Class

In this thesis, transactions are declared and managed as objects. A class called *Transaction* is constructed, which provides the three primitive operations required to declare and control transactions: *begin_transaction, commit_transaction* and *abort_transaction.*

```
class Transaction
{
private:
          Vote          prepare();
          void          commit();
          void          abort();
          ...
public:
                        Transaction();
          Trans_Status  begin_transaction();
          Trans_Status  commit_transaction();
          Trans_Status  abort_transaction();
};
```

Figure 6.4: The *Transaction* Class

Besides the three primitive operations, the class declaration in Figure 6.4 also shows some private operations (*prepare, commit* and *abort*) which are used to make up the first and the second phase of the D2PC protocol.

6.7.3 Constructing Transactions

To construct a transaction, programmers need to declare an instance of the *Transaction* class in a program, and then encapsulate computations by the *begin_transaction* operation

```
Account      John, Guang;
Transaction  T;

T.begin_transaction();
if (John.debit(1000) == Success)
 {
  Guang.credit(1000);
  T.end_transaction();
 }
 else
  T.abort_transaction();
```

Figure 6.5: A Transfer Transaction

and the *commit_transaction* operation. If during the execution of the computation it becomes necessary to abort the transaction then the *abort_transaction* operation is invoked. After a transaction is completed, the instance of *Transaction* can be reused to construct new transactions. However, two transactions should not be interleaved.

Figure 6.5 shows a transaction that intends to transfer some amount of money from one account to another. At first it tries to debit £1000 from account *John*, and if the debit succeeds, it credits £1000 to account *Guang* and commits; otherwise, it aborts.

6.7.4 Remarks

In this thesis, transactions are implemented as objects. There are also other approaches to constructing transactions. One approach is to produce a programming language that includes syntactic extensions which correspond to transaction declarations by either defining a new, or modifying an existing programming language. The resulting transaction syntax may then be used by the language's compiler to generate the necessary support for transactions. Another approach is to modify the underlying operating system to provide a set of system calls which support transactions. A third approach is a mixture of the previous two, where a language provides syntactic constructs but relies on support provided by the underlying operating system.

Each implementation approach has its advantages and disadvantages. For instance, an approach based on extending a language requires modifications to the compiler which may be non-trivial, yet the resulting transaction syntax is likely to be integrated into the language and as a result could be the easiest of the approaches to use.

A problem with language or operating system approaches is that they are closely associated with a particular concurrency and recovery method, and as a result are difficult to generalise to other methods. Implementing transactions as objects makes it easy to modify them to use new concurrency and recovery methods, or to support different implementations at the same time, since only the *Transaction* class needs to be re-implemented without any need to change its interface. Therefore, user programs that use transactions need not be changed at all.

A disadvantage of this approach is that checking the mismatch between *begin_transaction* and either *commit_transaction* or *abort_transaction* can only be done at run-time. If a language approach is employed however the compiler for the language can check the syntax of the transaction.

6.8 Summary

This chapter has described the approach taken in this thesis to construct atomic data types and transactions, and showed that in this approach both user-defined atomic data types and transactions can be easily constructed.

The approach is based on the concepts of object orientation. A special type called *Scheduler*, which provides the atomicity operations needed by atomic objects, is defined and implemented by the system. User-defined atomic data types can inherit these atomicity operations from it directly.

A small language is developed for the purpose of specifying conflict relations of objects. The language permits a lot of the semantics of object operations to be represented when specifying conflict relations, hence great concurrency may be achieved.

Transactions are treated as objects in this thesis. A *Transaction* class is defined and implemented by the system; its instances therefore are transactions. To construct a transaction, programmers need only to declare a transaction and encapsulate computations by the primitives provided by the *Transaction* class.

It is claimed that implementing atomic data types in such an approach is as easy as implementing them in a serial environment.

Chapter 7

A Prototyping Implementation

This chapter shows how the distributed transaction model and the special type *Scheduler* designed in the previous chapters can be implemented in an object-oriented language C++. The methods to interpret, represent and make use of conflict relations are also described.

A distributed transaction system consists of several virtual sites, connected by a virtual communication network. Each virtual site contains a distributed transaction manager (DTM), a local transaction manager (LTM), and some atomic objects. The LTM is responsible for coordinating local atomic objects, while the DTM is responsible for coordinating LTMs to ensure that the commitment of a transaction is atomic over all participant objects. The LTM of a site communicates with local atomic objects according to the local 2-phase commitment protocol (L2PC). The DTM of a site communicates with LTMs according to the distributed 2-phase commitment (D2PC) protocol.

For easy understanding, we first describe the implementation in a distributed system in which an object can invoke operations on remote objects in the same way as on local objects, and object operations can access objects concurrently without interfering with each other. Then we describe how these assumptions can be implemented.

7.1 The Scheduler

The approach taken in this thesis to implement user-defined atomic data types is to define a special type, called *Scheduler*, in which all atomicity operations needed for providing local atomicity are implemented. User-defined atomic data types can inherit atomicity operations directly from *Scheduler* to provide local atomicity for their objects. The *Scheduler* interface was presented in Chapter 6. We describe its implementation in this section.

7.1.1 The Definition

To provide local atomicity, an object needs to record information about transactions that share the object. The read phase manager (RPM) of an object maintains for each transaction a *performed-operation table* (POT), and an *accessed-object table* (AOT) with four sets: a *read set*, a *write set*, a *create set* and a *delete set.* The POT is used to record all operations that have been invoked by a transaction on the object with their parameters and results. The *read set* is used to record all *physical objects* that a transaction has read, and the *write set* to record all *physical objects* that a transaction has written. The *create set* is used to record the new created objects, and the *delete set* to record the deleted objects. The information recorded in the POT and the information recorded in the AOT will be used later by the logical validator (LOV) and the physical validator (PHV) respectively to validate transactions. Further, whenever a transaction begins, the RPM needs to record the timestamp of the latest committed transaction. The information is necessary for transaction validation.

The *Scheduler* and some other related types are defined in Figure 7.1 and Figure 7.2. The POT is represented by the state variable *event_table.* The AOT is implemented by the following state variables: *read_set, write_set, create_set* and *delete_set.* The state variable *last_committed* is used to record the timestamp of the latest committed transaction.

State variables *pending_queue* and *committed_queue* play a very important role in the dual-level validation (DLV) method. Both the LOV and the write phase manager (WPM) perform their functions based on them. An important property of the DLV method is that transactions are committed in their timestamp order. This property can be provided easily, if we require that at every object transactions are validated in their timestamp order, and if we make the validation phase and the write phase become an atomic operation. However, this would reduce object concurrency and availability greatly as pointed out in Chapter 5. Therefore, it is desirable, especially in a distributed transaction system, to permit transactions to be validated in an arbitrary order and to separate the validation phase from the write phase. We realise this by using the *pending_queue* and *committed_queue.*

The algorithm works in the following way. The LOV validates a transaction: if validation succeeds, it puts the transaction in the *pending_queue* with the status *valid* and begins to validate another transaction; if validation fails, the transaction is aborted. When receiving the final decision about a transaction, the cooperation manager (COM) sets the

```
struct Trans_Status {
    Status       state;
    Tid          tid;
    Timestamp    timestamp;
};

class Queue {
public:
                 Queue(int);
    int          insert(Tid, Status);
    int          remove(void);
    int          change_status(Tid, Status);
    Tid          get_next(int);
    Tid          get_head(void);
    int          location(Tid);
protected:
    Trans_Status queue[MAXTRANS];
    int          head;
    int          tail;
    int          size;
};

struct Trans_Idx {
    Tid          tid;
    Oid          oid;
    int          transIndex;
    Timestamp    committed;
};

struct Event {
    void*        (*func)(...);
    void         *argptr;
    char         *funcname;
    char         *rslt;
};
```

Figure 7.1: Types Used by the Scheduler

```
struct Read_Set {
      Oid          oid;
      Path         path;
};

struct Write_Set {
      Oid          oid;
      Path         path;
      Oid          shadow;
};

class Scheduler {
public:
                   Scheduler(int, int);
      int          trans_regis(Tid);
      int          trans_deregis(Tid);
      long         validate(Tid);
      int          object_commit(Tid);
      int          object_abort(Tid);
protected:
      Queue        pending_queue;
      Queue        committed_queue;
      Trans_Idx    trans_idx[MAXTRANS];
      Read_Set     read_set[MAXTRANS];
      Write_Set    write_set[MAXTRANS];
      Oid          create_set[MAXTRANS];
      Oid          delete_set[MAXTRANS];
      Event        event_table[MAXTRANS][MAXEVENTS];
      Timestamp    last_committed;
      void         oper_put(Tid, void* (*)(...), void*, char*, int);
      void         rslt_put(Tid, char*, char*);
      int          check(Event[], Event[]);
};
```

Figure 7.2: The Definition of the Scheduler

transaction's status to *commit* or *abort* accordingly. Meanwhile, the WPM checks the *pending_queue* from time to time to see whether the status of the transaction at the head of the *pending_queue* has become *abort* or *commit.* If it has, the WPM commits or aborts it accordingly, then removes it from the *pending_queue.* If a transaction is committed, it is put in the *committed_queue.*

This implementation ensures that transactions are committed in their timestamp order, although it permits transactions to be validated in an arbitrary order. This is because the WPM only commits a transaction when it becomes the head of the *pending_queue*, and transactions are maintained in the queue in their timestamp order.

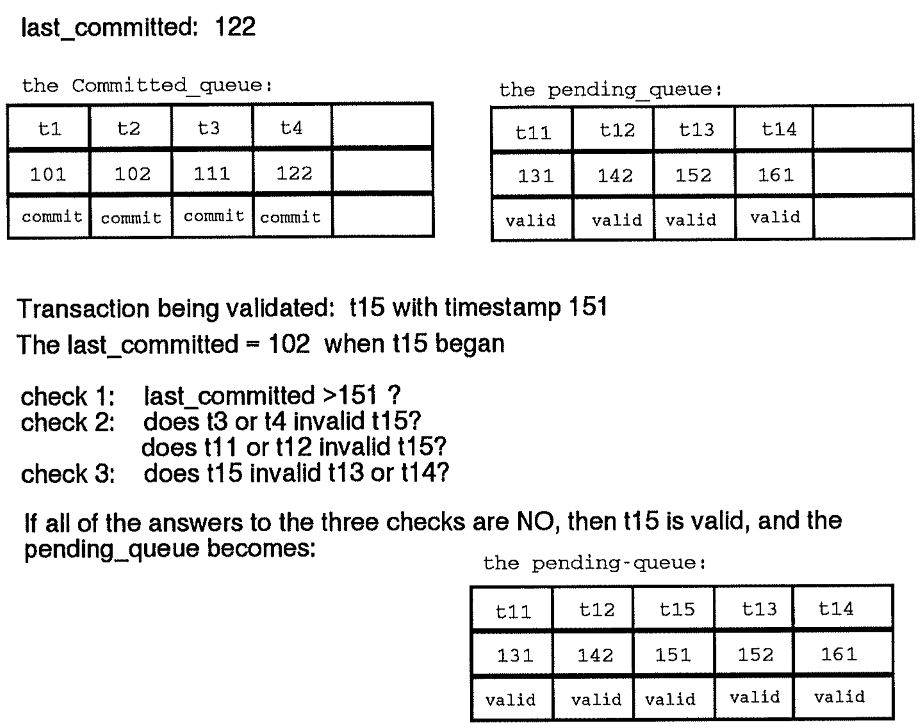
Furthermore, this implementation makes validating a transaction, getting the final decision about a transaction and applying the updates of a transaction into three independent actions. As soon as the LOV has finished validating one transaction, it can begin validating another without needing to wait for the completion of the first. A transaction that has got its final decision but has not become the head of the pending queue needs to be held until all transactions in front of it have been completed. It is worth pointing out, however, that the application program does not need to wait for the completion of a transaction. It can continue its work immediately after receiving the final decision about the transaction.

7.1.2 Validating a Transaction

To validate a transaction *T*, the LOV needs to check whether other transactions have invalidated *T*. Two kinds of transactions may invalidate *T*: transactions that committed after *T* began, and transactions that were validated after *T* began but have not yet committed and are older than *T*. The first kind of transactions should have been recorded in the *committed_queue*, and the second kind of transactions, in the *pending_queue.* In the example shown in Figure 7.3, where transaction t15 is being validated, t3 and t4 are examples of the first kind of transactions, and t11 and t12 are examples of the second kind.

On the other hand, the LOV needs to check whether *T* would invalidate any transaction that has already passed its validation. Transactions that may be invalidated by *T* are those that have passed their validations but have not yet committed and are younger than *T*. In the example shown in Figure 7.3, t13 and t14 are such kind of transactions.

Furthermore, the LOV needs also to check whether the latest committed transaction is younger than *T*. If it is, the validation fails because transactions must be committed in their timestamp order. This check can be done by comparing *T*'s timestamp with the



last_committed: 122

the Committed_queue:

t1	t2	t3	t4	
101	102	111	122	
commit	commit	commit	commit	

the pending_queue:

t11	t12	t13	t14	
131	142	152	161	
valid	valid	valid	valid	

Transaction being validated: t15 with timestamp 151
The last_committed = 102 when t15 began

check 1: last_committed >151 ?
check 2: does t3 or t4 invalid t15?
does t11 or t12 invalid t15?
check 3: does t15 invalid t13 or t14?

If all of the answers to the three checks are NO, then t15 is valid, and the pending_queue becomes:

the pending-queue:

t11	t12	t15	t13	t14
131	142	151	152	161
valid	valid	valid	valid	valid

Figure 7.3: Validating a Transaction

last_committed variable. In the example shown in Figure 7.3, *last_committed* = 122, which is smaller than the timestamp of transaction t15, and hence t15 passes this check.

To check whether a transaction T_1 may invalidate another transaction T_2, the LOV needs simply to check whether any event in the *event_table* of T_1 may invalidate any event in the *event_table* of T_2 according to the conflict relation of the object.

7.1.3 Recording Events

One responsibility of the RPM is to record the events of a transaction into its *event_table.* The information in this table is necessary for validating and re-executing the transaction.

In our implementation, transactions invoke object operations directly and the results of operations are also returned to transactions directly; hence recording the events of a transaction must be done by the operations themselves. However, providing concurrency transparency is an aim of our design; it is inappropriate to ask programmers to write the code to perform the recording work for every object operation. A preprocessing method therefore is adopted to solve this problem.

During preprocessing, the preprocessor adds to every object operation some code which records the operation's name, parameters and results into the *event_table* whenever the operation is executed. It is easy for the preprocessor to find out the name and parameters of an operation by analysing its head. However, it is impossible for the preprocessor to get the *results* of an operation without the help of programmers. Fortunately, results of operations need only to be distinguished as *succeed* or *failed*. Therefore, if programmers can tell the preprocessor whether a return point of an operation is a normal one or an abnormal one, the preprocessor can add appropriate code at the return point to record the result. Programmers can do the job simply by writing an abnormal return in the form of "abnormal_return" instead of "return".

Since recorded events must be associated with their transactions, it is necessary for an operation to know when it is invoked who the invoker is. Therefore, during preprocessing, a new parameter is introduced for each operation, which indicates the transaction identifier (*tid*) of the invoker. The introduced parameter is transparent to programmers. That is, programmers still invoke object operations in the usual way, and the extra parameter will be added automatically by the remote object invocation mechanism (see Section 7.6 for details).

Under this implementation, whenever it is invoked by a transaction, an operation will automatically record its name, its parameters and its results into the *event_table* associated with that transaction.

7.1.4 Manipulating Physical Objects

The RPM of an object needs to record the physical objects accessed by a transaction in the *read_set* and *write_set*. So far we have not described how the RPM does it. This is the topic of this subsection.

The state of an atomic object is represented by a physical object. It is a tree structure with primitive physical objects as its leaves. A *path* through the structure is used to identify a particular component. A path is the sequence of selectors required to reach a nested component at any level within the object structure. Physical objects are supported by an underlying system, called the persistent object store (POS) service, which provides manipulation of physical objects. The operations can be classified into four kinds:

$create()$	creates a new object and returns its identity.
$delete(oid)$	deletes the object named by *oid*.
$read(oid, path)$	reads the component selected by (*oid, path*).
$write(oid, path, val)$	writes *val* to the component selected by (*oid, path*).

The following operations are also provided for creating and merging shadow copies:

copy(*oid*, *path*)	creates a new object that is a copy of the component selected by (*oid, path*), and returns its identity.
merge(*soid*, *oid*, *path*)	overwrites the component (*oid, path*) by object *soid.*

Operations provided by the POS service are atomic—that is, the execution of an operation never appears to overlap (or contain) the execution of any other operation even when the operations are executed concurrently, and the overall effect of an execution is *all-or-nothing.*

Another important feature of the POS service is that physical objects can be accessed and updated at any granularity. Therefore, when making changes to a component of an object only that component needs to be rewritten, no other component of the object is affected at all.

Concurrency control is invisible to users, and object operations are implemented as if the above operations on physical objects were used directly. However, object operations are required to use the syntactically identical operations *phycreate, phydelete, phyread,* and *phywrite.* It is these operations which perform the RPM's work, *i.e.* recording physical objects accessed by transactions in corresponding sets. The semantics of the operations are as follows:

- *phyread*(*oid*, *path*) :
  1. if the component (*oid, path*) is in *deleted_set*, then signal error and return.
  2. if the object *oid* is in *created_set*, then read from it and return.
  3. if the component (*oid, path*) is in *write_set*,
     (a) if the component is not in *read_set*, then add (*oid, path*) to the *read_set*;
     (b) read from the shadow copy and return.
  4. if the component *(oid, path)* is in *read_set*, then read from the original object and return.
  5. if the component *(oid, path)* is not in any set,
     (a) add (*oid, path*) to the *read_set*;
     (b) read from the original object and return.
- *phywrite*(*oid*, *path*, *val*) :
  1. if the component (*oid, path*) is in *deleted_set*, then signal error and return.

2. if the component (*oid, path*) is in *create_set*, then write *val* to it and return.
3. if the component (*oid, path*) is in *write_set*, then write *val* to the shadow copy and return.
4. otherwise,
   (a) create a shadow copy for the component, get an object identifier *soid*;
   (b) add (*soid, oid, path*) to the *write_set*;
   (c) write *val* to the shadow copy and return.

