Efficient Concurrency Control for Libraries of Typed Objects

Thomas Raeuchle

TR 86-781
(Ph.D. Thesis)
September 1986

Department of Computer Science
Cornell University
Ithaca, NY 14853
EFFICIENT CONCURRENCY CONTROL FOR LIBRARIES
OF TYPED OBJECTS

A Thesis
Presented to the Faculty of the Graduate School
of Cornell University
in Partial Fulfillment of the Requirements for the Degree of
Doctor of Philosophy

by
Thomas Raeuchle
January 1987
© Thomas Raeuchle 1986

ALL RIGHTS RESERVED
Concurrency control algorithms use a conflict detection strategy to determine operations that have to be delayed to provide a correct serialization order. To keep the cost of detecting conflicts feasible, most algorithms employ rather crude strategy that sometimes delays operations when in reality they could proceed concurrently. In an object oriented system where calls to objects can be nested this problem is exacerbated by the fact that a concurrency control decision is made at each level in the calling hierarchy.

In this dissertation we address concurrency control issues in object oriented systems. We develop a model of execution for operations on objects, drawing on similar models developed for database systems. We propose a new serialization algorithm that keeps a partial history of operations at each site, and exploits semantic knowledge of operations to achieve a finer granularity of conflict detection. Conflict detection is based on user supplied conflict predicates, thus giving the user the option of fine grained concurrency control and the responsibility for the level of cost of the conflict detection strategy. We then show how to implement the algorithm in a distributed system where sites can fail and recover independently. Finally, we explore techniques to reduce the message overhead and latency for the algorithm in a distributed system.
Biographical Sketch

Thomas Raeuchle was born on March 8, 1956 in Goeppingen, West Germany. After rather eventful high school days which will be reported on elsewhere he attended the Universitaet des Saarlandes from October 1975 until August 1980. In August 1980 he graduated with a Diploma in Informatics. In September 1980 he joined the Department of Computer Science at Cornell with a scholarship from the German Academic Exchange Service (DAAD). He received his M.S. in 1984 and expects to receive his Ph.D. in 1986.
Acknowledgements

I would like to thank my advisor, Ken Birman, for his support and encouragement. His support was essential to successfully complete this undertaking. My thanks go to my friend and advisor Sam Toueg, who, by arguing endless hours about premature timeouts, introduced me to the art of research. His optimism was a constant source of inspiration. He also introduced me to the art of growing basil and making pesto. I also thank John Gilbert and Lee Schruben for serving on my committee. Amr ElAbaddi and Thomas Joseph spent numerous hours discussing resilient objects and concurrency control - thanks guys.

Many people contributed to making my stay in Ithaca bearable by feeding me pizza, pork chops “creole”, strudel, and other treats. To all of you, thank you for helping me survive.

I am particularly grateful to my friend Edith who helped me overcome these countless hours of doubt.

Finally, I wish to acknowledge the support of the National Science Foundation under grant DCR-8412582 and that of the Defense Advanced Research Projects Agency (DoD) under ARPA order 5378, Contract MDA903-85-C-0124. The views, opinions and findings contained in this report are those of the author and should not be construed as an official Department of Defense position, policy, or decision.
Table of Contents

1. Introduction ........................................................................................................... 1

1.1. Outline of the Dissertation ................................................................................. 3

2. Atomic Objects ....................................................................................................... 5

2.1. Abstract Data Types ......................................................................................... 6

2.1.1. The Interface Specification ............................................................................. 6

2.1.2. The Type Implementation ............................................................................. 7

2.1.2.1. Object Data ................................................................................................. 8

2.1.2.2. Operations .................................................................................................. 9

2.1.3. Data Abstraction as a Design Tool ............................................................... 9

2.1.4. Nesting of Types ......................................................................................... 10

2.2. Type Libraries ................................................................................................... 11

2.3. Concurrency ....................................................................................................... 15

2.3.1. Transactions ................................................................................................. 17

2.3.2. Discussion ..................................................................................................... 18

2.3.3. Concurrency Control .................................................................................. 19

2.4. Related Work ..................................................................................................... 22

2.5. Summary ........................................................................................................... 23

3. Model ................................................................................................................... 25

3.1. The Logical System Model ............................................................................. 26
3.2. Internal Structure of Atomic objects .................................................. 26
3.2.1. Transactions ................................................................. 26
3.2.2. The Transaction Manager ................................................... 28
  3.2.2.1. The Interface Module ............................................... 29
  3.2.2.2. The Serializer Module ........................................... 30
  3.2.2.3. The Driver Module ............................................... 30
3.2.3. The Data Manager ......................................................... 31
3.3. A Model of Execution ........................................................ 32
  3.3.1. Execution Logs ......................................................... 33
  3.3.2. Correctness of Executions .......................................... 36
    3.3.2.1. Interference .................................................... 37
3.4. Summary ............................................................................. 40
4. Concurrency Control .............................................................. 42
  4.1. Interference Predicates .................................................... 43
  4.2. Execution Snapshots ....................................................... 46
    4.2.1. Precedence Dependencies ....................................... 46
    4.2.2. Execution Snapshots ............................................. 47
4.3. Algorithm 1 ................................................................. 49
  4.3.1. Part 1: Operation Invocation ...................................... 49
    4.3.1.1. Phase 1: Identification of Conflicts ..................... 50
    4.3.1.2. Phase 2: Deciding the Serialization Order ............. 51
  4.3.2. Executing Operations .............................................. 52
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>6.3. Recovery</td>
<td>94</td>
</tr>
<tr>
<td>6.4. Summary</td>
<td>94</td>
</tr>
<tr>
<td>7. Optimizing Concurrency Control</td>
<td>96</td>
</tr>
<tr>
<td>7.1. Detecting Coverage Dynamically</td>
<td>97</td>
</tr>
<tr>
<td>7.2. Determining Coverage Statically</td>
<td>100</td>
</tr>
<tr>
<td>7.3. Summary</td>
<td>104</td>
</tr>
<tr>
<td>8. Summary</td>
<td>105</td>
</tr>
<tr>
<td>8.1. Future Work</td>
<td>106</td>
</tr>
<tr>
<td>LIST OF REFERENCES</td>
<td>107</td>
</tr>
</tbody>
</table>
List of Figures

Figure 2.1 Interface Specification of Dictionary Type .............................................. 8
Figure 2.2 Type Definition for Counter Type .......................................................... 16
Figure 2.3 Interleaving of Increment Operations ..................................................... 16
Figure 3.1 Internal Structure of Atomic Objects ..................................................... 27
Figure 3.2 Create Operation, Written as a Transaction ........................................... 29
Figure 3.3 Transaction Templates ........................................................................... 35
Figure 3.4 Nested Transaction Log ......................................................................... 35
Figure 4.1 Snapshot of Create Operation ................................................................ 49
Figure 4.2 Edge Insertion Algorithm ...................................................................... 50
Figure 4.3 Edge Deletion Algorithm ...................................................................... 53
Figure 5.1 Commutativity of INC() ........................................................................ 62
Figure 5.2 Modified Edge Insertion, Part 1 ............................................................ 66
Figure 5.3 Modified Edge Insertion, Part 2 ............................................................ 67
Figure 5.4 Modified Edge Deletion ........................................................................ 68
Figure 5.5 Promote Example ................................................................................... 70
Figure 5.6 Promote ................................................................................................. 71
Figure 5.7 Commutativity Clause for File System .................................................... 72
Figure 5.8 Order of Suboperations of Create ......................................................... 74
Figure 5.9 Removing Static Precedence Dependencies ........................................... 75
CHAPTER 1

Introduction

The advent of relatively small and inexpensive yet powerful personal computers and workstations has caused a trend towards decentralizing computer systems. Computer systems are now configured as a collection of personal workstations connected by a communication network. This configuration offers several advantages over the traditional configuration of a single mainframe computer accessed through terminals. A particularly important advantage is that failures of individual system components do not necessarily render the entire system inoperational, a property frequently referred to as graceful degradation. Unfortunately at the present time very few applications utilize this capability, and provide users with fault-tolerance at the application level. This shortcoming is caused partly by lack of experienced programmers who are familiar with the complex protocols necessary to achieve fault-tolerance, but more often by the lack of development tools to support the less experienced programmer in implementing fault-tolerance mechanisms.

One approach to solving this problem is to supply the application designer with a library of predefined distributed programs that exhibit fault-tolerance properties, together with tools to integrate the library programs into applications. The applications then derive their fault-tolerance from the fault-tolerance of the library routines. The library approach has been used extensively in existing programming languages, e.g. C [30]. It is also applicable in the context of database management systems, and is particularly well suited for object oriented systems whose proliferation is due
to the advent of languages like Ada, Modula-2, Smalltalk, and Argus [17, 57, 24, 34].

This dissertation is concerned with issues that arise when libraries are used to provide fault-tolerance in a setting characterized by concurrent access to data, and the use of a concurrency control algorithm to ensure that executions are serializable. Concurrency control algorithms introduce a serious form of overhead: if a conflict is believed possible, they avoid interference between concurrent programs by forcing one to wait while the other one executes. Since these delays are costly, our task will be to minimize the number of possible conflicts by exploiting semantic knowledge of various forms, and by otherwise permitting concurrent executions under as wide a range of situations as possible. Moreover, since we expect nested routines to and between library calls to be common, we develop techniques for avoiding inefficiencies that might result if a nested invocation of a procedure were not handled in a context specific manner.

Our work was motivated by the desire for efficient object libraries in the ISIS system [8, 6, 9], which transforms fault-intolerant specifications into fault-tolerant resilient objects in a way that tends to exacerbate the problem cited above. Lacking methods such as developed here it would not be feasible to provide object libraries in the ISIS system.

This dissertation is structured as follows. First we define a model of execution for operations on objects, drawing on similar models that have been developed in the context of database systems [3, 4]. We then develop a new synchronization algorithm that is tailored towards use with object libraries. Using our model of execution, we prove the correctness of the algorithm. Finally, we apply techniques for
increasing concurrency to our algorithm.

Another concern addressed in our work is the overhead introduced by replication. In the ISIS system, objects derive their fault-tolerance from replication of functionality over several processing sites; thus, the failure of some of these sites does not influence the functionality of the object. In contrast to non-replicated systems where interaction with the synchronization mechanism is mostly confined to a single site, replicated systems require coordination among the synchronization mechanisms of several processing sites. The necessity for coordination introduces a form of communication overhead that would not arise in the non-replicated case. This overhead directly impacts the design choices made in the algorithm itself. We develop a technique to minimize the number of synchronization interactions required, reducing overhead due to communication as well as overhead due to local interaction between concurrent object interactions.

1.1. Outline of the Dissertation

This section gives a brief overview of the remaining chapters of this dissertation. In Chapter 2 we review fundamental concepts relevant for our work, namely abstract data types, object libraries, and concurrency. We then point out some of the problems that arise in the context of using object libraries to build fault-tolerant distributed systems.

In Chapter 3 we define a model of execution for operations on objects and formally define what we mean by "correct" executions.