- *phycreate*() :
  1. create an object, get its *oid*;
  2. add the *oid* to the *create_set*.
- *phydelete*(*oid*) :
  1. if the object is in the *delete_set*, then signal error;
  2. add the *oid* to the *delete_set*.

7.1.5 Physical Validation and Physical Write

The physical validation and physical write are performed based on the information recorded in the *read_set, write_set, create_set* and *delete_set*. To validate a transaction, an object simply checks whether the *read_set* of the transaction intersects with the *write_set* of any transaction that was committed after the transaction began. If there is such an intersection, then some values read by the transaction have been changed; consequently the transaction's results might be invalidated, and hence the operations of the transaction should be reapplied.

The physical write of a transaction is done by merging all the shadow copies created for the transaction back into the original subobjects, deleting all the objects in the *delete_set* and discarding the shadow copies.

Before merging a shadow copy into the original subobject, an exclusive lock should be set to prevent any operation from reading inconsistent data. The lock can be released as soon as the physical write is finished. The physical write of a transaction must be performed atomically, *i.e.*, either all or none of the shadow copies created for the transaction are merged into their original objects.

7.2 The Persistent Object Store Service

The POS service is designed to support the management of physical objects. To meet the requirements of the DLV method, it should provide a large, shared, and persistent object store, directly accessible from programming languages used to build complex applications. A special requirement on the POS service is that it should support *structured object representation*. That is, a highly structured object can be represented directly by the POS service, and user processes can access objects at any abstraction granularity, from basic fields such as *integer* or *char* to entire objects. Another requirement on the POS service is that it should support concurrency control at any required granularity.

The POS service required in the implementation of the DLV method has been provided by the multi-service storage architecture (MSSA) [BMTW91] developed in the Computer Laboratory, Cambridge. The MSSA is an open hierarchy of services in which both unstructured and structured data are supported. It has two basic components, a low level storage service (LLSS) [Wil92] and a high level storage service (HLSS) [Tho90]. The aim of the MSSA is to support a flexible environment for application development above the storage service boundary. At this level there is no imposed naming scheme or access control policy. Below the boundary a capability scheme is employed.

7.2.1 The LLSS

The goal of the LLSS is to provide a high performance storage service by exploiting non-volatile memory. The LLSS works in terms of storage objects which are byte sequences (also called files) named and protected by capabilities. The capability scheme of Gong [Gon89] is used in which a capability is labelled with a principal. Only the principal may use the capability. If the principal wants to give access rights to another, then it may cause a capability to be created for that principal.

Groups of files are organised into file systems by clients. Byte sequences can be created and deleted, and subsequences read and written. The length of a byte sequence can be read, and a byte sequence can be truncated by writing its length. A byte sequence may be recreated which means that it is truncated to have 0 length and then written with the data specified. Associated with each byte sequence is a lifetime, which decreases with approximately real time. A byte sequence is guaranteed to exist for at least as long as its lifetime is positive. A byte sequence may be touched to reset its lifetime.

The LLSS offers concurrency control at whole-object granularity. Locking is orthogonal to access control and data operations. Two types of lock are provided: shared locks and exclusive locks. A shared lock may be used for multiple readers, single writer exclusion or to support concurrent write sharing between externally cooperating clients.

7.2.2 The HLSS

The HLSS is built on top of the LLSS. Unlike conventional file servers, it supports storage of structured objects by offering a storage type generation scheme to its clients. It provides two *primitive types*, byte sequence and SS-id (Storage Service identifier, a capability), and a number of *type generators*, sequence, record and union.

An SS-id is an identity-based capability which is used to name and protect an object, and may be stored as a component of a structured object. In this way a component of an object can reference another object. The byte sequence primitive type is a fixed-length or variable-length tuple of bytes. The value of a byte sequence is uninterpreted by the HLSS and can be used by clients to store any kind of data.

Structured types may be generated by means of the type generators. An object of *record* type is an n-tuple of components of different types. It differs from programming language record types in that the components of a record are selected by natural number rather than by text-name. Records can be created and deleted, and read and written. An object of *sequence* type is a tuple of components of the same type. Sequences may have fixed or variable length. Sequences can be created and deleted, and subsequences read and modified. A variable-length sequence can be appended to, truncated and recreated in its entirety. The current length of a sequence can be read. A *union* is exactly one of the components of an n-tuple of components of different types. The components of a union are selected by natural number in the order in which they appear in the type definition. A union can be created and deleted and read and written. An operation is provided to determine the selector of the current variant stored.

The HLSS provides a multiple granularity locking mechanism as described by Gray [Gra79], which can be used to lock any logical component of an object. The semantics of locks is independent of the semantics of HLSS operations. For example, the HLSS does not ensure that a shared lock is held on a read operation and that an exclusive lock is held on a write operation.

For each type of object, the HLSS provides certain operations to access or manipulate them. The HLSS ensures the atomicity of these operations, *i.e.*, either the execution of an operation is complete or not done at all.

7.3 Conflict Relations

In the last chapter, we described the language for specifying conflict relations of atomic data types. However, conflict relations need to be converted to their internal form so that the LOV can use them for validating transactions.

7.3.1 The Internal Form

Internally, the conflict relation of an atomic data type is represented as an array. Each of the conflict relation's items is represented as a tuple $(event1, event2, rel)$, where an event consists of the name and result of an operation, and the *rel* stands for the relation between the object items that the operations act on. The internal form is defined by the type *Conflict_Relation* shown in Figure 7.4.

```
struct conflict{
            char oper1[NMLEN];
            char rslt1[RSLTLEN];
            char oper2[NMLEN];
            char rslt2[RSLTLEN];
            int   rel;
};

typedef Conflict_Relation conflict[NUMCONF];
```

where: $rel = 0$ means $=$
$rel = 1$ means $\geq$
$rel = 2$ means $>$
$rel = 3$ means $\neq$
$rel = -1$ means $\leq$
$rel = -2$ means $<$

Figure 7.4: The Definition of the Conflict_Relation Type

The work of converting a conflict relation from its specification to its internal form is done by a preprocessor, whose implementation is trivial, and therefore not discussed.

The conflict relation of an atomic data type is stored in the persistent object store in its internal form. Whenever an object becomes active, its conflict relation is read into the memory so that the LOV can use it for validating transactions.

7.3.2 Checking Transactions

For validating transactions, the LOV needs to check whether a transaction invalidates another one from time to time by using the conflict relation of the object.

To check whether a transaction T_1 invalidates a transaction T_2 means to check whether any event in the *event_table* of T_1 invalidates an event in the *event_table* of T_2. An event e_1 invalidates another event e_2 if and only if (e_1, e_2, rel) belongs to the conflict relation of the object and the relation *rel* is held between the two items that are acted on. The operation *check* of *Scheduler* is defined for this purpose.

7.4 The LTM

The responsibility of the LTM of a site is to coordinate local objects according to the L2PC protocol, and to communicate with the coordinator according to the D2PC protocol. The latter protocol ensures that objects make the same decision about a transaction's outcome, *i.e.* to commit it or abort it, and that transactions commit atomically over all objects.

```
class Local_Trans_Manager
{
            Tid             trans_tab[MAXTRANS];
            Scheduler       *obj_tab[MAXTRANS][NUMOBJ];
    public:
                            Local_Trans_Manager(void);
            int             begin_subtrans(Tid);
            int             regist_object(Tid, Scheduler*);
            int             prepare_to_commit(Tid, Timestamp);
            int             local_commit(Tid);
            int             local_abort(Tid);
};
```

Figure 7.5: The Definition of the LTM

The definition of the LTM is shown in Figure 7.5. An LTM records all the transactions in which it participates in the state variable *trans_tab*; this is done when a DTM invokes the *begin_subtrans* operation on it. The *regist_object* operation is called whenever an atomic object is created or invoked by a transaction, and causes the object ID to be recorded in the state variable *obj_tab*.

The *prepare_to_commit* operation is called by a DTM when asking an LTM to prepare to commit a transaction at the end of its read phase. The operation is implemented in four steps. First, it requests for votes from every local participant object by calling the *validate* operation. Second, it makes the local decision about the transaction based on the replies from participant objects. Third, it sends the local decision to the DTM. Fourth, if the local decision is "abort", it asks participant objects to abort the transaction locally by calling the *object_abort* operation. The participant objects can be found in *obj_tab*.

The *local_commit* and *local_abort* operations are used by a DTM to propagate its final decision about a transaction to all participant LTMs. The *local_commit* operation makes local participant objects commit a transaction locally by calling the *object_commit* operation on each of them. The *local_abort* operation makes local participant objects abort a transaction locally by calling the *object_abort* operation.

7.5 The DTM

The DTM is not implemented as an independent server, but as a class, called *Transaction* (illustrated in Figure 7.6). Whenever it intends to construct a transaction, an application declares an instance of *Transaction* and then constructs transactions by using the *begin_transaction*, *commit_transaction* and *abort_transaction* operations.

The *begin_transaction* operation simply applies for a *tid* for a transaction and records it in the state variable *transid*. When an application program intends to abort a transaction, it calls the *abort_transaction* operation which makes all participant LTMs abort the transaction locally.

The implementation of the *commit_transaction* operation is a little more complex. It is an implementation of the D2PC protocol. First, it applies for a timestamp for the transaction through the *getTimestamp* operation and asks all participant LTMs to prepare to commit by calling the *prepare_to_commit* operation defined on the *Local_Trans_Manager*. Then it makes the final decision based on the replies from participant LTMs through the *makeDecision* operation. Finally, it propagates the final decision to all participant LTMs

```
class Transaction
{
    public:
                        Transaction();
            Tid         begin_transaction();
            int         commit_transaction();
            int         abort_transaction();
    private:
            Tid                    transid;
            Local_Trans_Manager*   participants[MAXPARTICIPANTS];
            ValiResult             valiresults[MAXPARTICIPANTS];
            long                   timestamp;
            TransDecision          decision;

            Tid                    getTid();
            Timestamp              getTimestamp();
            TransDecision          makeDecision();
};
```

Figure 7.6: The Definition of the DTM

by calling either the *local_commit* operation or the *local_abort* operation accordingly.

7.6 Remote Object Invocation

In previous sections, it is frequently stated that an object invokes an operation on another object. However, we have not mentioned how invocations on remote objects are done. This section addresses the issue.

Programming in a distributed system is fraught with potential difficulties caused, in part, by the physical distribution of the system itself. The complexity of the communication protocols required to make use of remote resources is often daunting, hence many researchers have attempted to overcome these burdens by adapting familiar programming techniques and metaphors to the distributed environment, for example, extending the procedure call notion to that of *remote procedure call* (RPC) [BN84]. By making the distribution of the system transparent it is hoped that the task becomes comparable with that of programming a more traditional centralised system.

7.6.1 The Object Model

A powerful paradigm in distributed computing is the client/server model. This model has been extremely useful in the design, analysis, and growth of distributed systems. The principal objective of this programming style is to make a collection of distributed services available on a network of computers or workstations. A *server* is an executing instance of one of these services, and a *client* is a program that makes use of the service at some time during its execution.

Our system supports the client/server model. Every resource or abstraction is represented by an object. A server provides service through a set of objects, and clients make use of a service by invoking operations defined on the objects. A remote object invocation is essentially a remote procedure call; the invocation specifies a target object, the name of an operation on that object, and some parameters. Remote object invocations, like remote procedure calls, are typically synchronous.

In our system, processes (threads) are coupled with objects, and permanently bound to the objects in which they execute. That is, an *active object model* is supported. In such a model, several server or worker processes (threads) are created for and assigned to each object to handle its invocation requests. Each process is bound and restricted to the particular object for which it is created. An operation is not typically accessed directly by the calling process, as in the case of traditional procedure calls. Instead, when a client makes an operation invocation, a process in the corresponding server object accepts the request and performs the operation on the client's behalf.

7.6.2 The ROI Mechanism

A *remote object invocation* mechanism is supported by our system to provide users the illusion that programs are still executing in a centralised environment. To the programmer it should not matter where in the distributed system the actual objects are located, all that is required is a means by which operation invocations can be sent to the correct objects wherever they reside. Thus programming a distributed application should be no more complex than programming a centralised application, provided that object location and access can be made transparent.

In order to provide transparency of remote object invocation, for each object resident on the server side, called the *server object*, we provide a corresponding object on the client side, called the *client object*, which provides the client interface. Each user runs a copy of the client objects, and invokes operations on the client objects to ask for service. Object binding, parameter passing and result returning are handled dynamically by the runtime system.

A client object has the same interface as its corresponding server object. However, it has a different implementation. A client object is implemented in such a way that each of its operations actually is an RPC call to the server side. The call message contains the target object's name and the operation's parameters, among other things. On the server side, when the call message arrives, the server process extracts the object's name and the procedure's parameters, invokes the corresponding operation on the server object, and sends a reply message.

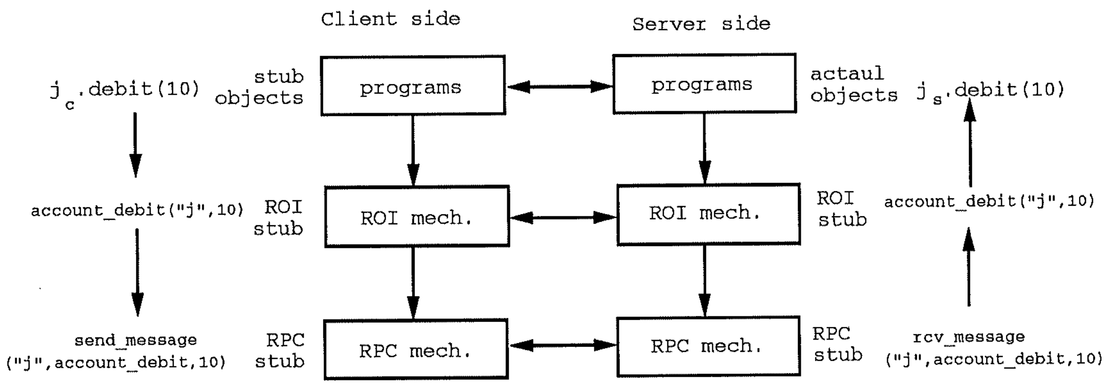


Figure 7.7: The Model of the ROI Mechanism

The remote object invocation (ROI) mechanism is implemented based on the Sun RPC [Sun88a, Sun88b] mechanism. It is illustrated in Figure 7.7 with an example. The user program presents an invocation request *debit* with a parameter (10) to a client object j_c. The *debit* operation on the client object j_c is implemented as an RPC call to the *account_debit* operation. Thus when it is executed, the RPC stub on the client side sends a call message to the server. When the RPC stub on the server side receives the call message, it extracts the object's name and the operation's parameters. Then it calls the *account_debit* operation which in turn invokes the *debit* operation on object j_s. The result is returned in the reverse way.

Since a client object has the same interface as its corresponding server object, there is no difference between invoking an operation on client objects and on server objects. That is, the remote object invocation is transparent to user programs.

7.6.3 The Stub Generator

A stub generator called *stubgen* is provided to help users to write ROI applications simply and directly. *Stubgen* is a compiler. It accepts class definitions written in C++, and produces C++ language output for ROI programs. This output includes the definition of client objects, client/server ROI stub programs, and a remote program interface definition written in Sun RPC language. The remote program interface then can be used as the input to *rpcgen* producing client/server RPC stubs which will be used by the RPC mechanism.

Accepting the class definition written in C++ as *stubgen's* input naturally enhances the transparency of the system. Programmers need only produce a single object description as if the system was not distributed and rely on the stub generator to produce the distributed version automatically.

The implementation detail of *stubgen* is extraneous to this thesis, and thus will not be presented here.

7.6.4 Naming and Binding

The binding between a client program and a server program in our system is done by the SUN RPC mechanism. It establishes binding at run-time, when a client program begins execution. The binding is explicit, *i.e.*, it is programmed by both the application and server programmers. When a server begins executing, it issues an export call, which registers the server's name and address with a network-wide name service. When a client begins executing, it issues an import call, which checks the name-server database and returns the appropriate addressing information to the client's RPC run-time system. The run-time system on the client side then performs a remote bind call to the server side, which establishes the connection to be used for the duration of the client/server interaction.

The binding between a client object and its corresponding server object is done by the ROI mechanism. A special operation *invoke* is defined on every client object. Before invoking any operation on a client object, the client program must call the *invoke* operation on it specifying the server name and the object name[1]. The ROI mechanism intercepts this invocation, makes use of the RPC mechanism to locate the specified server object, issues an *invoke* operation on the object (this registers the client program with the object), and returns the logical address of the object. Then the client object is bound to the server object, the ROI and RPC mechanism can transfer any invocation on the client object to

[1]The method for naming objects will be discussed in Chapter 8.

an appropriate invocation on the server object and possibly return a result.

The advantage of run-time binding is that the client can locate an object at execution time, which permits flexibility. The advantage of explicit binding is call-time performance. Once the binding is completed, however, the object must not move. This restriction is not severe; often the duration of binding between a client object and server object is short (during a transaction).

7.7 Implementing Concurrency

Another issue that we have not addressed yet is how concurrency inside an object is implemented. This is the topic of this section.

7.7.1 Threads

The most popular method for supporting concurrent programming is by allowing multiple lightweight *threads* within a single address space, used from a single program [Bir89]. A *thread* is a straightforward concept: a single sequential flow of control. Within a single thread, there is at any instant a single point of execution. Having multiple threads in a program means that at any instant the program has multiple points of execution, one in each of its threads. The programmer can mostly view the threads as executing simultaneously, as if the computer were endowed with as many processors as there are threads. Having the threads execute within a single address space means that the computer's addressing hardware is configured so as to permit the threads to read and write the same memory locations. In a high-level language, this usually corresponds to the fact that the off-stack (global) variables are shared among all the threads of the program. In an object-oriented language such as C++, this also corresponds to the fact that the state variables of an object are shared among all the threads which share the object. Each thread executes on a separate call stack with its own local variables. The programmer is responsible for using the synchronisation mechanisms of the thread facility to ensure that the shared memory is accessed in a manner that will give the correct answer.

Thread facilities are always advertised as being "lightweight". This means that thread creation, existence, destruction and synchronisation primitives are cheap enough that the programmer will use them for all his concurrency needs.