Chapter 4 introduces a new concurrency control algorithm for operations in our model. We show that the algorithm allows only correct executions, in the sense
defined in Chapter 3.

In Chapter 5 we show the limitations of existing techniques to improve concurrency, if applied in an object oriented system. In particular, we show that the definition of commutativity adopted by these techniques is unnecessarily restrictive. We then propose an extended definition of commutativity that overcomes these limitations and show how to incorporate the extended definition into our concurrency control algorithm.

Chapter 6 contains a discussion of distributed systems. We introduce a set of communication primitives and implement our concurrency control algorithm fault-tolerantly in a distributed system.

In Chapter 7 we investigate methods to reduce the cost of concurrency control in a replicated setting. We employ data flow analysis techniques to collect information about operations on objects and use this information to reduce the number of messages and the number of local interactions necessary to achieve "correct" synchronization.

Chapter 8 concludes this dissertation and give an outlook towards future work.
CHAPTER 2

Atomic Objects

In this chapter we introduce atomic data types, a design tool for fault-tolerant distributed systems. Atomic data types are abstract data types [33, 57] with the additional property that concurrent access and system failures do not result in unpredictable behavior [56, 34]. Instances of atomic data types are called atomic objects. Like abstract data types, atomic types can be constructed by combining and extending existing types. This leads to the idea of maintaining a library of predefined types, which a user draws on to solve particular application problems.

To isolate the programmer from issues relating to concurrency and failures, operations on atomic objects are executed as transactions. Concurrent access and recovery from failures are controlled by built-in concurrency control and recovery mechanisms. The choice of mechanism is crucial to the performance of atomic objects; conventional transaction mechanisms introduce severe performance problems if used naively in a system that allows nesting of objects. The main goal of this chapter is to identify some of these problems. Solutions are proposed in later chapters.

We begin the chapter with a brief review of the concept of abstract data types. In Section 2 we discuss type libraries and issues relating to the nesting of objects. Section 3 introduces transactions and concurrent execution of operations on atomic objects. Section 4 summarizes the chapter.
2.1. Abstract Data Types

An abstract data type defines a set of **data items** and a set of **operations** on the data. Instances of a type, called **objects**, serve as data repositories. The data in an object can be accessed only through the operations defined by the object type.

Abstract data types originated from the class concept in Simula [15]. They are now part of many programming languages, including Ada, Modula-2, CLU, C++, and ARGUS [17, 57, 33, 51, 34], as well as operating systems, including MESA, EDEN, CLOUDS, TABS, ARCHONS, LOCUS, and ISIS [32, 2, 36, 50, 29, 42, 8].

An abstract type is specified by a **type definition**, consisting of two parts: The **interface specification** which pertains to the user of an object, and the **type implementation**, which describes its internal structure. For the purpose of this dissertation, we will use a variant of the C programming language [30] to describe type implementations. If necessary, we extend the language in a straightforward way to accommodate new concepts. We will now describe both parts in more detail.

2.1.1. The Interface Specification

The interface specification identifies the operations of the object type and their parameters and specifies the effect of each operation. It also contains declarations of parameters and results of operations.

For precision, we describe the effect of an operation using axiomatic semantics [26, 16] with predicates describing the state of an object before and after the execution of the operation. An operation \( op \) is described by a triple \( \{ P \} \ op \ \{ Q \} \) which is to be interpreted as follows. If \( op \) is started in a state in which \( P \) is true, then, upon
termination of \( op \), the object will be in a state in which \( Q \) is true.

**Example 2.1:** Figure 2.1 below shows the interface specification for a dictionary type. We will use this type to illustrate our ideas throughout this dissertation.

The dictionary supports operations to insert and delete an entry, a search operation to locate an entry, an update operation to update the value of a data field for a particular entry, and an operation to reorder the internal structure of the dictionary for more efficient access. We only describe the insert and search operations in detail.

The insert operation creates a new entry, associates a list of data fields with it and initializes the fields. If it is started in a state that satisfies the predicate \{true\} (any state does), the dictionary will contain an entry with key \( k \) and fields \( fn_1, \ldots, fn_n \) with values \( fval_1, \ldots, fval_n \) upon completion of the insert.

Given a key, the search operation returns a pointer to an entry with matching key. If is started in a state satisfying \( \forall \text{entry} \in \text{dict}, \text{entry.key} \neq k \) the pointer variable \( ptr \) will contain \( NIL \) upon completion. Otherwise, if started in a state where the dictionary contains an entry with key \( k \), \( ptr \) will contain a pointer to this entry upon termination of the search operation. □

**2.1.2. The Type Implementation**

The type implementation contains declarations of the data, and code implementing the operations specified in the interface.
\textbf{type} dictionary = {

\{true\}

\{exists entry \in dict, entry.key = k \land \forall i \in 1, \ldots, n, entry.fn_i = fval_i\}

\{true\}

\textbf{delete}(\textbf{string} \; k); \\
\{\forall \; entry \in \; \textbf{dict}, \; \textbf{entry.key} \neq k\}

\{exists entry \in dict, entry.key = k\} \quad \{\forall \; entry \in \; \textbf{dict}, \; \textbf{entry.key} \neq k\}

\textbf{ptr} = \textbf{search}(\textbf{string} \; k); \\
{\textbf{ptr} = \textbf{address}(\textbf{entry})}

\{\textbf{ptr} = \textbf{NIL}\}

{\textbf{ptr} = \textbf{address}(\textbf{entry})}

\textbf{update}(\textbf{entry}^*\textbf{ptr}, \textbf{string} \; fn, \textbf{int} \; fval); \\
\{\textbf{entry.fn} = \textbf{fval}\}

\{P\}

\textbf{balance}()

\{P\}

\}

\textbf{Figure 2.1.} Interface specification of dictionary type

\subsection{Object Data}

The data declaration specifies the name and type of each data item. Names of items are not visible outside the implementation body, while their type might be revealed in the interface specification. In the above example, the type \textit{item} is assumed to be known in the interface, but names of individual entries are not
known by the user of the abstract data type.