In general, there are three major facilities supported by a thread package: thread creation, mutual exclusion, and waiting for events. A thread is created by calling a thread creation primitive, say *thread_create*, giving it a procedure and an argument record. The effect of *thread_create* is to create a new thread, and start that thread executing asynchronously at an invocation of the given procedure with the given arguments. When the procedure returns, the thread dies.

The simplest way that threads interact is through access to shared memory. In a high-level language, this is usually expressed as access to global variables. Since threads are running in parallel, the programmer must explicitly arrange to avoid errors arising when more than one thread is accessing the shared variables. The simplest tool for doing this is a primitive that offers mutual exclusion, specifying for a particular region of code that only one thread can execute there at any time.

More complicated scheduling policies can be expressed by using a mechanism that allows a thread to block until some event happens; condition variables may be associated with a particular mutex, and with the data protected by that mutex. A "wait" operation is provided, which atomically unlocks the mutex and blocks the thread. A "wakeup" operation is supported, which does nothing unless there is a thread blocked on the condition variable, in which case it awakens at least one such blocked thread. When a thread is awakened inside "wait" after blocking, it re-locks the mutex, then returns.

7.7.2 Object Operations as Threads

This thesis uses the multiple threads technique to implement object concurrency. We use a very simple thread package for C programs, which is available in the Computer Laboratory, Cambridge.

The package does not schedule threads on time slice, but instead on the *task_yield* call. This means, if a thread captures the process, it will not release the process until it finishes or it invokes *task_yield*. Threads are scheduled in round robin order. Threads are removed from the run queue when they die. Semaphores are supported by the package to provide mutual exclusion so that threads do not interfere each other.

Each operation defined on an atomic object is implemented as a thread: all the threads of an object share the object state variables. Each state variable is protected by a semaphore. In this implementation approach for atomic objects, we know that any atomicity operation, *i.e.* an operation defined on the *Scheduler* type, may access object state variables; hence it needs to protect itself from interference by others. An atomicity operation executes a **P** operation on the semaphore associated with a state variable before accessing it, and a **V** operation afterwards. An order is defined over state variables. If an operation needs to access more than one state variable, it must execute **P** operations on them in that order, hence avoiding the dead-lock problem among threads.

Object operations defined by user programmers are only permitted to access the state variables which represent application information. These object operations are protected from errors arising when multiple threads access shared state variables by using the atomicity operations via the transaction mechanism. No special arrangement needs to be made explicitly by programmers.

As *Scheduler*, the LTM is implemented using multiple threads so that it can serve requests from multiple DTMs. However, the implementation of the DTM does not need to use multiple threads, for it is not implemented as a server.

The RPC mechanism is responsible for mapping an object operation into a thread. When a server starts, several threads are created for each object. When the RPC staff on the server side receives a request from a client, it assigns a free thread to service the request. After servicing a request, the thread becomes free again.

7.8 Summary

In this chapter, we presented a prototyping implementation of the implicit approach to implementing user-defined atomic data types. We first described the implementation of the special type *Scheduler*, then explained the internal form for representing conflict relations, and finally presented the implementation of the LTM and the DTM. Implementations were described at first under the assumption that remote object operations can be invoked transparently and that objects can be accessed concurrently without interference. Then the realisation of the assumption was explained by giving a brief introduction to the remote object invocation mechanism, and the multiple threads concept.

It is clear from the presentation that the implicit approach is not difficult to implement if a suitable persistent object store service, such as the MSSA, and a remote object invocation mechanism can be used.

Chapter 8

The Persistent Programming Language—PC++

The durability of transactions requires that the system state modified by transactions becomes permanent when they commit. This is done by using an underlying system, the POS service, in the prototyping implementation described in the last chapter. To manipulate a physical object, an object operation has to read it from the POS at first, and then write it back afterwards. There are several problems with this approach to providing data persistence. Firstly, an object has to be converted from the memory form to the persistent form that can be accepted by the POS when writing it to the POS, and *vice versa* when reading it to memory. Much space and time is taken by code to perform translations between the persistent form and the memory form. Secondly, programmers have to understand and manage the mapping which frequently distracts them from their work. The third major disadvantage is that the data type protection offered by the programming language on its data is often lost across mapping. In most programming languages the simplest way to break the type system is to output a value as one type and input it again as another. Thus the type security is lost over the persistent store.

Another approach to supporting data persistence is by using persistent programming languages [RC89, ACC81, ABC+83]. When a programming language supports persistence, an object may be declared that exists beyond the lifetime of the application program in which the object was created. If persistent objects are used to model the permanent system state then the durability of transactions may be provided by ensuring that persistent objects reflect the updates of transactions.

The motivation behind the concept of persistence is to remove the two views of storage (volatile and non-volatile) supported by conventional programming systems. Normally, an object accessed by an application exists in volatile storage and will be deallocated when that application terminates. In a conventional programming system, if the state of an object is required to exist beyond the execution of the application, then it must be converted into a form that can be stored in the non-volatile storage supported by the system. In a persistent programming system, however, data objects survive between program runs, and users can manipulate the objects with normal expression syntax, *i.e.*, physical I/O is transparent to users.

In this chapter, we describe the design and implementation of a persistent programming language—PC++, and show how atomic data types can be implemented in PC++.

8.1 Issues

Several issues arise when implementing programming languages that provide data persistence, and these are described in this section.

Object naming:
In any system that supports persistence, a mechanism is needed to enable users to name and subsequently to access an existing persistent resource. A file system is a system which offers one style of persistent resources: the *file*. In traditional file systems, such as Unix file system, a file is named by a string at the user level. In addition to the user level naming scheme, the system requires a uniform method for naming resources. This system level naming scheme can then be used by the mechanisms that support the persistence of resources. Clearly, a mapping between the two naming schemes is required, which is usually done by a directory service. A commonly used naming approach at the system level is to make use of identifiers that are guaranteed to be unique throughout the system.

Binding and type checking:
An essential property of a language with persistence is that objects in the object store can be manipulated using the same expression syntax as for volatile objects. In order to execute such an expression, however, there must first exist a binding between symbols in the program and objects in the object store. When such bindings are established, how they are specified, and to which program symbols they apply are all important issues.

Binding may be performed statically or dynamically. Static binding occurs at compile time. Once established bindings are immutable. On the other hand we may wish to obtain the latest version of the object and delay binding until we actually access it —

dynamic bindings.

If a user wishes to bind a subject to an object in persistent store then it must have a name and a description of the object to which it wishes to bind. The description is usually provided in the form of a type. Since the subject and object may be constructed separately, type checking is needed to ensure that the types match by some rule.

Data migration:
Given that persistent objects of a program reside in stable storage, we must decide the mechanism by which these objects migrate in and out of main memory during a program run. We must also decide on the granularity of data migration.

To simplify the implementation of this mapping mechanism, stable storage may be organised as an *object store*, thus providing a suitable interface for the management of objects in stable storage. When a programming system supports persistence, the mapping mechanism and the automatic movement of objects to and from the object store are provided for each class of objects.

Object activation:
A persistent object is said to be *active* while it is binding to some variables in memory, and is said to be *passive* when it is not binding to any such variable. Before invoking operations on an existing persistent object, the object must be active. The activation of an object could occur any time between the declaration of the object in the application and the invocation of the first operation on the object. When an object is created the initialisation operation provided for the class of the object is invoked.

Referential integrity:
In a system which supports persistence, attention should be paid to preserving referential integrity [MA90]. That is, if an object is pointed at or shared by two or more others, this sharing should be preserved both in their memory state and in their persistent state. Usually, a memory object is referred through its memory address. However, a method should be provided for an object to reference others in object store, and a mechanism is needed for mapping between these two kinds of references.

8.2 Object Model

A persistent programming language called PC++ is presented in this thesis. Before describing its implementation, we describe its object model in this section.

PC++ borrows and extends C++'s object definition facility, called the *class*. A class defines a type, and its definition includes both the physical representation of any instance of the class as well as the operations that may be performed on an instance.

8.2.1 Objects

An object consists of some private memory that holds its state and some methods that encapsulate its behaviour. Methods consist of code that manipulates or returns the state of an object. Methods are a part of the definition of the object. However, methods, as well as object state, are not visible to outside of the object.

An object store is a collection of persistent objects, each identified by a unique identifier, called the *object identity*. Memory is visualised as consisting of two parts: volatile and persistent. Volatile objects are allocated in volatile memory and are the same as those created in ordinary programs. They are allocated on the program stack or heap and their lifetime is bounded by the life of the program. Persistent objects are allocated in persistent store and they continue to exist after the program that created them has terminated. Interaction with these objects is routed through an object manager, but this is hidden from the programmer.

8.2.2 Object Definitions: Classes

"Similar" objects are grouped together into a *class*. All objects belonging to the same class are described by the same data structures (for the state) and the same methods (for the behaviour). Objects that belong to a class are called *instances* of that class. A class describes the form of its instances and the operations applicable to its instances.

In PC++, a new abstract data type—that is, a class is declared by the *class* construct. Class declarations consist of two parts: a specification and a body. The *specification* represents the class "user interface". It contains all the information necessary for the user of a class. The *body* consists of the bodies of methods declared in the class specification.

Class specifications have the form:

```
class <class-name>
{
            <private-members>
        public:
```

```
        <public-members>
    protected:
        <protected-members>
}
```

The construct <class-name> identifies the name of the new class definition. A member may be either a data item, called *data member*, or a method, called *member function.* The data members of a class describe the object state, and the member functions describe the operations applicable to the state.

All the data members and member functions of a class are called the *properties* of the class. Generally, properties can be classified into *public, private* or *protected* according to their accessibility. The *public* properties, which represent the class user interface, are accessible both inside or outside the class definition. The *private* properties, representing internal details of the class, can only be accessed within the class definition. The *protected* properties behave as public properties to a *derived* class; they behave as private properties to the rest of the program.

In PC++, data members can only be defined as either private or protected to enforce information hiding. This enhances reliability, since if a user can only access an object's state through public member functions, it is easier to guarantee that objects are not accidentally corrupted.

Data Members

The declaration of data members is similar to the variable declarations. Data members can be of any given *primitive type* or of a *structured type* defined by the *generators* provided.

A set of *primitive types* and a set of *type generators* are provided to define data members. A type generator represents a set of related types. Type generators enable users to generate *structured types* defined in terms of *constituent types.* Constituent types may be of any type, or a reference to an object.

It is often convenient to introduce objects whose components can refer to other objects; this is implemented by *reference.* A reference has an object as its value; two or more references, typically from different objects, may share the same object as their value.

The primitive types supported consist of the *long int*, *double*, *char*, *boolean*, *string*, *OID* and *SSID.* The structured types that may be generated by means of the generators include the *record* and *sequence.* They have a similar meaning to the corresponding generators in imperative programming languages such as C and Pascal.

The OID and the SSID Type

In PC++, a persistent object is an entity that encapsulates some private state information and a set of associated operations that manipulate the state information. Every persistent object is named and protected by an *oid* (object identifier) which is a capability labelled by a principal. An *oid* can be used only by that principal to identify an object that is unique system-wide: this protects against unauthorised access to persistent objects. An *oid* is also tagged with the identifier of its type which is used to ensure that it is manipulated correctly, *i.e.*, through the operations defined on it. The *oids* are described by the type OID.

Every object in the MSSA is named and protected by an *ssid* (storage server identifier) which is a capability labelled by a principal. An *ssid* can be used by that principal to identify an object uniquely within the MSSA. As stated earlier, PC++ uses the MSSA for providing the POS service, and in general hides it from users to provide persistent transparency. However, in some applications, users need to create and manipulate some objects directly through storage services (called storage server's objects), but at the same time, still need to manage other objects through PC++ (called PC++'s objects) and maintain relationships across these two kinds of object. To support these applications, PC++ provides the SSID type to users and permits persistent class declarations to include *ssids*. As a result, application programmers may construct and manipulate objects which contain references to any items managed within the open architecture of the MSSA.

Member Functions

Member functions of a class provide a set of operations to manipulate objects of the class. Public member functions of a class provide an interface to access its objects. Private member functions are usually used by public ones to perform certain tasks. Member functions have full access privilege to all the public, protected and private properties of the class.

One set of specialised member functions, *manager functions*, manage class objects, handling activities such as initialisation, assignment, memory management, and type conversion. Manager functions are usually invoked implicitly by the compiler.

An initialisation member function, called a *constructor*, is implicitly invoked each time a class object is defined or allocated by operator *new*. A constructor is specified by giving it the class name.

A special, user-defined member function, referred to as a *destructor*, is invoked whenever *delete* is applied to a class pointer. A member function is designated the class destructor by giving it the tag name of the class prefixed with a tilde ("∼").

8.2.3 Type Inheritance

Type inheritance allows objects to be organised in taxonomies in which the more specialised objects inherit properties of more generalised objects. Similar objects with a few different properties can be modelled by specifying a common part, called the base (super) class, for the common properties and then deriving specialised classes from this base class.

A derived class inherits all the properties of its base classes. The protected and public properties of a base class are accessible to its derived classes, but its private properties are hidden to its derived classes.

A base class is either a *private* or a *public* base class. The inherited properties of a public base class maintain their access level within the derived class, *i.e.* protected properties remain protected, and public ones public. However, the inherited public and protected properties of a private derivation become private properties of the derived class.

A derived class can add more specific properties by specifying additional data members or member functions, and can refine inherited member functions to meet new requirements. In the latter case, the redefined member functions must be declared as *virtual* functions in the base class.

Two kinds of name collision may happen when defining a derived class. One is that an inherited property's name is reused in the derived class; the other is that two or more base classes define an inherited property with the same name. When name collision happens, all of the properties are still defined on the derived class as if there had been no collision, but the inherited properties maintain their base class membership. Each can be accessed using the class scope operator.

The relationship between a base class and its derived class is a kind of type/subtype one. All the objects of a derived class can be used as if they were objects of its base class. All operations a client could perform on the objects of the base class could also be performed on the objects of its derived classes. An object of a derived class can be assigned to any object of its base class without requiring an explicit cast. However, an attempt to assign an object to another object of an unrelated class will result in a type error (compile-time or run-time).

8.2.4 Object Identity

Object identity is a property of an object that distinguishes the object from all other objects in the application. Every object of a class has an *oid* (object identity) to uniquely identify it system-wide. Whenever a new object is created, it is assigned an *oid.* Using an *oid*, a user can access the corresponding object if it has the right to do so. Another use of *oids* is to provide object sharing. Using *oids*, users can dynamically construct arbitrary graph-structured composite or complex objects — objects that are constructed from sub-objects.

8.2.5 Overloading and Dynamic Binding

One of the most powerful and useful concepts of object orientation is operation *overloading.* Overloading allows operations with the same name but different semantics and implementations, to be invoked for objects of different types. How is a particular operation name bound to a particular implementation? Typically, this is determined dynamically by the target object of the operation. *Dynamic binding* means the system binds operations to the methods that implement them at run time (instead of at compile time). The particular methods used in binding depend on the recipient object's class.

The advantages of dynamically binding overloaded operations are:

1. *Extendibility:* The same operator applies to instances of many classes without modifying its code.

2. *Development of more compact code:* Combining overloaded operators with dynamic binding allows users to avoid conditional branch constructs that check the object type and invoke the corresponding operator.

3. *Clarity:* The generated code is more readable and comprehensible. This enhances the robustness and efficiency of the development effort.

Performance costs are the main disadvantage of dynamically binding, overloaded operators. A penalty is paid for the run-time binding and/or type checking that must be done to guarantee correctness. Dynamic binding performed through a search for each invocation of each method can be expensive.

PC++ allows overloading of function names and operators. Function names can be overloaded to have a varied number of parameters of different types. System-defined operators such as +, −, ∗ and others can be overloaded for user-defined classes. PC++ also supports dynamic binding through virtual functions.

8.2.6 Accessing Persistent Object State

Member functions of persistent classes usually need to access the state of persistent objects. Given that persistent objects reside on persistent object store, it cannot be accessed in the same way as objects in the memory. However, in order to provide persistence transparency, PC++ permits member functions to access persistent data in much the same way as memory objects, but in a limited form.

PC++ does not permit persistent objects to be updated directly, but permits them to be assigned to or from memory variables by using "=" operator. This approach is chosen for two reasons: first, it allows member functions to use the operations on memory data to manipulate persistent data; second, it makes the implementation much easier.

An object, or any component of it, can be assigned to or from a memory variable. A component of an object can be referred to by giving its name path in the normal way, *i.e.*, a component of a record is referred to by its name and a sequence by its index. Type checking for assignments is done at compile time to ensure that the types of the persistent object and the memory variable match according to some rule.

8.3 The Current Implementation of PC++

This section describes how persistent classes have been implemented in the first version of PC++. It begins by defining the representation of persistent objects both as C++ objects and POS objects. Next the PC++ preprocessor and the PC++ run-time system architecture are described. Finally there is an explanation of PC++'s solution to some of the important issues that were raised earlier in this chapter.

When incorporating persistence in PC++, several principles are kept in mind:

- There should be no penalty for user programs which do not deal with persistent objects.
- There should be no change to the C++ compiler.

- Persistence should be as transparent as possible to users, *i.e.*, users can define and access persistent objects in much the same way as C++ objects.
- All properties of C++ should be kept, such as strong typing, object encapsulation and multiple inheritance.

8.3.1 Representation of Persistent Objects

The *class* mechanism of C++ and its *multiple inheritance* feature are useful to complex applications. PC++ extends C++ by making objects persistent, sharable and distributed. In order that all of the properties of C++ are preserved and that PC++ can be implemented easily, PC++ is implemented by a preprocessor.

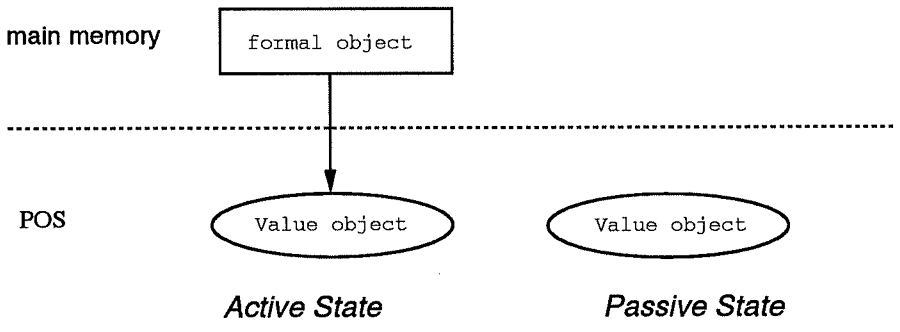


Figure 8.1: Persistent Object Compositions

In PC++, a persistent class is implemented by two parts: a definition for data members and a definition for member functions. The former is represented by a persistent data type, called the *state type*. The latter is represented by a C++ class, called the *formal class*. The formal class defines the member functions of the persistent class, while the state type defines the data members of the persistent class in a very small language accepted by the HLSS. Under this implementation, a persistent object is composed of two parts: a *formal object* that is a C++ object and a *value object* that is a POS object. The formal object represents the operation part of a persistent object, and the value object the data part. Value objects are stored in the POS, thus they are persistent and can be shared by different users. Formal objects are transient: they are created by and local to a user program, and are destroyed automatically at the end of a program session.

A user program creates persistent objects by calling the *create* operation on formal objects which should be declared in the program. The *create* operation creates the data part of the persistent object, the *value object*. An *oid* is generated for each created object which uniquely identifies the object system-wide. After a program session ends, the value objects created remain in existence in the POS, however the formal objects are destroyed. Once it is created a value object can be accessed by any program which has adequate rights.

To access an existing value object, a user program must declare a formal object which is of the same type, and then call the *invoke* operation to bind the value object to the formal object, thus making the persistent object active. After activating an object, the program can manipulate it by calling any operations defined on it. Figure 8.1 shows the object compositions in different states.

8.3.2 The PC++ Preprocessor

Users define persistent objects by using the *persistent class* mechanism. As noted, a persistent class is implemented as a formal class and a persistent data type. The work of the PC++ preprocessor therefore is to translate a persistent class into a formal class and a persistent data type.

A *class module* consists of some related persistent classes, and data types and constants which are used by those persistent classes. The PC++ preprocessor processes one *class module* at a time. It takes the definition of a *class module* as its input, and produces three kinds of output: a persistent class definition (PCD) module, a persistent class head (PCH) file and a persistent class operation (PCO) file.

Processing Specifications

A persistent class definition consists of two parts: a specification and a body. We describe how the PC++ preprocessor processes the specification part in this subsection, and the body part in the next subsection.

When preprocessing a class module, the PC++ preprocessor at first parses its data member definitions. Since a data member definition may use other data types and constants to describe an object state, the PC++ preprocessor therefore needs to process those definitions as well. If there is no error in data member definitions, a PCD module is produced for the class module. A PCD module is the meta data about a class module, which includes the appropriate type trees, constant definition tables and class lattice descriptions.

Based on the PCD module for a class module, the PC++ preprocessor generates for each class a persistent data type definition which describes the structure of data members. The persistent data type definition is written in the HLSS's type definition language, and is

INPUT: a persistent class specification:

```
struct string {
        char data[16];
};

class student {
        string  surname;
        string  forename;
        int     age;
        string  address;
        string  telephone;
public:
        void    init(string, string, int, string, string);
        string  get_address();
        string  get_telephone();
};
```

OUTPUT: the formal class specification:

```
class student {
        oid     obj_oid;
public:
        void    init(string, string, int, string, string);
        string  get_address();
        string  get_telephone();
        oid     get_oid();
        int     invoke(oid);
        int     invoke(string);
        oid     create(string);
};
```

OUTPUT: the persistent data type (for HLSS):

```
REC[BYTESEQ[16],BYTESEQ[16],BYTESEQ[4],BYTESEQ[16],BYTESEQ[16]]
```

Figure 8.2: Processing a Persistent Class Specification

treated as a string by C++. The POS uses this definition when a value object is being created or accessed.

To support object migration to and from main memory, a pair of *get* and *put* functions are generated for each class and each related data type. The *get* and *put* functions are used to read data from or write data to persistent store.

Another thing that needs to be done by the PC++ preprocessor is to produce a formal class for each persistent class. A formal class, being a C++ class, keeps the same interface as the persistent class, but does not keep any of its data members. A few data members however are introduced in a formal class for keeping run-time information, such as object bindings. Furthermore, several member functions are also introduced in a formal class which are used to create or invoke objects, or fetch an object's *oid*. Since the data part of an object is totally hidden from users and the interface of the member functions remains unchanged, user programs are not aware of any change. The formal classes produced for a class module are put in a PCH file.

In summary, a persistent class is represented by a formal class and a persistent data definition after being preprocessed. The persistent data definition describes the data members of the persistent class, while the formal class defines the operations on the data members. An example is shown in Figure 8.2.

Processing Class Body

Member functions of a persistent class are defined in the body part of the class. A member function usually includes statements that access persistent data, called *persistent statements*. Since persistent data resides in the POS, it cannot be accessed in the same way as the memory data. In order to provide persistence transparently and at the same time make the preprocessor simple, PC++ permits user programs to access persistent data in the same way as memory data (but only in assignment statements). It is the preprocessor's responsibility to translate persistent statements to appropriate function calls which read persistent data from or write persistent data to the POS. Persistent statements are labelled by a symbol $.

We have seen that during the processing of specifications the PC++ preprocessor generates a pair of migration functions *get* and *put* for each type. Therefore, preprocessing a persistent statement can be done by replacing it with an appropriate *get* or *put* function, *i.e.* the migration function for the type of the persistent data. The type of the persistent data can be found out from the PCD module, and hence also the migration function for that type. The only problem is that PC++ and the HLSS service have different mechanisms for selecting data components, and hence the name path in the statement cannot be passed directly to the HLSS. A path name mapping is needed.

Input:

```
string student::get_address()
{
  string addr;
$ addr = #address;
  return(addr);
};
```

Output:

```
string student::get_address()
{
  string addr;
  int    path[5];
  path[0] = 3;
  string_get(oid, path, &addr);
  return(addr);
};
```

Figure 8.3: Processing a Persistent Class Body

Persistent data can be accessed at variable granularity in PC++. Member functions can access any component of an object by giving the name path of that component. The name path is of the same style as that in C++, *i.e.* consisting of field names of records or index numbers of sequences. We call this kind of name path the *symbol form.* However, the name path mechanism provided by the HLSS service has another form, called the *number form*, for it only consists of numbers. In the HLSS service, all structured objects have an ordered tree structure in which the nodes represent the components. Components are selected by sequential natural numbers starting from 0. Therefore, the PC++ preprocessor must translate the name path of persistent data from the symbol form to the number form. A little attention needs to be given to the case where the index number is not a number but an integer variable or constant. The number corresponding to a field name can be found out from the PCD module.

After preprocessing, all member functions become standard C++ functions and are put into a PCO file which can be compiled by the C++ compiler. An example is given in Figure 8.3. Notice that the persistent statement in the fourth line of the input is replaced by three lines in the output. First, a variable *path* is declared for representing the name path of the component *address*. Then the index number of the *address*, 3, is assigned to *path*. Finally, a migration operation is called to read the component from the POS to the memory variable *addr* that has the same type as *address*.

8.3.3 Implementing Applications

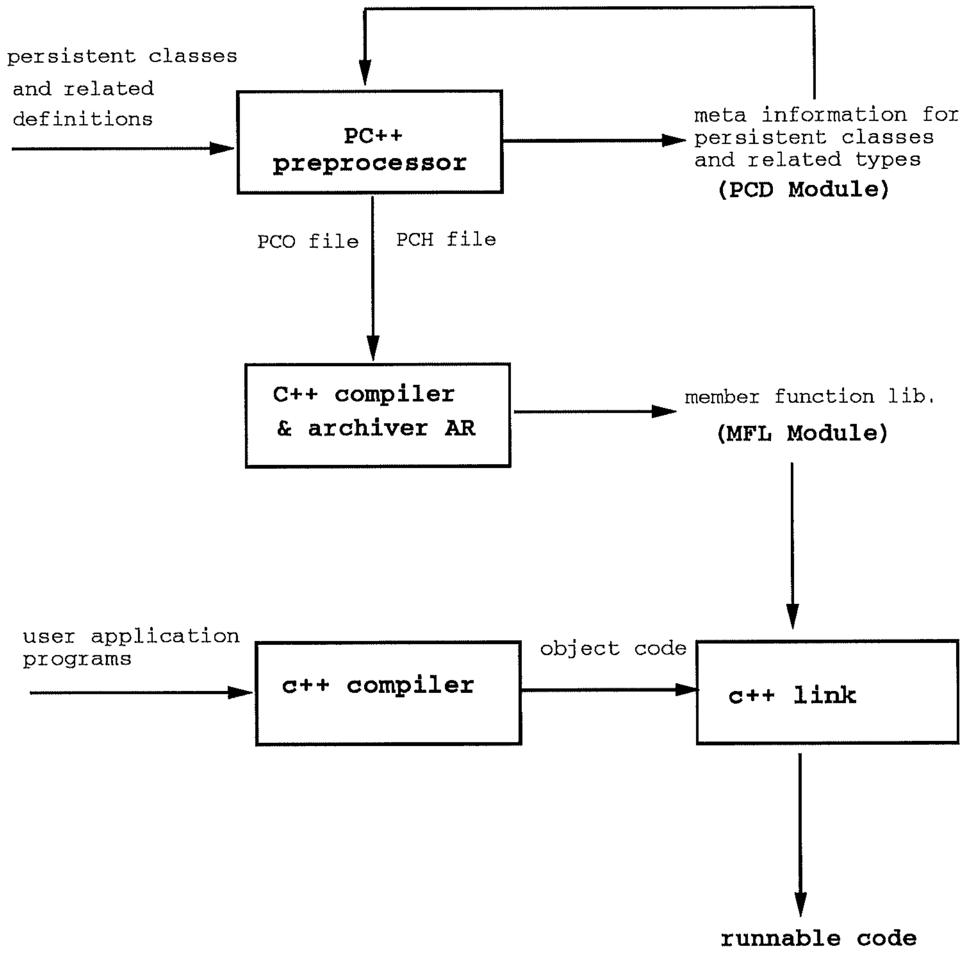

persistent classes and related definitions
PC++ preprocessor
meta information for persistent classes and related types
(PCD Module)
PCO file
PCH file
C++ compiler & archiver AR
member function lib.
(MFL Module)
user application programs
c++ compiler
object code
c++ link
runnable code

Figure 8.4: The Process of Implementing Applications

To implement applications in PC++, a user first defines various kinds of objects through the persistent class mechanism, and then hands the definitions to the PC++ preprocessor. After getting the preprocessing results, a user compiles the PCO file and forms the object code into a member function library (MFL) module which is then put into the member function library store.

Users write their application programs in the C++ language and can use any class defined in the PCH file assuming that file is included. That is, data persistence and sharing are totally transparent to application programs. Before being executed, an application program needs to be linked with the appropriate MFL modules.

Figure 8.4 shows the whole process of implementing an application in PC++.

8.3.4 Object Naming

In any system that supports persistence, a mechanism is needed to enable users to name and subsequently access an existing persistent resource. Usually, persistent resources are named by strings at the user level. In addition to the user level naming scheme, the system requires a uniform method for naming resources. This system level naming scheme can then be used by the mechanisms that support the persistence of resources. Clearly, a mapping between the two naming schemes is required. A commonly used naming approach at the system level is to make use of identifiers that are guaranteed to be unique throughout the system.

In PC++, every object has a system level name, an *oid*, which can be used to identify an object uniquely system-wide. The *oid* of an object can be used directly by users. However, an object can also be given a text name if a user wishes. The user level name of an object is not made mandatory so as to provide flexibility and good performance. Giving an object a text name is just for easy of use. However, in object-oriented systems, many objects are referred to only by other objects (this kind of reference can be done directly through object *oids*) rather than by users. Therefore, they need not have user level names at all. For example, if several objects form an object graph structure, only root objects need be accessed directly by users. Others can be accessed through the objects which refer to them. If an object is referred to directly by its *oid*, the mapping from user level name to system level name is avoided. Accesses to objects are therefore speeded up. By not making the user level name mandatory, PC++ also allows different applications to build their own user level naming scheme according to their requirements.

A simple user level naming scheme is however provided by PC++: applications can therefore use it directly if suitable. A persistent object can be given a string name by users. A mechanism is developed for mapping a text name to an *oid*. To increase the user name-space, the text name employed by a user may be further qualified by the context within which the name is used. Since objects are instances of a class, the class name of an object may be used. In addition, since a class is defined in a PCD module, the name of the PCD module can also be employed. Hence, the fully-qualified name of an object consists of three parts: *PCD module name, class name* and *instance name.* However, users need not be aware of this fully-qualified name because PC++ can map partially qualified names supplied by a user to an equivalent fully-qualified name.

8.3.5 Type Checking

As noted, PC++ is a superset of C++. Therefore, type checking of the statements or expressions in PC++ which do not involve persistent objects is the same as in C++, and is done by the C++ compiler or run-time system. However, type checking for the statements or expressions which involve persistent objects cannot be done by C++ alone, for it has no knowledge of persistent objects.

There are two cases where type checking related to persistent objects needs to be done. One is when binding a value object in the POS to a formal object in user space, which is called *object invocation.* The other is persistent object assignment. In the object invocation case, it must be guaranteed that value objects are bound to correct formal objects. Otherwise, data abstraction may be violated because after invoking an object, the user program is granted permission to access the value object through the formal object. In the assignment case, where either a component of a persistent object is assigned to a variable in user space or *vice versa*, type checking must be done to guarantee that the variable and the component are of the same type.

A persistent assignment involves persistent data and a C++ variable. Type checking cannot be done by either the C++ compiler or the PC++ preprocessor alone, because each of them has the type information only either for the persistent data or for the C++ variable, but not both. Therefore, this kind of type checking is done in two steps. When performing preprocessing, the PC++ preprocessor finds out the type of the persistent data and passes the information to the C++ compiler in some way. When compiling, the C++ compiler then checks whether the C++ variable has the same type as the type passed by the PC++ preprocessor.

The problem now becomes to find a way in which the PC++ preprocessor can pass the type information to the C++ compiler. It can be solved by using the *put* and *get* functions. It has been mentioned that when preprocessing member functions, each persistent assignment is replaced with a *put* or *get* function call which is on a per-type basis. The memory variable, which is required to be of the same type as the persistent data, is used as an argument of the function call. Therefore, if the memory variable is not of the same type, it can be rejected at compile time by the C++ compiler.

Now, let's look at an example, suppose that the PC++ preprocessor meets an assignment as below:

$ name = #pname;

where: *name* is a memory variable,

pname is a persistent data member.

The PC++ preprocessor first finds out the type of *pname*, say *nametype*, by searching the PCD module. Then it replaces the assignment with a *get* operation in the form:

report= nametype_get(oid, "cls_name", ..., path, name).

It is important to notice that the last argument in the function call above is the memory variable to be assigned. The prototype of the function *nametype_get* should look like this:

int nametype_get(long, char, ..., int[ ], nametype).*

This requires that the last argument for calling this function be of the type *nametype*. Therefore, if the memory variable *name* in the function call above is not of the type *nametype*, the C++ compiler will reject it when the function call is being compiled.

Type checking of object invocations can only be done dynamically, because only at run-time can the type information of a value object be learnt. In order that dynamic type checking of object invocations, a tag which records the identifier of its class is included in the *oid* of an object. The identifier of a class is assigned by the PC++ preprocessor and can be used to identify a class uniquely system-wide. When an *oid* is used to invoke an object by a program, a check is done to see whether the tag included in the *oid* is the same as the class identifier of the formal object to which the value object is going to be bound. Only if they are the same is the binding done. Otherwise, an exception is raised. By adding appropriate code to member functions during preprocessing, this check can be done directly in user space, thus causing little run-time penalty.

8.3.6 Binding and Data Migration

An essential property of a language providing data persistence is that persistent objects can be manipulated using the same expression syntax as for volatile objects. In order to execute such an expression, however, there must first exist a binding between symbols in the program and data in the persistent store. Given that the data resides on persistent data store, we also need to provide a mechanism for data migration.

In PC++, a value object stored in the POS is bound to a memory symbol at run-time by executing a special *invocation* operation on the symbol. That is, a dynamic binding method is adopted by PC++. The value object is named either by a text name or an *oid.* An object is said to be in the *active* state, if its value object is bound to a memory symbol; in the *passive* state, otherwise. User programs can only operate on active objects. Thus before accessing an object, user programs have to bind its value object to an appropriate memory variable.

Data migration does not happen with object binding, but during expression evaluation. Although the granularity of binding is the object, PC++ allows data to migrate in and out of user memory space at any granularity. Therefore, users can save memory space and time by migrating only those components which are really needed to evaluate an expression. No special buffer pool is created for data migration in PC++, instead, data is migrated to user space directly. If an operation intends to manipulate a component of an object, it needs to assign that component to a local variable through the "=" operator and then operate on that variable. This is a R-value binding [MA88], *i.e.*, any change on that variable afterwards has no effect on that component and *vice versa.* The state of an object can be changed by assigning a variable to a component of it through the "=" operator.

8.3.7 Data Abstraction and Protection

Persistent object systems support large collections of data that have often been constructed incrementally by a community of users. Such data is inherently valuable and requires protection from deliberate or accidental misuse [MBC+90]. Protection is required to guard against system malfunctions, to ensure users from misusing the common facilities and to protect users from other users and even themselves. In PC++ however only the last kind of protection is involved, *i.e.* to guarantee that value objects are only accessed by authorised users through the operations defined on the objects. Two levels of protection are supported. One is to guarantee data security, *i.e.* to make sure that only the authorised users access value objects. The other is to guarantee data abstraction, *i.e.* to make sure that the authorised users access value objects correctly.

Data security in PC++ is achieved by using a capability mechanism developed by the MSSA. Traditionally, a capability is a unique and cryptographically-protected name for an object, as well as being a set of access rights to an object. In the MSSA, a capability is also identity-based in that only the users to which it is issued can use it [Gon89]. A special "any" identity is provided which allows any user knowing the capability to use it.

It has already been mentioned that a persistent object is named at the system level by an *oid*. In fact, an *oid*, being a capability, is also used to protect the object.