Object data satisfy user imposed consistency constraints. For example, a constraint might require that the file names in a file system are unique. Consistency constraints are invariants in the specification. In the above example, the invariant is that for every sequence of operations \( H = o_1, \ldots, o_n \) that has been performed, the dictionary contains all items that have been inserted but not deleted.

2.1.2.2. Operations

The code that implements operations consists of control statements governing the flow of control and data statements to access data. Control statements include if, while, sequential composition, and block structure. The cobegin \( \cdots \) coend statement expresses nondeterministic ordering of statements. Its semantics are as follows: Execution of cobegin \( S_1 ; \cdots ; S_n \) coend begins by starting any one of \( S_1, \ldots, S_n \). Then, the rest of the \( S_i \) are executed in any order. The cobegin statement terminates once all \( S_i \) have been executed.

Data statements include read and write statements to access object data, and calls to other objects. The latter will be discussed in more detail in Section 3.2.2. A detailed example that outlines the implementation of a filesystem type is given in the next section.

2.1.3. Data Abstraction as a Design Tool

The use of data abstraction in system design offers several benefits. Firstly, the type specification gives a precise definition of the correct behavior of an object. A number of proof methods exist [46,26] to aid the type designer in verifying that
the operations preserve the type invariant. The same methods can be used to verify that the implementation of an operation meets its specification.

Secondly, systems designed using abstract data types exhibit high flexibility. Since a user relies only on the specified effects of an operation the type implementor is free to change the implementation of the operation.

Thirdly, the separation of the specification from the implementation is an important factor for data security. The fact that a user can manipulate data only through operations provided by the type designer precludes uncontrolled manipulation of the data. It also gives the type designer the freedom to incorporate protection mechanisms into the operations, by for example, requiring authentication arguments for certain critical operations.

2.1.4. Nesting of Types

Like abstract data types, atomic types can be nested: The definition of a type can contain references to other previously defined types. This leads to nested objects and thus to nesting of invocations, i.e. an operation on an object A can result in calls to other objects referenced in the implementation of A.

The use of nested types allows for hierarchical decomposition of systems into smaller subsystems and makes standard software engineering methods like bottom-up and top-down design applicable to the development of fault-tolerant distributed systems. A basic premise that must be satisfied by any implementation is that nesting must be efficient if object oriented systems are to be useful.
2.2. Type Libraries

A type library consists of a set of abstract data types. It contains both the interface specification and the type implementation for each of the types. Only the interface specifications are visible to a user, while the implementations are not directly accessible and thus cannot be tampered with. A type designer can then consult the library for predefined types that would aid in achieving the desired functionality of the new type.

In the following example we take advantage of nesting of types by using the dictionary type defined above as a directory in the implementation of a filesystem type. For brevity, we omit the semantic specifications in the type interface.

Example 2.2: The filesystem type supports create and delete operations to create and remove files, and read, write and seek operations to access data. Data space for files is allocated contiguously from a character array disk. The deletion of a file which is stored between other files might create holes in the disk array. To be able to reuse this space, holes in the data space are collected in the array free. For simplicity, we assume that all arrays are infinite. In particular we assume that the filesystem does not run out of space.

```plaintext
type file_system = {
    create(string fn, Int size);
    delete(string fn);
    read(string fn, char buffer, int n);
    write(string fn, char *buffer, int n);
}
```
seek(string fn, int offset);
}

(type_body file_system = {
    char disk[];
    int free[];

    export create, delete, read, write, seek;
    import dict.insert, dict.delete, dict.search, dict.update, dict.balance;

    /* initialize list of holes */
    init() {
        free.loc = -1;
        free[0].loc = 0;
        free[0].size = inf;
    }

    /* allocate n bytes of disk space */
    allocate(int n) {
        i = 0;
        while(free[i].size < n)
            i = i + 1;
        new = free.loc;
        free[i].size = free[i].size - n;
        free[i].loc = free[i].loc + n;
        return(new);
    }

    /* deallocate n bytes of disk space, starting at loc */
    deallocate(int loc, n) {
        i = 0;
        while(free[i].loc < -1)
            i = i + 1;
        free[i].size = n;
        free[i].loc = loc;
    }

    /* create a file with name fn, allocate n contiguous bytes */
    create(string fn, int n); {
        cobegin
        ptr = allocate(n);
        //
        loc1 = dict.search(fn)
        coend;
        if( loc1 != NIL) {
            deallocate(ptr, n);
            return(ERROR)


```c
else {
    dict.insert(fn, "loc", ptr, "size", n, "offset", 0);
    dict.balance();
    return(OK)
}
```

/* remove all files with name fn */
```c
delete(string fn); {
    file = dict.search(fn);
    if((file != NIL) && (file->loc != NIL))
        dealloc(file->loc, file->size);
    else
        return(ERROR)
}
```

/* read n bytes from file fn into buffer */
```c
read(string fn, char buffer, int n); {
    file = dict.search(fn);
    if((file == NIL) || (file->offset + n > file->size))
        return(ERROR)
    else {
        copy(file->loc + file->offset, buffer, n);
        return(OK);
    }
}
```

/* copy n bytes from buffer to file fn */
```c
write(string fn, char *buffer, int n); {
    file = dict.search(fn);
    if((file == NIL) || (file->offset + n > file->size))
        return(ERROR)
    else {
        copy(buffer, file->loc + file->offset, n);
        if(file->offset + n > file->size)
            dict.update(fn, file->offset + n);
        return(OK);
    }
}
```
The library approach suggests that abstract types should be designed in a very
general way so as to make them usable in a variety of applications. However, gen-
erality may have its price. For example, the dictionary allows an arbitrary number of
data fields per entry. For our file system application, however, it suffices to have
only one field per key, and the implementation of such a special dictionary type
could be more efficient than that of the general type. On the other hand, generality
saves the effort of reimplementing minor variations of the same type. For exactly
this reason a library would contain the general implementation.