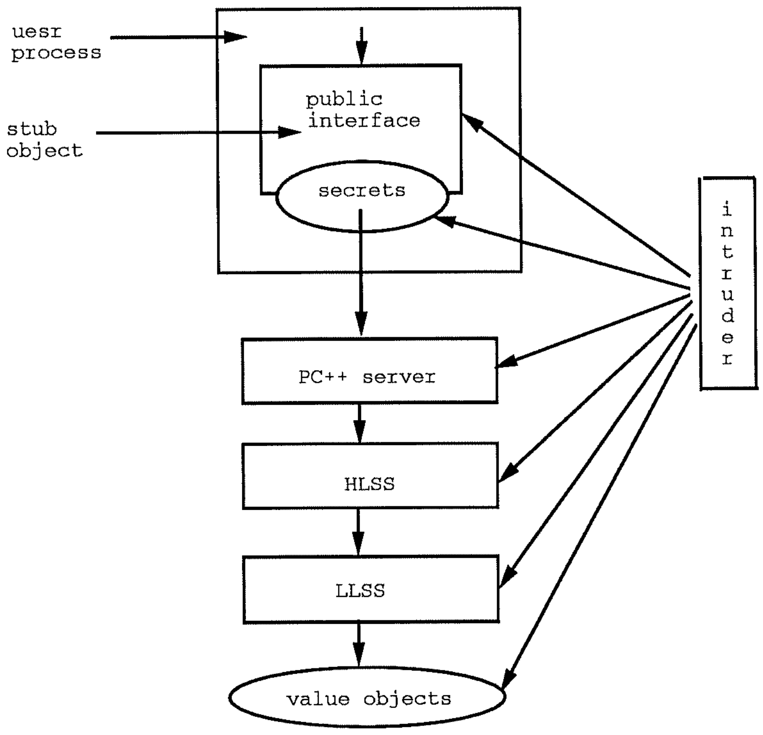


Figure 8.5: Access Control Points

All value objects in the persistent store are created by the PC++ server on behalf of users. When asked by a user to create a value object, the PC++ server creates an object in the MSSA on behalf of the user. However, instead of passing the *ssid* (the capability issued by the MSSA) of the object to the user, the PC++ server issues the user with another capability (an *oid*) which includes the identity of the user, and records the *ssid* and *oid* in a mapping table. Even if a user by chance gets the *ssid* of an object, access will not be permitted unless the user is authorised. Hence, value objects in the persistent store are protected from being accessed directly by a user rather than by the PC++ server. On requesting the PC++ server to access a value object on its behalf, a user must pass the *oid* that it possesses for that object; the PC++ server performs the access only when authentication and authorisation verification have succeeded. Therefore, by using the capability mechanism PC++ guarantees that persistent objects can only be accessed by authorised clients.

C++ can protect a formal object from being accessed by a user through an undefined operation. Therefore, to ensure data abstraction the PC++ server only needs to guarantee that a value object is bound to an appropriate formal object, and that a value object is accessed only through the formal object bound to it. Binding correctness is guaranteed by the dynamic type checking discussed in the last subsection. Furthermore, because any invocation of operations on value objects is generated by the PC++ preprocessor, as long as the preprocessor is implemented correctly the correctness of access to value objects is guaranteed.

Figure 8.5 shows the path along which a user can access a value object, and the points where an intruder might attack.

8.3.8 Referential Integrity

In a system which supports persistence, some attention should be paid to preserving the referential integrity [MA90]. That is, if an object is pointed at or shared by two or more others, this sharing should be preserved when they are migrated to the persistent store, and also when they are activated again.

In PC++, one value object can share another value object as a component by storing the *oid* of that object into it, and a formal object can share another formal object by pointing to it. When an object is activated, its data representation, *i.e.* the value object, is bound to a formal object in user space. When an *oid* is accessed, the PC++ server activates the object identified by the *oid* and returns the memory address of the object to the user process so that the user process can access the shared object through this address. Figure 8.6 shows how PC++ preserves referential integrity. A persistent object can be

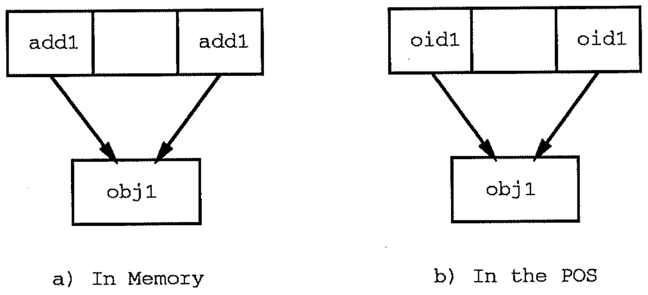


Figure 8.6: Preserving Referential Integrity in PC++

shared using several other objects by *references*. Therefore, it is possible that an object is activated more than once by a user program, which may result in a value object being bound to several formal objects. PC++ prevents this problem by keeping an active object list (AOL) for each class in a user process. Object activation is done by first checking whether the object is already registered in the AOL. If it is, then the memory address of the object is returned directly. Only when it has not been registered is the activation done. The *oid* of the object and its memory address are then inserted into the AOL.

8.4 Remarks

One of the terms coined by PS-Algol is *orthogonal persistence* [ABC+83], that is, the possibility that any object can be made persistent, independent of its type or the way it is used in the program. Orthogonality is convenient for the programmer, who may not know in advance which objects will need to persist. It is also convenient for the implementation of the run-time system, since it allows for a uniform treatment of all objects. In [ABC+83] the authors argued that in most database systems only objects of certain types were allowed to persist, and this inhibited the use of database management in many applications.

In PC++, persistence is not orthogonal to type, as it is in PS-Algol. An object may be persistent only if it is an instance of a *persistent class*. PC++ does not provide orthogonal persistence for two reasons. First, in any application there are many data structures which are known *not* to be persistent and which experience a very high frequency of access. Defining such data structures as persistent objects would unnecessarily degrade the access performance to these objects. This contradicts the first principle of the design of PC++. Second, we desired to leave the C++ subset of PC++ unaffected so that a C++ program can be accepted by PC++ without change.

Although PC++ does not provide orthogonal persistence, by taking certain measures it avoids most of the problems which might arise. First, the movement of data between long term and short term store has been simplified as an assignment. Second, strong typing is maintained by performing static and dynamic type checking at object binding and persistent data assignment. Third, references to persistent objects can be maintained in long term store using the special *ssid* type.

PC++ does not support full persistence transparency as PS-Algol does in oder to make the implementation simple. Persistent data can only be manipulated after it is assigned to a memory variable. However, this does not cause as much of a problem as one might expect. The users of PC++ can be classified into two kinds: the implementor of persistent classes, and the programmer of applications based on persistent classes. The implementor who defines the operations of persistent classes must be aware of the data movement between persistent objects and normal objects. The programmer of applications which access persistent objects through their operations, however, is not aware of this data movement.

8.5 Implementing Atomic Data Types in PC++

Since PC++ is a superset of C++, the implementation of the *Scheduler* type, the LTM and the DTM can be accepted by PC++ without change. However, the implementation of user-defined atomic data types becomes easier in PC++. This is because users can access physical objects with normal expression syntax in PC++, *i.e.*, physical I/O is transparent to users.

The *Account* example introduced in Figure 6.3 is re-implemented in PC++ in Figure 8.7.

8.6 Summary

This chapter has presented the design and implementation of PC++. Some important issues that arise in the design and implementation of persistent languages were reviewed and PC++'s solution to them were described.

PC++ extends C++ with persistent classes. A persistent class in PC++ keeps all the features of class, and its instances, called *persistent objects*, are allocated in persistent store and continue to exist after the program that created them has terminated. A persistent class is implemented by two parts in PC++: a definition for data members and a definition for member functions. The former is represented by a persistent data type, called a *state type*. The latter is represented by a C++ class, called a *formal class*.

PC++ is implemented by a preprocessor. Users define persistent objects through the persistent class mechanism, which is almost the same as the class mechanism in C++, except that the persistent class has a *confliction relation* part. The PC++ preprocessor processes persistent class definitions and produces three files: a persistent class definition, a persistent class head file and a persistent class operation file.

PC++ is integrated with the MSSA and uses the HLSS for storing both data and metadata. PC++ supports the names (SSIDs) of objects managed within the MSSA as a special type. Persistent class declarations can include SSIDs, and the application programmer may therefore construct and manipulate object data structures whose components are stored within the MSSA.

```
        Account::Account()
{
$       #amount = 0;
};

int     Account::credit(Money money)
{
        Money balance;
$       balance = #amount;
        balance = balance + money;
$       #amount = balance;
        return Success;
};

int     Account::debit(Money money)
{
        Money balance;
$       balance = #amount;
        if (balance - money geq0)
        {
          balance = balance - money;
$       #amount = balance;
          return Success;
        }
        else
          return Overdraw;
};

Money Account::check()
{
        Money balance;
$       balance = #amount;
        return balance;
};
```

Figure 8.7: The Implementation of Class Account

Chapter 9

An Application—An Active Badge System

This chapter illustrates the use of PC++ by describing an example implementation of a naming database for an active badge system. The active badge system is a good example of a potential PC++ application because it benefits from the support provided for data persistence, atomic data types and data distribution.

9.1 Introduction

Efficient location and coordination of personnel in any large organisation is a difficult and recurring problem. A solution to the problem of automatically determining the location of an individual has been to design a tag in the form of an *Active Badge* that periodically emits a unique code [WH$^+$92]. These periodic signals are picked up by a network of sensors placed around the host building. There is a master station connected with the network, which polls the sensors for badge "sightings", processes the data, and then makes it available to clients that may display it in a useful visual form.

A demonstration system was installed at the Computer Laboratory, Cambridge that was intended to be an aid for a telephone receptionist. Sensors were mounted in the offices, common areas and major corridors. The system provides a table of names against a dynamically updating field containing the nearest telephone and a description of that location. Each sighting is displayed in the following format:

name *phone* *time* *location* *(userid, office-phone)*

If the last sighting of a badge is sufficiently recent, the time field contains a percentage indicating the quality of sighting. As the last sighting ages, the time field contains a number of minutes since the last sighting, then a time of day, then a day of the week, then a date.

There are several servers in the system:

- ABNaming: manager of naming database
- ABMaster: manager of sightings database
- ABSlave: interface to sensor network
- ABImport: importer of sightings from a remote site
- ABExport: exporter of sightings to remote sites

To detect badges in transit through a building, a sensor network must provide thorough coverage through adequate placement and density of sensors. Each active badge network has an ABSlave instance associated with it. The ABSlave communicates with the badge network via a telnet connection. An ABSlave timestamps information as soon as it is received from the badge network. When an ABSlave starts up, it notifies all instances of type ABMaster that it exists.

An ABMaster initially contacts all instances of type ABSlave from which it is interested in receiving sighting information. An ABMaster will resolve active badge sightings across multiple networks, ending up with a database of sightings of the form:

- badge identifier
- last-sighted timestamp
- local network (ABSlave) where sighted
- sensor identifier on network
- sighting type (normal/multiclick)
- quality of sighting

The sightings can be retrieved through simple enquiries:

- *getbadge:* get the last sighting for a particular badge

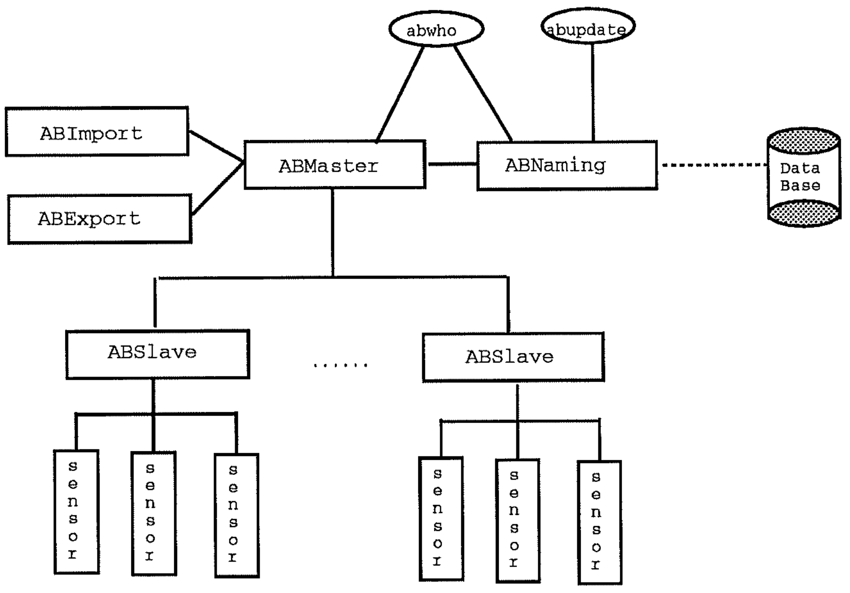


Figure 9.1: The Architecture of the Active Badge System

- *getallbadges:* get all the sightings from badges last seen at a particular sensor
- *getall:* get all sightings

The name server (ABNaming) provides descriptions for the badge and sensor identifiers returned by ABMaster. The name server also keeps a list of 'away messages', which are entered by badge wearers to specify their location when out of range of the badge system (e.g. when at a conference). The *away* entries can be automatically activated at a certain time, and cancelled by the name server when the badge is seen again.

Each ABNaming server exports a query interface for applications, allowing the applications to convert sightings from ABMaster into useful data for users.

The ABImport server is responsible for obtaining sightings from other badge sites. Translations of badge identifiers can be performed. Each site typically has one exporter server, ABExport, which can send local sightings to many remote importers.

The architecture of the active badge system is shown in Figure 9.1.

There are several display applications of the system for displaying current location information:

- *abwatch:* for watching one/many badges/sensors.
- *abwho:* for obtaining status information on one/many badges/sensors either as a snapshot or continuously.

- *xab:* an X windows version of abwho with many other features.
- *abmap:* a two-dimensional map for watching badge movements.

The official active badge system in the Computer Laboratory was developed by the Laboratory and the Olivetti Research Laboratory. The system described in this thesis is a private version that is used to demonstrate and testify the capability of PC++.

9.2 The Naming Database Server

An ABNaming server maintains a naming database which contains naming information about badges, sensor locations, sensor networks, and organisational domains. The information will be used by the ABMaster server in the process of obtaining the data from the current sensors; and used by applications such as *abwho* for obtaining descriptions of badge and sensor identifiers to convert sightings from ABMaster to useful information for users.

The naming information needs to be persistent, because it is used from time to time by applications; to be shareable, for several users may run application programs simultaneously; and to be available to remote programs, for the active badge system is intended to provide a service to all staff who may login on any machine. PC++ is an ideal language in which to implement the ABNaming server since it provides data persistence, data sharing and data distribution transparently.

9.2.1 The Definition

Information is categorised by domain. All badge, sensor and network records refer to one domain. A domain represents the logical and physical scope of an active badge system. One domain is designated to be the home domain, and the name server using the database is assumed to be running in this domain.

A badge record describes the textual name corresponding to a given badge identifier. The record also contains a *userid*, and an indication of the usual sensor location of the badge: $(home\_net, home\_stn)$. All the badge records of a domain are grouped together and are maintained by the ABNaming server of that domain. The structure of a badge record is described by the structure *Badge* shown in Figure 9.2, and the set of badges of a domain is described by a class *badges* shown in Figure 9.3.

```
struct Badges {
    int       length;
    Badge     data[MBADGE];
};
struct Badge {
    BadgeId   badge;
    int       attribute;
    char      domain[8];
    char      home_net[16];
    int       home_stn;
    char      uid[8];
    char      surname[32];
    char      forename[32];
    char      phone[16];
    int       mobile;
};
```

Figure 9.2: The Definition of *Badge*

The badge *attribute* is used to specify whether the badge is tagging a regular badge user, a piece of equipment, or a visitor. The home address is used by applications to allow a user's own phone to be specified whenever he/she is sighted at a sensor in the same group asff the home address. The *mobile* field determines whether the home phone is a mobile — if so, the mobile's number should be displayed as the local phone number for the badge wherever the badge is sighted.

Besides the badge records, the class *badges* also keeps a *timestamp* variable to remember the current version of the information. The *timestamp* will be increased whenever any change is done on the *data*. There are four query operations defined on the class *badges*: *querybadge* returns the information of the badge identified by a *badgeId*; *querybadges* returns the information of all the badges in the domain; *querybadgeuid* returns the information of the badge identified by a *userid*; and *querybadgename* returns the information of the badges named by a *user name*. The operations: *addbadge, deletebadge,* and *changebadge* are used to update the *data* contents. The *stamp* operation returns the current timestamp.

The *conflict relation* part specifies all the possible invalidations. Invalidations may only be caused by the three update operations since only these operation may change the object

```
class badges {
    private:
        int      timestamp;
        int      num;
        Badge    data[MBADGE];
    public:
        int      init(void);
        int      stamp(void);
        Badges   querybadge(BadgeId);
        Badges   querybadges(void);
        Badges   querybadgeuid(char*);
        Badges   querybadgename(char*);
        int      addbadge(BadgeId, Badge);
        int      changebadge(BadgeId, Badge);
        int      deletebadge(BadgeId, Badge);
    conflict relation:
        ((addbadge, succeed); (querybadge, failed)/(addbadge, succeed); =)
        ((addbadge, succeed); (deletebadge, failed)/(changebadge, failed); =)
        ((addbadge, succeed); (querybadges, succeed)/(querybadgeuid, failed); any)
        ((addbadge, succeed); (querybadgename, failed); any)
        ((changebadge, succeed); (querybadge, succeed)/(changebadge, succeed); =)
        ((changebadge, succeed); (querybadges, succeed); any)
        ((changebadge, succeed); (querybadgeuid, succeed); any)
        ((changebadge, succeed); (querybadgename, succeed); any)
        ((deletebadge, succeed); (changebadge, succeed)/(addbadge, failed); =)
        ((deletebadge, succeed); (querybadge, succeed); =)
        ((deletebadge, succeed); (querybadgename, succeed); any)
        ((deletebadge, succeed); (querybadgeuid, succeed)/(querybadges, succeed); any)
};
```

Figure 9.3: The Definition of Class *badges*

state.