We believe that the benefits of abstract data types far outweigh their disadvan-
tages. Efficiency is measured not only in terms of runtime, but also in terms of sys-
tems development and maintenance cost. In this view of efficiency, a carefully
designed general type which is provably correct will, in the long run, prove to be
more cost effective than a collection of "fast" special purpose implementations. This
belief motivates the work in this dissertation: Our goal is to develop techniques to
overcome some of the more glaring inefficiencies arising from nesting of types in
order to encourage the use of data abstraction and type libraries in the design of fault-tolerant distributed systems.

2.3. Concurrency

If atomic objects support concurrent invocation, multiple callers could access an object simultaneously, resulting in considerable performance improvements. On a single processor, concurrent execution refers to the interleaving of steps of different operations which might eliminate unnecessary waiting periods and decrease latency. In a multiprocessor system concurrency refers to simultaneous execution of several operations, which leads to increased system throughput. Uncontrolled concurrency, however, can lead to unpredictable behavior, as the following example illustrates.

**Example 2.3:** Consider an object of type *counter* that supports an increment operation `inc()`, that increments the value of the counter by 1. Figure 2.2 shows the definition of the counter type. The `inc()` operation is implemented by first reading the value of the variable `c_value` into a temporary variable `temp` which is then incremented. Then the new value is written back into `c_value`.

Figure 2.3 shows an interleaving of two increment operations in which the effect of one of the increments is lost. □

While both operations in the above example satisfy their specification if executed in isolation - each of them leaves the data in a state satisfying \{counter = C + 1\} when it is started in state \{counter = C\} - their combined effect is not what one would expect from executing two increment operations. Such unpredictable outcome is caused by the fact that during the execution of an
type counter = {
    {count = C}
    inc()
    {count = C + 1}
}

type_body counter = {
    ...
    int c_value;
    inc() {
        int temp;
        temp ← read(c_value);
        temp ← temp + 1;
        write(c_value, temp);
        {
            ...
        }
    }
}

Figure 2.2. Type Definition for Counter Type

inc_1()                        inc_2()
1.1: temp ← read(counter);
1.2: temp ← temp + 1;
    2.1: temp ← read(counter);
    2.2: temp ← temp + 1;
1.3: write(counter, temp);
    2.3: write(counter, temp);

Figure 2.3. Interleaving of increment operations

operation the type invariant might be temporarily violated. To ensure predictable outcome, operations on atomic objects are required to execute as transactions.
2.3.1. Transactions

A transaction is a sequence of steps, delimited by BEGIN and END markers. The markers are not necessarily explicit: in the case of atomic data types they are inserted automatically by the compiler at the beginning and end of an external operation. The steps are partially ordered, i.e., their execution order is not fully determined in advance. Steps not related in the partial order can be executed concurrently; however, undesired interleavings of these steps have to be prevented as well. When an object invokes an operation on another object the concept of nested transactions as introduced by Moss [37] is immediate.

In the nested transaction model, a transaction can contain subtransactions, which in turn can contain subtransactions of their own. Thus a whole hierarchy of transactions is possible. The top of the hierarchy - a transaction which has no enclosing transaction - is called top-level or root transaction. Standard tree terminology is used to refer to the relations among transactions in the hierarchy. Thus, internal nodes correspond to operations pending within a transaction, and leaf nodes to indivisible operations on physical data items.

Subtransactions are formed by enclosing a sequence of steps into BEGIN ... END markers. The cobegin ... coend statement introduced above can be used to express concurrency within a transaction by writing concurrent steps as branches of the statement. A branch can be enclosed in a BEGIN ... END bracket, forcing it to execute transactionally. It will become clear later in this chapter that this use is consistent with the semantics of the cobegin statement.
There are two possible outcomes of a transaction. If the transaction is successful, it commits and its effects become permanent. Otherwise it aborts and leaves no effect on the data. Transactions might not succeed for various reasons: for example, a disk containing the file to be written might not be accessible, or, data might be in a state that causes an arithmetic error in an operation, or, a failure in form of a system crash might occur. A discussion of different types of failures appears in Chapter 6.

While in the transactional model of execution operations might still temporarily violate the consistency constraints, i.e. type invariants, it is guaranteed that no other transaction can observe such an inconsistent state. This is expressed by the concepts of atomicity and serializability of transactions.

An operation is atomic if it is executed completely or not at all, and no partial effects of the operation on the data are visible to other transactions.

The concept of serializability relies on the assumption that operations preserve consistency constraints if executed in isolation. Consequently, serial execution of a set of operations preserves data consistency. Concurrent execution of a set of operations is serializable if the effect is the same as if the operations were executed in some serial order. Serializable execution in atomic objects is enforced by a built-in concurrency control mechanism.

2.3.2. Discussion

We have described a data abstraction approach suited to the maintenance of object libraries. In a distributed setting characterized by concurrency, nested transactional execution is introduced as a natural means for ensuring correctness of
operations on abstract data types.

Transactions are a powerful abstraction, isolating the programmer from issues relating to concurrency and failures, thus making the development of fault-tolerant distributed programs similar to programming traditional sequential systems.

Nested transactions offer two main advantages over single-level transactions. Firstly, they allow concurrency within transactions. This aspect becomes particularly important in the context of atomic objects, where the execution of an operation on one object might include several calls to other objects. If executed sequentially, the communication delays incurred by each of the calls add up and overall system performance might become unacceptable. Here concurrent execution can lead to significant improvement in system performance.

Secondly, nested transactions allow for smaller units of atomicity. Consequently, the effects of failures can be confined to a smaller scope, thus reducing the cost of recovery from failures. This becomes important in systems that cannot tolerate long delays for recovery.

2.3.3. Concurrency Control

Techniques to enforce atomic execution of operations can be broadly classified as pessimistic or optimistic. In a pessimistic approach, transactions are prevented from observing inconsistent states by making data available only when the transaction updating the data is committed. Concurrency is thus limited to operations accessing disjoint sets of data items. This approach thus adopts the pessimistic assumption that conflicts are fairly frequent and acts to prevent possible inconsistencies. The optimistic alternative is to assume that conflicts are unlikely and to "cut
corners" in the hope that no conflict will occur, and rolling back (aborting) transac-
tions that violate this assumptions.

For example, a more optimistic approach to return results of an operation is to
make them available as soon as they are computed. Other transactions can then
use these data immediately, relying on the assumption that the data will be commit-
ted later. If aborts are rare, this technique can lead to a substantial increase in con-
currency. If, however, a transaction aborts, all transactions which have observed
effects of the aborted transaction have to be aborted also. As a result, a whole
chain of transactions might have to be aborted; this effect is known as cascading
aborts. (This phenomenon was observed as early as 1946 when Borges described
that undoing the past might sometimes necessitate murder [10]. ) The extend to
which the optimistic approach is feasible depends on the probability of such an
abort and its expected cost. In most systems cascading aborts are costly to handle.
An extensive bookkeeping scheme is required to keep track of which transaction
have observed uncommitted data, and in a system where aborts are expected to
happen frequently, increased concurrency might not compensate for the additional
overhead. Thus, if the probability of transaction aborts cannot be kept close to zero,
the optimistic approach is unjustified.

In a later chapter we will describe a technique that falls in between these two
extremes. It takes advantage of the fact that in a nested transaction system some
of the bookkeeping is provided by the nesting mechanism. This technique is
designed to perform well in a system where aborts are expected to be infrequent; if
aborts occur anyway, the scope of the cascade is limited to a single nested transac-
tion.
While the nested transaction model facilitates the use of the sort of optimistic concurrency control methods described above, it nonetheless represents a source of inefficiency itself. In the nested transaction model, concurrency not only arises between top level transactions, but also between subtransactions of the same parent. Concurrency between those must be controlled, and in general a concurrency control mechanism must be invoked at every level of the transaction hierarchy. In an atomic object system some of these invocations can be avoided by taking advantage of the fact that set of possible transactions can be derived from the object definition. From this set of transactions we can determine points where concurrent execution might arise and limit invocations of the concurrency control mechanism to these points. In Chapter 7 we will use data flow analysis techniques to determine the set of transactions and then develop a method to limit concurrency control to a set of precomputed synchronization points. In addition, we introduce the notion of scope of concurrency that can be used to restrict concurrency control to the local site when a conventional algorithm might require a distributed synchronization action.

Concurrency can be further increased by incorporating information about the semantics of operations into the concurrency control method. This information must be supplied by the type designer. In Chapter 5 we enhance our concurrency control method to use information about commutativity of operations in order to increase concurrency in atomic objects.
2.4. Related Work

Concurrency control has been the study of extensive research, e.g. [20, 40, 12, 44]. In [4] Bernstein and Goodman present a survey of numerous methods. Weihl [56] and Herlihy [27] extended the concept of serializability introduced by Papadimitriou [40] to operations on abstract data types. Our algorithm is based on a similar extension.

Jefferson [28] proposed a timestamp based concurrency control method for distributed simulation. It is similar to Reed's [44] timestamping schema in that it forces transactions to commit in timestamp order. It achieves greater efficiency than [44] by providing limited rollback of transactions in case of aborts. It provides deadlock-free scheduling, which is of particular importance in simulation applications. Two-phase locking on the other hand is prone to deadlocks and thus not readily applicable in this domain. The drawback of timestamp-based algorithms is that they can lead to performance degradation in the presence of long transactions. This drawback however is hardly a problem in the domain of distributed simulation where transactions are typically short operations manipulating event queues.

In our domain transactions are general operations on data and the drawback of timestamp-based concurrency control methods poses a potential performance problem, and we adopt an approach that is a hybrid between timestamping and locking. It avoids deadlock by explicitly preventing circular transaction dependencies, and it avoids the performance drawbacks of timestamp algorithms by scheduling suboperations as they arise, thus preventing suboperations of long transactions from blocking other transactions. Transaction latency is further reduced by allowing com-
mutative suboperations of two transactions to execute in any order. While our algorithm is applicable to distributed simulation, its generality might impose additional cost that can be avoided by using an algorithm geared towards a particular application domain, e.g. [28].