From the specification of the conflict relation, it is obvious that by taking the operation semantics into account, the system can permit more concurrency than a traditional system which classifies operations only into *read* and *write*. Granularity is another factor that affects the degree of concurrency; by taking a badge record as the access granularity, many invalidations are excluded from the conflict relation which otherwise could not be.

9.2.2 Implementation

The implementation of member functions of a persistent class in PC++ is simple, for neither concurrency nor remote invocation needs to be considered. On the other hand a little attention needs to be paid to the fact that only the operator "=" can operate on object state. Some member functions of the class *badges* are shown in Figure 9.4; other ones can be implemented in a similar way.

Besides the badge information, the naming database must also manage the network information and the sensor information. A *network* record describes the name of a network within a domain. Badge sightings from sensors on a network are categorised by the name of the network. The set of networks within a domain is described by a class *networks*.

A sensor record gives a text name for the location of a given sensor on a network. It also describes the telephone number at that location. Sensor regions may overlap, and sensors may be grouped together to form one logical region. This is denoted by a reference to another sensor in the group. This reference may be to itself for a single sensor not in a group. The set of sensors within a domain is described by a class *stations*.

The classes *networks* and *stations* are very similar to the class *badges*, and so will not be described further.

9.2.3 Experience

In this section, the definition and implementation of the class *badges* have been described to illustrate how user-defined atomic types can be defined and implemented in PC++. As expected, defining an object is simple because there is no special requirement within PC++ for representing an atomic object. Implementing member functions of an atomic object is very straightforward; no lock needs to be set or released within the functions. Any member function is implemented as if the object is owned by itself and no others can access it. The description of the conflict relation is also simple, although it may be a little

```
Badges* badges::querybadgeuid(char* uid)
{
   Badge    items[MBADGE];
   Badges   abadges;
   abadges.length = 0;
$  items = #data;
   for (int i = 0; i < MBADGE; i++)
   {
     if (strcmp(items[i].uid, uid) == 0)
            abadges.data[abadges.length++] = items[i];
   };
   return(&abadges);
};
int* badges::addbadge(Badge abadge)
{
   int         result = FALSE;
   int         bnum, bstamp;
$  bnum = #num;
   if (bnum == MBADGE)
     return(&result);
$  #data[bnum] = abadge;
   bnum++;
$  #num = bnum;
$  bstamp = #timestamp;
   bstamp++;
$  #timestamp = bstamp;
   result = TRUE;
   return(&result);
};
```

Figure 9.4: Some *badges* Member Functions

```
Badges* badges::querybadges()
{
   Badges   abadges;

$  abadges.length = #num;
$  abadges.data = #data;
   return(&abadges);
};

int* badges::changebadge(Badge abadge)
{
   int        result = FALSE;
   int        i, bstamp;
   Badge      items[MBADGE];

$  items = #data;
   for (i = 0; i < MBADGE; i++)
   {
     if (compare_badge(&(abadge.badge),&(items[i].badge)) == 0)
$             #data[i] = abadge;
   };
   if(i >= MBADGE)
     return(&result);
$  bstamp = #timestamp;
   bstamp++;
$  #timestamp = bstamp;
   result = TRUE;
   return(&result);
};
```

Figure 9.4: Some *badges* Member Functions(continued)

hard to decide whether a member function invalidates another in some cases.

Arising from the implementation, we find out that the *set* type generator would be useful for queries. Supporting sets would save member functions from having to retrieve whole objects when searching for a particular item. Also, not supporting variable length sequences may waste storage space and make member functions a little more difficult to implement.

More experience needs to be gained with the use of *SSID* and *OID* types. The *OID* type should be useful for representing object references, and also for object sharing. The *SSID* type is introduced for naming MSSA items external to the PC++ system. Neither of these two types is used by the class *badges*. Another important feature that is not used by the ABNaming server to define objects is type inheritance. PC++ does however make use of type inheritance to provide synchronisation to atomic objects.

From the implementation of member functions, it is also shown that PC++ does not support full persistence transparency. The implementor of atomic data types is aware of some difference between persistent objects and normal objects. This is because before doing any operation on a persistent object component, the implementor has to assign it to a memory variable; and then assign it back if updates on the component needs to be persistent. However, this does not cause much inconvenience to the implementor because data migration, data coding and decoding are done automatically by the system in PC++.

It is worth pointing out that data persistence, however, is totally transparent to programmers who implement applications based on atomic data types.

9.3 Clients of ABNaming

Each ABNaming server exports a query interface that allow applications to convert sightings from ABMaster into useful data. Examples of such applications are *abwho, abwatch, xab* and *abmap* which display location information to users. The ABMaster server also uses this interface to fetch sensor and network information from the naming database.

Each ABNaming server also exports an interface for users to query and manipulate the naming database, consisting of three operations:

1. *abupdate:* for updating badge, sensor, network and domain information in the naming database.

2. *abaway:* for listing/altering *away* messages stored in the naming database.

3. *abquery:* for displaying badge, sensor, network and domain information in the naming database.

In this section, we describe the implementation of the *abwho* application and the *abupdate* operation to show how the naming database can be used.

9.3.1 The Implementation of *abwho*

The *abwho* command provides a simple interface to the information provided by the active badge system. With no arguments, *abwho* lists all active badge sightings. Each sighting is displayed in the following format:

name *phone* *time* *location* *(userid, office-phone)*

With arguments *domainnm* and *uid*, *abwho* restricts the scope of the listing to sightings textually matching the *uid* within the given domain. A simplified implementation of this operation is shown in Figure 9.5. The operation consists of five steps:

1. fetching from the naming database all the badges of domain *domainnm* by calling the *QueryBadges* operation;
2. finding out, from the results of the last step, the badge which matches *uid*, say *B*;
3. retrieving the sightings of *B* from the *ABMaster* server by calling the *QueryByBadge* operation;
4. fetching from the naming database the information about the sensors which sight *B*;
5. displaying sightings to users;

There are two operations which access the naming database: *QueryBadges* and *Query-Station*. They are implemented as two transactions which in turn invoke the corresponding operations defined on the classes *badges* and *stations*. If the *BeginTransaction* operation fails, the operations return directly to the caller with an error message; otherwise they will activate the corresponding object in the ABNaming server and then fetch the information using the appropriate operations defined on the object. Since PC++ provides remote invocation transparency, these two operations can invoke and access remote objects as if the objects are local to them.

```
void abwhobyuid(char *domainnm, char *uid)
{
 Badges      badges;
 Station     station;
 Sightings   sightings;
 int         i, j, k, l, report;

 do {
    report = QueryBadges(domainnm, badges);
 } while(report != 0);

 for (i = 0; i < badges.length; i++) {
    if (strcmp(badges.data[i].uid, uid) == 0)
    {
              {sightings} = QueryByBadge(badges.data[i].badge)
              if (sightings.length > 0)
                {
                  do {
                    report = QueryStation(sightings.data[0].domain,
                           sightings.data[0].station, &station);
                    } while(report != 0);
                    display_sighting(badges.data[i], station, sightings.data[0]);
                }
    else
              display_sighting(badges.data[i], null_station, Null_sighting);
    }
  return;
}
```

Figure 9.5: Part of the Implementation of *abwho*

```
   DisTransManager  _dtm;
   Stations         _stations;
   Badges           _badges;

int QueryStation(char* domainnm, int astation, Station* tstation)
{
   int              report;

   report = _dtm.BeginTransaction();
                    if (report != 0) goto exit2;
                    report = _stations.invoke(domainnm, &_dtm);
                    if (report != 0) goto exit;
                    tstation = _stations.querystation(domain, astation);
exit:
   report = _dtm.EndTransaction();
exit2:
   return(report);
};

int QueryBadges(char* domainnm, Badges* tbadges)
{
   int              report;

   report = _dtm.BeginTransaction();
                    if (report != 0) goto exit2;
                    report = _badges.invoke(domainnm, 100, 99, &_dtm);
                    if (report != 0) goto exit;
                    tbadges = _badges.querybadges();
exit:
   report = _dtm.EndTransaction();
exit2:
   return(report);
};
```

Figure 9.5: Part of the Implementation of *abwho* (continued)

9.3.2 The Implementation of *abupdate*

The *abupdate* command provides a method for users to update the contents of the naming database: adding, deleting, or changing a badge record, a station record, a network record, or a domain record.

```
int addbadge(char* domainnm)
{
  Badge badge;
  int i, j, report;

  Sys_Init(host, databasenm, BADGE);
  DisTransManager d1 = DisTransManager();

  d1.BeginTransaction();
       _badges.invoke(domainnm, d1);
       _changes.invoke(domainnm, d1);
       if (read_badge(badge) != SUCCESS) {
         d1.AbortTransaction();
         return(-1);
       }
       else {
         _badges.addbadge(badge);
         change.action = Add;
         change.change = Bdg;
         change.data = badge;
         _changes.addchange(change);
       }
  report = d1.EndTransaction();
  return (report);
};
```

Figure 9.6: The Implementation of Operation *addbadge*

The *abupdate* command consists of several operations each of which performs one of the following functions: adding a badge, deleting a badge, changing a sensor, *etc.* It first parses the program arguments to decide the user's intention and then calls the appropriate operation to perform the work.

A simplified implementation of the operation *addbadge* is shown in Figure 9.6. First, the *addbadge* operation sets up a connection with the ABNaming server by calling *Sys_Init* with the host machine name *host*, the naming database name *databasenm* and the ABNaming server name *BADGE*. After starting a transaction, it activates the object *_badges* which records the badge information, and the object *_changes* which is used to record the updates on the naming database. Then, it asks the user to input the badge information. If there is no input error, it adds the badge record to the database by calling the *addbadge* operation defined on the object. Finally, it composes a change record and inserts it into the object *_changes*.

Again, the implementation is very simple since it does not need to deal explicitly with remote object invocation and persistence. Other operations can be implemented in a similar way, and are hence not shown here.

9.3.3 Experience

In this section, two applications of the ABNaming server have been described to show how applications can be written using atomic objects. PC++ requires that any operation on an atomic object must be performed within a transaction. Therefore, an application consists of transactions within which atomic objects are accessed.

The implementation of applications is simple. Accessing remote objects can be done in the same way as accessing local objects, since PC++ provides transparency for remote object invocation. Transactions can be constructed easily by using *BeginTransaction*, *EndTransaction* and *AbortTransaction* operations. However, an exception handling mechanism would be very useful when invoking remote objects. Otherwise, user programs need to check and handle exceptions every time they access a remote object.

Another experience we gained from the implementation of the active badge system is that serialisability is sometimes a too strong requirement. For example, to display sightings to users, *abwho* needs to fetch the current badge and station information. Even though it is not strictly necessary for the information obtained to be up to date, PC++ still requires the fetch operation to be enclosed in a transaction, thus participating in concurrency control. This would slow down the execution of the operation. To provide more flexibility, a pseudo-transaction mechanism, called the *fast-read-transaction*, is introduced in PC++.

Read only operations may be enclosed by a pair of *BeginFastRead* and *EndFastRead* operations to form a *fast-read-transaction.* A *fast-read-transaction* does not participate in concurrency control. There is no validation phase and write phase for it, hence it can be executed much faster than a normal transaction. The result of a *fast-read-transaction* may be inaccurate, *i.e.*, it may reflect only part of the effect of committed transactions. It is the user who is responsible for deciding whether the results are accurate enough to use or not. Experience gained from the implementation of the active badge system shows that the *fast-read-transaction* mechanism is useful.

9.4 Summary

In this chapter, the implementation of a naming database in an active badge system was described to illustrate how PC++ may be used, and to test its capability and feasibility.

It is clear from the example implementation that implementing user defined atomic data types in PC++ is not a difficult task. Except for the description of conflict relation, an atomic data type can be defined as easily as for a normal object, *i.e.* assuming no concurrency and no failures. Although the implementor of an atomic data type must be aware of the special circumstances surrounding data migration, data persistence provided by PC++ makes it as simple as an assignment.

On the other hand, the implementation also shows that some other mechanisms would be useful for writing member functions, such as the *set* type generator, and *variable* length sequences. However, they are not directly related to the subject of the research.

The applications of the ABNaming server described in the last section shows that applications making use of atomic data types can be easily implemented, This is because in PC++ remote objects and local objects can be accessed in a similar way. However, an exception handling mechanism would be very useful when invoking remote objects. By supporting both the *transaction* and *fast-read-transaction* mechanisms in PC++, users have the flexibility to make compromise between strong consistence and performance according to application requirements.

In summary, the example implementation shows that PC++ provides a good interface both for implementing and using atomic data types, although there are still some features need to be improved resulted from the simplified implementation of PC++.

Chapter 10

Comparison with Related Work

The aim of this thesis is to find a mechanism by which user-defined atomic data types can be implemented easily, efficiently but still permitting great concurrency. To achieve this aim, an implicit approach was taken for implementing user-defined atomic data types. Associated with the approach, this thesis proposed an optimistic concurrency control method, the *dual-level validation* method for atomic objects to provide local atomicity; and a language for users to specify the semantics of object operations. There are a number of systems which support transactions by utilising atomic objects. In this chapter, we briefly review these systems and compare each of them with the work presented in this thesis. Emphasis in the comparison is placed on the following features:

1. *The definition approach:* Approaches taken for defining atomic data types can be divided into three classes: implicit approach, explicit approach and hybrid approach, according to whether the system or programmers are responsible for implementing the synchronisation and recovery operations.

2. *How the application information is represented:* In the case where an explicit or hybrid approach is taken for defining atomic data types, it is possible to represent application data structures and methods either independent of or related to synchronisation information. Generally speaking, the latter may permit greater concurrency. On the other hand, however, it might make the coding of the class operations more difficult.

3. *Specifying the semantics:* The conflict semantics of object operations may be specified declaratively, separately from the implementation of the operations. Alternatively, it might be encapsulated in the coding of the methods. The latter approach not only makes operations harder to implement and to verify, but it also means that

it is more difficult to effect changes.

4. *The semantics level:* What level of semantics is taken into account when specifying operation conflict? The more precise the specification, the greater is the potential concurrency.

5. *The concurrency control and recovery method:* Different systems support local atomicity using different methods for concurrency control and recovery.

6. *The persistent model:* The persistent model used by a system may affect concurrency and crash recovery of the system as well as the convenience for the programmer.

10.1 Argus

Argus [LCJS87, Lis88] is a reliable distributed programming language which provides support for nested transactions. In Argus, a distributed program consists of a collection of operations on *guardians* [LS83] which are stable, crash resistant object managers. Each guardian provides a set of *handlers* which constitute the public interface to the objects it manages. Each handler invocation creates a new process and nested transaction to manage the call. To support atomicity Argus provides *atomic data types*, instances of which, called *atomic objects*, are serialisable, recoverable and persistent.

Both inter-transaction and inter-operation synchronisation are explicit in Argus. Inter-operation synchronisation needs to be done by programmers using the built-in type generator **mutex** and the **seize** statement. The goal of inter-operation synchronisation is to ensure that concurrently invoked operations are executed in mutual exclusion. Locks set on a mutex are only held for the period that an operation accesses the object state, thus they do not protect temporary results of transactions from being seen by other transactions.

The goal of inter-transaction synchronisation is to ensure that the effects of transactions are serialisable. It not only needs to determine whether an operation from a transaction should be done, but also needs to ensure that the operation, if it could be done, does not read temporary results. Before accessing any item of an object, an operation must check the status of that item to determine whether it is usable, *i.e* to check whether the transaction which last modified that item has been committed. Inter-transaction synchronisation in Argus is done by programmers using locks on built-in atomic objects. Associated with each item of an object, there is a built-in atomic object which is used to indicate the status of that item: *present* if the transaction that modified that item has committed; or *absent* otherwise. Any operation which modifies an item sets the object status as *absent* at first, once the transaction has committed, the object status becomes *present*.

There are two ways to complete a transaction in Argus: in the implicit approach no user code runs when a transaction completes; however in the explicit approach, it is the programmer who supplies the code which will run when completing a transaction.

Usually, the state of an atomic object in Argus is represented as a history (log) of previously executed transitions. The execution of an operation on the atomic object is implemented as an addition to this collection of transitions. Explicit synchronisation is needed before the addition to determine whether the current operation can proceed immediately or has to be retried. The goal of the synchronisation is to decide whether the serialised sequence generated at this atomic object is still valid after the new transition is added. Information kept in the history of previously invoked transitions is used to make that decision.

Argus provides a number of built-in atomic data types and constructs which enable user-defined atomic data types to be implemented. Typically, the representation of an object state is a combination of atomic and non-atomic objects, with the non-atomic objects used to hold information that can be accessed by concurrent transactions, and the atomic objects containing information that allows the non-atomic data to be interpreted properly. The entire representation is enclosed in a mutex, which is used to provide mutual exclusion for user processes.

Argus is designed to support implementation of *(pessimistic) dynamic* atomic data types based on a generalisation of strict two-phase locking. The built-in atomic data types employ the multiple reader/single writer policy for locking, whereas user-defined atomic types can implement type-specific concurrency control to provide increased concurrency. A *general locking protocol* [Wei84] is used by Argus, which permits partial and non-deterministic operations. That is, operations may not be defined on some states and invoking an operation on a state may not always have the same result. The other difference between the general locking protocol and normal locking protocols is that it permits information about the result of executing an invocation to be used when scheduling invocations. Therefore, it permits more concurrency than normal locking protocols which restrict invocations to be total and deterministic. Argus recovers directly from the history object. A transaction can be aborted simply by deleting its transitions from the history object. No further recovery is necessary. Recovery operations must be provided by programmers in the explicit approach.

A system must record enough information in secondary storage so that an object can be restored to a consistent state should the node on which it resides fail. In Argus, objects are kept in volatile memory while they are used by transactions, and are written to stable storage when a transaction that modifies them commits. Argus applies the *commit log* technique, *i.e.* recording the relative changes made to the object since some previously recorded state. The system knows when a built-in atomic object needs to be copied to stable storage; however, the programmer must tell the system what mutex objects should be written to the stable storage by using the mutex operation **changed**.

When a node crashes, all guardians residing at that node become inaccessible. When a node recovers, its guardians restart, and the information in stable storage is used to restore the states of the stable objects.