Atomic data types as a design tool for systems with concurrent access to data are supported by several languages and operating systems, e.g. ARGUS, ISIS, CLOUDS, TABS, ZEUS [34,8,36,50,11]. Most of these systems support standard concurrency control methods, e.g. two-phase locking [20]. Schwarz and Spector [49] propose to incorporate semantic knowledge of operations into concurrency control decisions. Our work can be viewed as an extension of their method to include nested transactions and arbitrary operations apart from reads and writes. Alchin [1] proposes four levels of concurrency control in the CLOUDS system; except for the default two-phase locking strategy, all levels are under the control of the user, and the object designer is responsible for implementing a correct concurrency control method. The approach taken in this dissertation relieves the object designer of this responsibility and gives him/her a uniform method to incorporate operation semantics into concurrency control decisions by means of conflict predicates.

2.5. Summary

In this chapter we introduced the concept of atomic data types as a development tool for fault-tolerant distributed systems. Atomic data types extend the concept of data abstraction and allow abstraction from concurrency issues and failures. Built-in concurrency control and recovery mechanisms guarantee that operations on atomic objects are executed as transactions.
Efficiency considerations limit the use of standard concurrency control and recovery methods in the implementation of atomic objects. The problems introduced by these methods can be summarized as follows:

(i) Pessimistic concurrency control methods limit system performance by restricting internal concurrency to operations that access disjoint sets of data.

(ii) Optimistic concurrency control methods necessitate costly bookkeeping mechanisms to handle cascading aborts.

(iii) A general purpose nested transaction mechanism might invoke synchronization actions which are redundant in the context in which a particular transaction executes, or which are distributed although a non-distributed action would suffice.

It is the goal of this dissertation to show how these inefficiencies can be overcome and to encourage the use of atomic data types in the design of fault-tolerant distributed systems.
CHAPTER 3

Model

In this chapter we develop a formal model of execution for operations on atomic objects. The goal is to provide a framework within which we can formalize properties of transactions, i.e. serializability and atomicity, and prove the correctness of algorithms enforcing these properties. The model presented is closely related to the internal structure of atomic objects; hence we will first give an overview of the internal mechanisms used in atomic objects and then proceed to describe the details of the model. The model itself is derived from similar models proposed in [4, 3, 49, 27, 56, 52, 5].

Atomic objects constitute an interface between logical operations issued by a user and physical operations on data. We present the model in terms of a logical system that ignores many of the physical details related to the execution of an operation. In a later chapter we then show how to implement this abstraction in a distributed computer system using protocols which guarantee that the system exhibits the desired logical behavior, even in the presence of failures, recoveries, and replication of data.

The chapter is divided into 4 sections. In Section 1, we describe the environment in which atomic objects are implemented. In Section 2 we give an overview of the implementation. In Section 3 we develop a formal model of execution, and finally summarize the chapter in Section 4.
3.1. The Logical System Model

The system consists of a collection of processes, each having a local state and its own address space. Processes communicate solely through messages. The system supports remote procedure calls (RPC) \[38\] with value return parameters; the RPC mechanism is used by one object to invoke operations in another object.

The communication system is reliable; messages are neither lost nor garbled, and they are delivered in the order in which they were sent (for messages sent concurrently the order of delivery is unspecified).

3.2. Internal Structure of Atomic objects

Following the structure proposed by Bernstein and Goodman \[4\], we view an atomic object as composed of three logical entities (see Figure 3.1): transactions, the transaction manager, and the data manager. Transactions are the means by which users communicate with objects. The object itself is composed of two processes. The transaction manager process (TM) supervises the interaction between users and objects and shares responsibility for the correct execution of operations. The data manager process (DM) accepts and executes requests for data access from the TM. It also provides a mechanism that supports the transaction manager in implementing transaction atomicity. We now describe each of these entities in detail.

3.2.1. Transactions

To isolate the object designer from failure and concurrency issues, operations on atomic objects are executed transactionally. As defined previously, a transaction
is a sequence of steps, enclosed within BEGIN ... END brackets. The steps may be
atomic read and write steps to access data, calls to operations in other objects, or
subtransactions. A read step takes as argument the name of a data item and
returns the current value of that item. Write steps take two arguments, the name of
an item and the value to be written into that item. Both read and write steps return
an error indication if the operation was not successful. A call to some other object
is of the form Obname.Op(a₁, ⋅⋅⋅, aₖ), where Obname is the name of the object,
Op is the name of the operation to be called, and a₁, ⋅⋅⋅, aₖ are its arguments. A
subtransaction within a transaction is indicated by enclosing a sequence of steps
and/or calls in a BEGIN ... END bracket and, optionally, labeling the sequence with
a name.

Throughout this dissertation we will use the following notational conventions for
transactions. $T = \{t₁, ⋅⋅⋅, tₙ\}$ denotes a transaction $T$ which is implemented using
object name. This enforces unique systemwide TIDs, since the addressing mechanism guarantees unique object names. TIDs for subtransactions are formed by appending the subtransaction index to the parent's TID. TIDs for non top-level transactions are formed by appending the value of t_count to the TID of the call. Then t_count is incremented and the operation together with its arguments and the TIDs is passed to the serializer.

3.2.2.2. The Serializer Module

The serializer decides on the serialization order for new transactions, based on the state of the object and a record of currently active transactions which it maintains in an execution snapshot. Snapshots represent the active parts of an execution log (defined in the next section); they are discussed in Chapter 4, together with concurrency control algorithms that are used to make serialization decisions.

3.2.2.3. The Driver Module

The driver oversees the physical execution of operations. Once a serialization order for an operation has been determined, the driver translates the logical operation into its physical equivalent and supervises execution of the different steps. Logical read and write requests are translated into physical requests which are then passed on to the data manager. If data are replicated, this translation might result in multiple physical writes, one at each copy, for a single logical write. A call step results in a remote procedure call to the addressed object with the TID of the calling transaction inserted into the call. The results returned by read steps and calls are stored in appropriate temporary variables.
After all steps of the transaction have terminated, or a failure has occurred, the driver makes a commit or abort decision for the transaction as a whole. It then informs the DM of the outcome.