10.2 Arjuna

Arjuna [SDP91, Par88, Dix88] is an object-oriented programming system which provides a set of tools for constructing fault-tolerant distributed applications. Arjuna supports nested transactions for structuring programs. Programs invoke operations on atomic objects. In Arjuna, objects are long lived entities and are the main repositories for holding system state.

Inter-transaction synchronisation in Arjuna is done explicitly by using locks. Associated with each object there is a lock variable to indicate the current status (*held* or *retained*) and the current mode (*read* or *write*) of the object. Operations are classified as *readers* or *writers*. The locking scheme used by Arjuna is the well known pessimistic method of single writer/multiple readers. Before accessing an object, an operation must check the lock variable to see whether the operation could cause conflict. When no conflict arises, the operation can access that object; otherwise it needs to wait until the lock is released. In order to ensure that strict two-phase locking is followed, *i.e.*, any lock cannot be released until the transaction commits or aborts, Arjuna requires that the *releaselock* operation is called by the transaction system when a transaction commits or aborts rather than directly by the programmer. Thus, an implicit approach is taken by Arjuna to complete a transaction.

Note that no inter-operation synchronisation needs to be done in Arjuna for it does not support type-specific concurrency. If any transaction holds a read lock on an object, then no transaction holds a write lock, so all are free to read the object without fear of its being modified as they read it. Conversely, if one transaction holds a write lock on an object, no other transaction can hold either type of lock, so it need not fear interference. This simplicity is achieved by sacrificing potential concurrency.

Concurrency control in Arjuna is implemented by at first defining a basic concurrency controller, called *LockCC*, which is based on the common technique of two-phase locking; user-defined atomic data types can then make use of it. The type *LockCC* 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.

User-defined atomic objects can be derived from the concurrency controller, thus they are able to utilise inherited operations to set locks on its instances. User-defined atomic objects need only to define the representation of application information and the representation of the lock variable.

The default mechanism for supporting recovery in Arjuna is to take a snapshot of the state of an object before it is modified for the first time within the scope of a transaction. If the transaction aborts then the old state can simply replace the new one, thereby achieving recovery.

The current Arjuna implementation makes use of the Unix file system for long term storage of objects, with a class *ObjectStore* providing an object-oriented interface to the file system. When not in use persistent objects are stored in a passive form in an object store. When first used they are automatically activated by the system which results in the conversion of the object into its active form. Deactivation occurs when the transaction commits at which time the object is again converted back into a passive form. Conversion of objects between their passive and active form is controlled by the system but uses operations supplied by the programmer. These operations must be provided otherwise neither the recovery nor the persistence mechanism will function properly.

10.3 TABS

The TABS (TransAction Based System) prototype [SBD+85] developed at Carnegie-Mellon University provides support for distributed transactions that operate on atomic data types that do type-specific synchronisation and recovery. Objects in TABS are instances of atomic data types and are encapsulated in processes called *data servers*. An operation on an object is invoked via a request message to the data server. In TABS, the responsibility for recovery and synchronisation is divided between the system and individual types in a fashion that contributes to efficient operation and provides for composition of operations on different objects. The programmer indicates in each operation the kind of lock that should be set on the object by using the *LockObject* routine; however, it is the system that is responsible for ensuring that locks are set and released in accordance with the rules of two-phase locking.

In TABS, data servers support synchronisation requirements of transactions by using locking to synchronise access to the objects they read and modify. To achieve increased concurrency, data servers can exploit the semantics of operations by using type-specific locking. Every operation on an object acquires a lock from the set of lock modes associated with that object. A type-specific lock compatibility relation is used to determine whether a lock may be acquired by a particular transaction. 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. For reasons of efficiency, data servers may encapsulate many objects and these may be of more than one type.

Two separate write-ahead log algorithms are implemented in TABS. One is based upon *value logging*, in which the undo and redo portions of a log record contain the old and new values of an object's representation. Upon transaction abort, the recovery manager follows the backward chain of log records that were written by the transaction and sends messages to the servers instructing them to undo their effects. Another is an operation-based recovery algorithm, in which data servers write log records containing the names of operations and enough information to undo them. Operations are redone or undone, as necessary, during recovery processing to restore the correct state of objects.

In TABS, inter-operation synchronisation is also done using exclusive locks. When type-specific locking is used by an atomic object, in which case different transactions may concurrently update the same object, the operation-based recovery mechanism must therefore be used. Otherwise, an aborted transaction cannot be recovered.

TABS supports data persistence by using the underlying facilities provided by the *Accent kernel*. Accent provides a kind of memory object called the *recoverable segment*. Recoverable segments are backed by non-volatile storage. The backing storage for a recoverable segment is permanently assigned, and the paging system updates this storage directly.

To support the write-ahead log algorithms, the kernel sends three kinds of messages to the Recovery Manager. The first message indicates that an in-memory page of a recoverable segment has been modified for the first time since it has been paged in. The second message indicates that the kernel wants to copy a modified page of the recoverable segment to non-volatile storage. The kernel does not write the page until it receives a message from the recovery manager indicating that all log records that apply to this page have been written to non-volatile storage. The third message indicates that a page in a recoverable segment has been successfully copied to non-volatile storage.

10.4 Clouds/Aeolus

The goal of the Clouds [AM83] project is the implementation of a fault-tolerant distributed operating system based on the notions of objects and transactions. In Clouds, scheduling of responses to operation invocations is controlled by objects. Serialisability is defined in terms of the semantics of operations. Clouds takes an integrated strategy for synchronisation and recovery which uses relationships between objects to track dependencies between transactions. The goal of the Aeolus [LW85] language is to make possible access to the synchronisation and recovery features of Clouds from a powerful programming language which provides features such as strong typing.

Clouds provides four levels of synchronisation facilities, each of which includes progressively more semantics in the determination of whether a conflict may form. At the first level, the system automatically provides an acceptable locking scheme at the object level, in which only two lock modes *read* and *write* are provided. At the second level, the programmer can specify operation compatibilities which must be true for all possible parameters and all possible object states. However, the programmer must provide appropriate recovery at this level, which must be specified for each operation. At the third level, Clouds provides programmer-controlled locks which permit any item in the domain of interest to be locked. Multiple lock modes are specified here as well. The programmer must manipulate locks explicitly and provide appropriate recovery. At the fourth level, Clouds provides an approach which can achieve greater concurrency for some applications. In this approach, each object maintains two lists to remember transactions that are ready to run and that have accessed the object but have not committed. The lists represent a running history which can be used to determine precisely whether a conflict will be created. Clouds uses pessimistic concurrency control synchronisation at all levels.

To support recovery there is a one per-machine, write-once log which is directly integrated with the virtual memory system using a write-ahead log protocol. As operations are performed on an object, the object is responsible for saving sufficient information in the log to undo the operations unless the first level is used.

Clouds supports resilience through use of stable storage. Various features are provided which cause the object support system to record sufficient information on stable storage to allow the state of an object to be recovered after a hardware failure. It is required that the state of an object must be written to stable storage whenever a transaction which modified the object commits. To specify what must be written to stable storage, Clouds allows an entire object or any data item within it to be specified recoverable. If the entire object is recoverable, then all of its contained data items are written to stable storage when a commit occurs.

The language Aeolus provides support for objects, but not inheritance. The programmer may use the support provided by the Clouds kernel for synchronisation and recovery (at the first level). In this case, the keyword *autosynch* is required in the object definition part along with the keywords *modifies* and *examines* in the relevant operation declaration.

Alternatively, using the features provided by the language and the Clouds system, the programmer may take advantage of semantic knowledge about the application to explicitly code more appropriate recoverability and synchronisation. The *lock* mechanism may be used to specify customised synchronisation rules.

10.5 Camelot/Avalon

Camelot [SBD+86] is a general purpose system which supports nested transactions executing in a distributed environment. An object-oriented programming environment is provided by the Avalon [DHW88] project which employs linguistic constructs, in the form of extensions to languages such as C++, on top of the facilities provided by Camelot to offer atomic actions for use in an application.

Camelot supports two compatible types of concurrency control: standard two-phase locking and hybrid atomicity. The concurrency control of each form is explicit with locking being provided via a call to the routine *Camlib_Lock* which takes a lock name and mode as parameters. Support for hybrid atomicity requires that objects explicitly take part in the process of transaction commit. Camelot implements this by allowing programmers to declare routines that will be called whenever they become involved in the commit or abort of a transaction.

Objects are maintained by data servers. Transaction commit information and object modification records are written to a log which is implemented using stable storage techniques.

In Avalon, support for synchronisation and recovery is provided by a number of classes: *recoverable*, *atomic* and *subatomic*, so that new classes may be devised which inherit the basic functionality. The most basic is the *recoverable* class, which offers persistence to its derived classes. The restored state of a recoverable object is guaranteed to reflect all operations performed by transactions that committed before the crash.

The *atomic* class is a subclass of *recoverable*, specialised to provide two-phase read/write locking and automatic recovery. Locking ensures serialisability, and the recovery mechanism inherited from *recoverable* ensures transaction consistency. User-defined atomic data types can be derived from *atomic*. The *atomic* class contains long-term locks: *read_lock* and *write_lock*. Long-term locks are used for inter-transaction synchronisation and will be held by transactions until they commit or abort. User-defined atomic data types should divide their operations into writers and readers. To ensure serialisability, reader operations should call *read_lock* on entry, and writer operations should call *write_lock* on entry. Note that, by taking such an approach, no short-term mutual exclusion lock on the object is necessary to do inter-operation synchronisation. The Avalon run-time system guarantees transaction consistency by performing special abort processing. Thus programmers of atomic data types need not provide explicit commit or abort operations.

The *subatomic* class is also a subclass of *recoverable*. Like *atomic*, *subatomic* allows objects of its derived classes to ensure atomicity. However, *subatomic* provides more complex primitives to give programmers more detailed control over their objects' synchronisation and recovery mechanism. Programmers can use this control to exploit type-specific properties of objects, permitting higher levels of concurrency and more efficient recovery. Besides the long-term locks, *subatomic* also provides the *seize, release* and *pause* operations for inter-operation synchronisation. Each subatomic object contains a short-term lock similar to a monitor lock or semaphore. Only one transaction can hold the short-term lock at a time. To implement transaction consistency, *subatomic* provides commit and abort operations. However, programmers are allowed to re-implement these operations. Thus, *subatomic* allows type-specific commit and abort processing.

10.6 Comparison with PC++

The substantial difference between PC++ and other systems is that PC++ takes an implicit approach for both synchronisation and transaction completion. On the contrary, the other systems (except for level 1 in Clouds) take an explicit approach for synchronisation, although some systems do take an implicit approach for transaction completion. In PC++, application information and synchronisation information are represented independently. The representation of synchronisation information, and the implementation of the synchronisation and transaction completion operations are done in the base class *Scheduler*. User-defined atomic data types can be implemented by inheriting synchronisation properties directly from *Scheduler*. Therefore, programmers only need to define the structure to represent application information, and to implement object operations that manipulate the information. The representation of application information depends only on the application requirements, without concern for how synchronisation and recovery could be done. Object operations can be implemented as in a serial environment.

Although level 1 of Clouds, like PC++, takes an implicit approach, it provides only two lock modes *read* and *write*. That is, it does not support type-specific concurrency at all, which is after all a main aim of atomic data types.

Another substantial difference is that PC++ allows users to specify a high degree of the semantics of object operations declaratively. The others either allow only limited semantics to be represented (Arjuna, TABS and the first three levels of Clouds), or can express a high degree of semantics but in an encapsulated way (Argus, Avalon and the fourth level of Clouds). In PC++, the semantics of object operations is represented through the conflict relation of an object. A special language is developed for the purpose. The language can represent the first four levels of semantics stated in Chapter 6, *i.e.*, the parameters and the results of operations can be used to specify conflict between operations.

Although all the other systems take an explicit approach, there are large differences between them depending on how the semantics is represented and used. Argus uses a *general locking protocol* for concurrency control, which permits partial and non-deterministic operations. Application information and synchronisation information are represented together as a history (log) of previously executed transactions. Information kept in the history of previously invoked transactions is used to make synchronisation decisions. Since synchronisation is embedded in the implementation of object operations, the semantics of object operations can be used directly when making synchronisation decisions. Avalon takes an approach similar to Argus for object representation and synchronisation. It is different from Argus in that it provides the facilities to support user-defined atomic data types by using type-inheritance, instead of using built-in atomic data types and special statements. Therefore, it is more flexible than Argus. Level 4 of support in Clouds takes an approach similar to Argus for synchronisation. Each object maintains a *running history* which can be used to determine precisely whether conflict can arise. The drawbacks of this kind of approach are obvious. The implementation of user-defined atomic data types becomes difficult by taking such an approach. This is because the programmer must provide synchronisation and recovery, and further, utilising the type semantics is an *ad hoc* process in such an approach. Therefore, such an implementation requires considerable sophistication on behalf of the implementor and it is very likely to include bugs. Secondly, changing the level of semantics used for synchronisation, or changing the synchronisation or recovery scheme, would require reimplementation of object operations. Thirdly, each object operation becomes more complex and needs more execution time, since before accessing an item, it is necessary to check the status of the item and to deduce the current value of the item from the object history (log). To prevent a history from increasing indefinitely, a *clear-up* process must be executed periodically. Another problem with this approach is that it is not suitable for applications which have complicated data structures.

TABS also takes an explicit approach for synchronisation, however, synchronisation information and application information are represented independently. The semantics of object operations can be specified in a declarative way in TABS, for example, through a lock compatibility matrix. It uses an extended two-phase locking protocol for synchronisation, which supports type-specific locks in order to increase concurrency. An operation must first acquire a lock in the mode defined for its class. A lock can be acquired if no concurrent activity holds a conflicting lock. Locks are released when activities complete. If an operation is unable to acquire its lock, it waits until conflicting activities complete. Recovery must be specified for each operation by programmers. Level 2 of support in Clouds takes the same approach as TABS. Level 3 in Clouds is similar, but it permits any item of an object to be locked. Implementing synchronisation is much easier in this case than in Argus, for it is not implemented in an *ad hoc* way, but according to a typical locking pattern. However, this approach provides less concurrency, for only the first two or three levels of semantics can be represented. Another disadvantage of this approach is that programmers are responsible for recovery. This is not always an easy task because in some applications operation-based recovery cannot be used, as discussed in Chapter 2.

Like TABS, Arjuna also takes an explicit approach for synchronisation using the two-phase locking protocol; however, it only classifies operations as readers or writers. Therefore, it does not provide type-specific concurrency which is a main aim of atomic data types.

Another difference between PC++ and other systems is that PC++ uses an optimistic method for synchronisation while others use pessimistic methods. Research has shown that pessimistic techniques are more robust, while optimistic techniques are cost-effective under some specialised circumstances [Her87a], and that optimistic and pessimistic methods behave differently when integrated with quorum-consensus replication [Her87b]. Pessimistic techniques trade concurrency for availability; weakening the constraints on one may tighten the constraints on the other. Optimistic techniques are different: enhancing validation to accept more interleaving has no effect on availability. In other words, optimistic methods are more suitable than pessimistic methods in some applications. PC++ provides a chance for such applications to use suitable concurrency control methods.

PC++ is also different from the other systems in providing more inter-operation concurrency. The other systems use *mutex* to provide inter-operation synchronisation which allows only one operation at a time to operate on an object. PC++ uses an optimistic method on a multiple granularity basis, which permits more concurrency. Although inter-operation concurrency is not as critical as inter-transaction concurrency, it is still important for applications which have complicated data structures such as B-trees, where execution of the operations takes a long time.

PC++ also uses a different model of persistence from the other systems. In PC++, objects reside on persistent store, even if they are bound with memory variables. Only the components which are actually accessed by a transaction are migrated into memory. Therefore, PC++ allows applications to cope with very large objects. This model also permits more concurrency during execution, since synchronising access to objects may be done at a fine granularity. Similarly, the amount of data copied to secondary storage when a transaction commits can be reduced.

Like PC++, Arjuna and Avalon also use type inheritance for implementing user-defined atomic data types. However, PC++ is very different from both of them. PC++ uses type inheritance directly to provide a synchronisation mechanism for user-defined atomic data types. On the contrary, Arjuna and Avalon use type inheritance to provide facilities with which programmers themselves may implement synchronisation mechanisms for user-defined atomic data types. PC++, Avalon and Arjuna are all good examples illustrating that type inheritance provides an effective way to construct atomic data types.

In summary, the existing systems either permit limited semantics to be represented, and thus provide less concurrency, or permit a high degree of semantics but in an encapsulated way, which makes the implementation of atomic data types difficult. On the other hand, PC++ can take advantage of a high degree of semantics when synchronising transactions, thus providing great concurrency, and at the same time allow programmers to define atomic data types in a simple and efficient way.

10.7 Summary

In this chapter, several systems which support atomic data types were described, then they were compared with PC++. PC++ is different from these systems in the following aspects. First, PC++ takes an implicit approach for synchronisation, whereas others take an explicit approach. Second, PC++ lets users to specify a high degree of semantics of object operations in a declarative way, whereas others either permit limited semantics to be represented or permit a high degree of semantics but in an encapsulated way. Third, PC++ uses an optimistic method for concurrency control, whereas others use pessimistic methods. Fourth, PC++ provides more inter-operation concurrency than others. Finally, PC++ adopts a different persistent model which allows applications to cope with very large objects, lets operations have more concurrency, and causes less data migration between memory and secondary storage.

In summary, PC++ provides a mechanism which makes the implementation of user-defined atomic data types simple, efficient but still permitting great concurrency.

Chapter 11

Conclusion

This chapter concludes the thesis by summarising the work done, and making suggestions for further work.

11.1 Summary

Existing systems that support atomic data types take explicit approaches for implementing user-defined atomic data types. They either permit limited semantics of object operations to be represented thus providing less concurrency, or permit a high level of semantics to be represented but in an encapsulated way, thus resulting in complicated implementations. In this thesis, consideration was given to an implicit approach that permits a high level of semantics to be specified in a declarative way. This makes the implementation of user-defined atomic data types as simple as in a sequential environment, thus lessening the programmer's burden but still permitting great concurrency.

To this end, we first presented a new optimistic concurrency control protocol, called the *dual-level validation* (DLV) method, formalised it and verified its correctness. The DLV method extends existing protocols in two ways: it increases internal concurrency by permitting operations on an object to be executed concurrently; it also simplifies the implementation of object operations and the recovery and commitment procedure by using suitable object representations. An implicit approach was then proposed for implementing user-defined atomic data types. Synchronisation information as well as the related operations to synchronise, recover and commit transactions by using that information are defined in a special type, the *Scheduler* class. User-defined atomic data types, by using the type-inheritance mechanism available in object-oriented languages, can then inherit these properties directly and use them to provide local atomicity for their objects. A language has been developed for specifying the semantics of object operations, by which a high level of semantics can be represented in a declarative way.

Various other related issues were also considered, namely the implementation of a transaction mechanism, distributed transaction commitment, remote object invocation and data persistence. Consideration of these requirements led to the design of PC++. PC++ supports data persistence, atomic data types and distributed transactions.

A prototype of PC++ has been implemented and has been used to implement a real application, a naming database for an active badge system. While more work needs to be done on providing more programming mechanisms such as the *set* type generator and *variable* length sequences, the application implementation shows that applications can be implemented quite easily in PC++.

11.2 Further Work

Performance Evaluation: It is clear both from the given examples and from the application implementation that user-defined atomic data types can be implemented quite easily in PC++. However, no performance measurement has yet been carried out due to shortage of time. Two aspects of performance need to be measured for PC++: the concurrency control method and the persistent model, both of which are different from existing systems. From analysis, the performance of the DLV method should be better than the optimistic method proposed by Herlihy [Her90] due to high internal concurrency and suitable object representation. However, how much performance can be gained from them needs to be measured in real applications. In contrast, the persistent model used by PC++ is designed to provide applications with high concurrency and the capability to cope with very large objects. It seems that its performance could be worse than the model which uses virtual memory because in the PC++ model access times to secondary storage are higher. Again how much performance would be lost due to providing the capabilities needs to be measured.

Pessimistic Methods: It has been shown in this thesis that by using the DLV method to provide local atomicity, an implicit approach can make the implementation of user-defined atomic data types simple but still permitting great concurrency. However, it is necessary to investigate whether it is also possible to implement an implicit approach based on a pessimistic method. It seems that an implicit approach based on a pessimistic method can be implemented reasonably easily if only the first three levels of the semantics stated in Chapter 6 are used for synchronisation. However, it is not clear how to implement an implicit approach based on a pessimistic method using a higher level of semantics.

Customised with New Methods: Many researchers have pointed out that serialisability is often a far stronger constraint than is really necessary, and that in some special application situations, temporary inconsistency would be acceptable. Research needs to be done to propose new validation methods which produce non-serialisable schedules but still, however, produce results that are appropriate to the application so that more concurrency can be achieved. We believe that the combination of type inheritance and function overloading provides a simple and flexible way to achieve incremental modification of the properties of the *Scheduler* class, so that it can be customised with new validation methods for special applications.

Nested Transactions: Nested transactions [Mos81] are useful for decomposing activities into smaller units. Nested transactions provide increased failure-tolerance: subtransactions of a transaction fail independently of each other and independently of the containing transaction. In addition, nested transactions can be used to run parts of the same activity concurrently, while ensuring that their execution is serialisable. Currently PC++, which was implemented during a short period of time, does not support nested transaction. However, the DLV method as well as the implicit approach taken by PC++ should not present any problem to the implementation of nested transactions.

State-Based Validation: The validation method taken by the dual-level synchronisation is a conflict-based validation, *i.e.*, it uses predefined conflicts between pairs of events for validation. Although it permits high concurrency by using a high level of semantics when doing validation, it will nevertheless restart certain transactions unnecessarily. For example, one *debit* operation need not be invalidated by another if the balance covers both debits. However, no conflict-based method can permit concurrent debits simply on the basis of conflicts between pairs of events. Instead, the accuracy of validation can be enhanced only by taking objects' states into account, *i.e.* by using the fifth level of semantics. Such state-based validation is more expensive than conflict-based validation, since it may amount to re-executing part of the transaction [Her90]. Nevertheless, state-based validation may be cost-effective in special cases where predefined conflicts are too restrictive, and where validation conditions can be evaluated efficiently.

Multimedia Applications: At the Computer Laboratory, Cambridge we are moving towards a world in which multimedia displays are managed by editing, browsing and composing tools [Bat93]. The MSSA, based on which PC++ provides its data persistence, has been designed for such applications. The architecture is open and adopts a layered approach. At the bottom level physical servers store byte sequence files: this level supports quality of service guarantees using sessions and tickets. At higher levels there are services to manage video and audio, together with the HLSS. PC++ has been developed with the requirements of multimedia applications in mind; the persistent class declarations can include storage service ID's, and the application programmer may therefore construct and manipulate objects which contain references to any items managed within the open architecture of MSSA, including video and voice. The optimistic concurrency control method, we believe, is more suitable for flexible real-time applications than pessimistic methods such as two-phase locking. However, research should be continued on integrating PC++ with MSSA, including support for quality of service, for use in developing multimedia authoring and presentation tools; and on evaluating within such multimedia applications the use of optimistic concurrency control mechanisms that exploit operation semantics.

11.3 Conclusions

The PC++ prototype has shown that it is possible to provide a mechanism that makes the implementation of user-defined atomic data types simple, efficient, while still permitting great concurrency. This has been achieved by taking an implicit approach for implementing atomic objects, which permits a high level of semantics of object operations to be specified in a declarative way. That is, the system defines a special class for performing synchronisation and recovery, and provides a language for specifying the semantics of object operations; users define atomic data types by inheriting these properties through type inheritance and specifying the semantics of object operations in the language. The key point here is that the method used for synchronisation and recovery should not need support from the representation of object state and the implementation of object operations, and that the method can use the semantics represented in the language to make synchronisation decisions. The DLV method proposed by this thesis meets this requirement, and hence can be used in PC++.

Experience with the active badge application is encouraging. The implementation is largely straightforward and the system is apparently robust. The application bears out the potential of the implicit approach to implementing atomic data types, in which methods can be coded as if in a sequential environment. The optimistic approach provides an ideal model for the application. There are many questions that require further investigation, but the general approach has been shown to be feasible.

Bibliography

[ABC+83] M. P. Atkinson, P. J. Bailey, K. J. Chisholm, P. W. Cockshot, and R. Morrison. An approach to persistent programming. *The Computer Journal*, 26(4):360–365, 1983.

[ACC81] M. P. Atkinson, K. J. Chisholm, and W. P. Cockshott. PS-Algol: an Algol with a persistent heap. *ACM SIGPLAN Notices*, 17(7):24–31, July 1981.

[AM83] J. E. Allchin and M. S. McKendry. Synchronization and recovery of actions. In *Proceedings of the 2nd annual ACM Symposium of Principles of Distributed Computing*, pages 31–44, August 1983.

[Bac93] J. Bacon. *Concurrent Systems: An Integrated Approach to Operating Systems, Database and Distributed Systems.* Addison–Wesley, 1993.

[Bat93] J. Bates. *Support for Real-time Interactive Presentation of Distributed Multimedia.* PhD thesis, Cambridge University Computer Laboratory, 1993. In preparation.

[BG81] P. N. Bernstein and N. Goodman. Concurrency control in distributed database systems. *ACM Computing Surveys*, 13(2):185–221, June 1981.

[BG83] P. A. Bernstein and N. Goodman. Multiversion concurrency control—theory and algorithms. *ACM Transactions on Database Systems*, 8(4):465–483, December 1983.

[BHG87] P. A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems.* Addison-Wesley, 1987.

[Bir89] A. D. Birrell. An introduction to programming with threads. Technical Report 35, DEC Systems Research Center, January 1989.

[BMTW91] J. Bacon, K. Moody, S. Thomson, and T. D. Wilson. A multi-service storage architecture. *ACM Operating Systems Review*, 25(4):47–65, October 1991.

[BN84] A. D. Birrell and B. J. Nelson. Implementing remote procedure calls. *ACM Transactions on Computer Systems*, 2(1):39–59, February 1984.

[BRGP78] P. A. Bernstein, J. B. Rothnie, N. Goodman, and C. H. Papadimitriou. The concurrency control mechanism of SDD-1: A system for distributed databases. *IEEE Transactions on Software Engineering*, SE-4(3):154–168, May 1978.

[CP84] S. Ceri and G. Pelagatti. *Distributed Databases: Principles and Systems.* McGraw-Hill, 1984.

[Dat86] C. J. Date. *An Introduction to Database Systems*, volume 2. Addison-Wesley, 4th edition, 1986.

[Dav73] C. T. Davies. Recovery semantics for a DB/DC system. In *Proceedings of the 1973 ACM National Conference*, pages 136–141, August 1973.

[Dav78] C. T. Davies. Data processing spheres of control. *IBM System Journal*, 17(2):179–198, 1978.

[DHW88] D. L. Detlefs, M. P. Herlihy, and J. M. Wing. Inheritance of synchronization and recovery properties in Avalon/C++. *IEEE Computer*, 21(12):57–69, December 1988.

[Dix88] N. Dixon. *Object Management for Persistence and Recoverability.* PhD thesis, Computing Laboratory, University of Newcastle upon Tyne, July 1988. Technical Report TR276.

[DS83] C. Dwork and M. D. Skeen. The inherent cost of nonblocking commitment. In *Proceedings of the 2nd annual Symposium on Principles of Distributed Computing*, pages 1–11, August 1983.

[EGLT76] K. P. Eswaran, J. N. Gray, R. A. Lorie, and I. L. Traiger. The notion of consistency and predicate locks in a database system. *Communications of the ACM*, 19(11):624–633, November 1976.

[EM90] J. Eliot and B. Moss. Design of the Mneme persistent object store. *ACM Transactions on Information Systems*, 8(2):103–139, April 1990.

[Gon89] L. Gong. A secure identity-based capability system. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 56–63, May 1989.

[Gra79] J. Gray. Notes on database operating systems. In R. Bayer et al., editors, *Operating Systems- an Advanced Course*, pages 391–481. Springer-Verlag, 1979.

[H84] T. Härder. Observations on optimistic concurrency control schemes. *Information Systems*, 9:111–120, June 1984.

[Her87a] M. Herlihy. Optimistic concurrency control for abstract data types. In *Proceedings of the 5th annual Symposium on Principles of Distributed Computing*, August 1987.

[Her87b] M. P. Herlihy. Availability vs. concurrency: Atomicity mechanisms for replicated data. *ACM Transactions on Computer Systems*, 4(3):249–274, August 1987.

[Her90] M. Herlihy. Apologizing versus asking permission: Optimistic concurrency control for abstract data types. *ACM Transactions on Database Systems*, 15(1):96–124, March 1990.

[HW88] M. Herlihy and W. E. Weihl. Hybrid concurrency control for abstract data types. In *Proceedings of the 7th ACM-SIGMOD-SIGACT Symposium on Principles of Database Systems*, pages 201–210, March 1988.

[Koh81] W. H. Kohler. A survey of techniques for synchronization and recovery in decentralized computer systems. *ACM Computing Surveys*, 13(2), 1981.

[Kor81] H. F. Korth. *Locking Protocols: General Lock Classes and Deadlock Freedom.* PhD thesis, Princeton University, 1981.

[KR81] H. T. Kung and J. T. Robinson. On optimistic methods for concurrency control. *ACM Transactions on Database Systems*, 6(2):213–226, June 1981.

[Lam78] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, July 1978.

[Lam81] B. Lampson. Atomic transactions. In M. Paul B. W. Lampson and H. J. Siegert, editors, *Distributed Systems: Architecture and Implementation. Lecture Notes in Computer Science 105*, pages 246–265. Springer-Verlag, 1981.

[LCJS87] B. Liskov, D. Curtis, P. Johnson, and R. Scheifler. Implementation of Argus. *ACM Operating Systems Review*, 21(5):111–122, November 1987.

[Lis88] B. Liskov. Distributed programming in Argus. *Communications of the ACM*, 31(3):300–312, March 1988.

[Low87] C. Low. A shared, persistent object store. In *Research Direction in Object-Oriented Programming*, pages 390–410, 1987.

[LS83] B. Liskov and R. Scheifler. Guardians and actions: Linguistic support for robust, distributed programs. *ACM Transactions on Programming Languages and Systems*, 5(3):381–404, July 1983.

[LW85] R. J. LeBlanc and C. T. Wilkes. Systems programming with objects and actions. In *Proceedings of the 5th International Conference on Distributed Computing Systems*, pages 132–139, May 1985.

[MA88] R. Morrison and M. P. Atkinson. Bindings in persistent programming languages. *ACM SIGPLAN Notices*, 23(4):27–34, April 1988.

[MA90] R. Morrison and M. P. Atkinson. Persistent languages and architectures. In J. Rosenberg and J. L. Keedy, editors, *Security and Persistence*, pages 9–28. Springer-Verlag, 1990.

[MBC$^+$90] R. Morrison, A. L. Brown, R. C. H. Connor, Q. I. Cutts, G. Kirby, A. Dearle, J. Rosenberg, and D. Stemple. Protection in persistent object systems. In *Security and Persistence*, pages 48–66. Springer-Verlag, 1990.

[Mey92] B. Meyer. *Eiffel: The Language*. Prentice-Hall, 1992.

[Mos81] J. E. B. Moss. Nested transactions: An approach to reliable distributed computing. Technical Report MIT/LCS/TR-260, Laboratory for Computer Science, MIT, April 1981.

[Pap79] C. H. Papadimitriou. The serializability of concurrent database updates. *Journal of the Association for Computing Machinery*, 26(4):631–653, October 1979.

[Par88] D. Parrington. *Management of Concurrency in a Reliable Object-Oriented System*. PhD thesis, Computing Laboratory, University of Newcastle upon Tyne, July 1988. Technical Report TR277.

[RC89] J. E. Richardson and M. J. Carey. Persistence in the E language. *Software-Practice and Experience*, 19(12):1115–1150, December 1989.

[Ree78] D. P. Reed. *Naming and Synchronization in a Decentralized Computer system.* PhD thesis, Dept. of Computer Science, Massachusetts Institute of Technology, 1978. Also available in Tech. Rep. MIT/LCS/TR-205.

[Ree83] D. P. Reed. Implementing atomic actions on decentralized data. *ACM Transactions on Computer Systems*, 1(1):3–23, February 1983.

[SBD$^+$85] A. Z. Spector, J. Butcher, D. S. Daniels, D. J. Duchamp, J. L. Eppinger, C. E. Fineman, A. Hessaya, and P. M. Schwarz. Support for distributed transactions in the TABS prototype. *IEEE Transactions on Software Engineering*, SE-11(6):520–530, June 1985.

[SBD$^+$86] A. Z. Spector, J. J. Bloch, D. S. Daniels, R. P. Draves, D. Duchamp, J. L. Eppinger, S. G. Menees, and D. S. Thompson. The Camelot project. *Database Engineering*, 9(4), December 1986.

[SDD$^+$85] A. Z. Spector, D. Daniels, D. Duchamp, J. L. Eppinger, and R. Pausch. Distributed transactions for reliable systems. In *Proceedings of the 10th ACM Symposium on Operating Systems*, pages 127–146, December 1985.

[SDP91] S. K. Shrivastava, G. N. Dixon, and G. D. Parrington. An overview of the Arjuna distributed programming system. *IEEE Software*, January 1991.

[STP$^+$87] A. Z. Spector, D. Thompson, R. F. Pausch, J. F. Eppinger, D. Duchamp, R. Draves, D. S Daniels, and J. J. Bloch. Camlot: A distributed transaction facility for Mach and the internet—an interim report. Technical Report CMU-CS-87-129, Carnegie Mellon University, 1987.

[Str86] B. Stroustrup. *The C++ Programming Language.* Addison-Wesley, 1986.

[Sun88a] Sun Microsystems Inc. External data representation standard protocol specification. in Network Programming Manual, 1988.

[Sun88b] Sun Microsystems Inc. Remote procedure call protocol specification. in Network Programming Manual, 1988.

[Tho90] S. E. Thomson. *A Storage Service for Structured Data.* PhD thesis, Cambridge University Computer Laboratory, November 1990.

[Ver78] J. S. M. Verhofstad. Recovery techniques for database systems. *ACM Computing Surveys*, 10(2):167–195, June 1978.

[Wei84] W. E. Weihl. *Specification and Implementation of Atomic Data Types.* PhD thesis, MIT Laboratory for Computer Science, March 1984. Tech. Rep. MIT/LCS/TR-314.

[Wei87] W. E. Weihl. Distributed version management for read-only actions. *IEEE Transactions on Software Engineering*, SE-13(1):55–64, January 1987.

[Wei89] W. E. Weihl. Local atomicity properties: Modular concurrency control for abstract data types. *ACM Transactions on Programming Languages and Systems*, 11(2):249–282, April 1989.

[Wei90] W. E. Weihl. Linguistic support for atomic data types. *ACM Transactions on Programming Languages and Systems*, 12(2):178–202, April 1990.

[Wei91] G. Weikum. Principles and realization strategies of multilevel transaction management. *ACM Transactions on Database Systems*, 16(1):132–180, March 1991.

[WH$^+$92] R. Want, A. Hopper, et al. The active badge location system. *ACM Transactions on Information Systems*, 10(1):91–102, January 1992.

[Wil92] T. D. Wilson. *Increasing Performance of Storage Services.* PhD thesis, Cambridge University Computer Laboratory, 1992.

[WL85] W. E. Weihl and B. Liskov. Implementation of resilient, atomic data types. *ACM Transactions on Programming Languages and Systems*, 7(2):244–269, April 1985.

Appendix A

Conflict Relation Syntax

We use an extended BNF grammar to define the syntax of a conflict relation specification. A production is written:

nonterminal ::= alternative | *alternative* | ... | *alternative*

The following extension is used:

nonterminal ...
nonterminal /...

The first is used to mean a list of one or more nonterminals. The second is used to mean a list of one or more nonterminals separated by '/'.

Nonterminals appear in lightface. Terminals are printed in **boldface**.

A conflict relation specification must have the syntax defined by *conf_rel*.

conf_rel ::= item ...
item ::= (opers; opers; obj)
opers ::= oper / ...
oper ::= (name, rslt)
obj ::= $=$ | $<$ | $>$ | $\leq$ | $\geq$ | $\neq$ | **any**
rslt ::= **succeed** | **failed** | **any**
name ::= string