3.2.3. The Data Manager

The data manager serves as front end to the physical data storage. It processes read, write, commit, and abort requests issued by the TM and provides a version store mechanism to support transaction atomicity.

The version store is similar to mechanisms proposed by Moss [37] and the mechanisms used in ISIS and ARGUS [9,34]. The first time a transaction accesses a data item, a new copy of the item is created and labeled with the transaction’s TID. While it is executing, the transaction operates on this version of the data. Upon commitment, the values of its version are copied to permanent storage in one atomic step. If the transaction aborts, its versions are discarded and no effect is visible on the permanent data.

To enforce atomicity of nested transactions a collection of versions is required, one for each level of nesting. For reasons of uniformity we will refer to permanent copies of data as versions as well, and also assume that they are labeled with the empty TID ε. In particular, we will use the fact that ε is a prefix of any TID. Note that the version schema described below serves as a model of execution only. Any implementation will contain various optimizations to reduce the amount of data copied and to speed up commitment of operations. We now describe in detail the processing of each of the DM’s operations.
suboperations \( t_1, \ldots, t_n \). \( \text{Sub}(T_i) \subseteq \{t_1, \ldots, t_n\} \) denotes the set of suboperations which are to be executed as subtransactions, and \( \text{Steps}(T_i) \subseteq \{t_1, \ldots, t_n\} \) denotes the set of primitive atomic suboperations which are not separate subtransactions. We sometimes write \( \text{Steps}_R(T) \) for the set of all suboperations that only read data, and \( \text{Steps}_W(T) \) for suboperations that update data items.

**Example 3.1:** Figure 3.2 shows the create operation of the file system with 3 user defined subtransactions, two of which are named. The alloc and search transactions can be executed concurrently, as indicated by the `cobegin` statement. The third subtransaction is not named and is executed after the search and alloc transactions are completed. The transaction contains several calls to a dictionary object named `dict`. In this case, \( \text{Sub}(\text{create}) \) includes the alloc, search and the "nameless" subtransaction, and \( \text{Steps}(\text{create}) \) includes `deallocate()`. □

Each transaction carries a unique transaction id (TID), which is assigned by the TM. The steps in a transaction are marked with the transaction's TID. The subtransactions of each parent transaction have a unique index according to their position in the code, and the TID for subtransactions is formed by appending their index to the TID of the parent transaction.

**Example 3.2:** In the above example, the create operation is assigned TID \( t \), the alloc and search subtransactions have TIDs \( t.1 \) and \( t.2 \), and the last subtransaction has TID \( t.3 \). □

### 3.2.2. The Transaction Manager

The transaction manager constitutes the intermediary between a user and the object. It consists of three modules: The interface processes requests for
/* create( string fn, int n); */

/* TID = t */
BEGIN

cobegin
    /* TID = t.1 */
    alloc: BEGIN ptr = allocate(n); END

    /* TID = t.2 */
    search: BEGIN loc1 = dict.search(fn); END
coend;
if( loc1 != NIL) {
    deallocate(ptr, n);
    return(ERROR)
}
else {
    /* TID = t.3 */
    BEGIN
        dict.insert(fn, "loc", ptr, "size", n, "offset", 0);
        dict.balance();
        return(OK)
    END
}
END

Figure 3.2. Create operation, written as a transaction

operations and assigns transaction ids to transactions. The serializer performs concurrency control and the driver supervises the dispatch of operations to the underlying DM.

3.2.2.1. The Interface Module

Once a logical operation is invoked, the interface module generates TIDs for the operation and its subtransactions. For this purpose, it maintains a transaction counter $t_{count}$ to assign a unique number to each transaction on the object. For a top-level transaction, the TID is generated by appending the value of $t_{count}$ to the
read(x) (write(x, val))

Let $t$ be the TID of the read (write) request. If there exists a version of $x$ with TID $t$, read (write) that version of $x$. Otherwise, create a new version of $x$, labeled $t$. Initialize it by copying the value of the version labeled $t'$, such that $t'$ is a prefix of $t$ and no other version of $x$ is labeled with a longer prefix of $t$.

Then read (write) that version.

Commit($t_k$)

Delete $t_k$ from all labels containing $t_k$. Install all versions with label $\epsilon$ as permanent.

Abort($t_k$)

Discard all versions with labels containing $t_k$.

The read/write rule guarantees that each transaction operates on its own version. The commit rule implements conditional commits, i.e., the changes made by a subtransaction are not made permanent until all its ancestors are committed. Shortening the label corresponds to passing the version to the parent transaction until it reaches the top-level transaction. Only when this transaction commits are the changes installed. The abort rule guarantees that no effects of aborted transactions are visible.

3.3. A Model of Execution

We now define a formal model of execution and discuss criteria for determining the correctness of a particular execution. We first introduce execution logs and then discuss conditions under which logs are to be considered equivalent. Based on log equivalence we define $\delta$-serializability as correctness criterion for executions.
3.3.1. Execution Logs

Execution of a set of transactions is modeled using an execution log \([4, 40]\). Informally, an execution log is a sequence of triples (transaction, step, argument) describing the order in which the steps of transactions are executed. Formally, consider a transaction \(T = \{t_1, \cdots, t_n\}\) with \(T\) partially ordered by the relation \(\prec_T\). Two suboperations \(t_i, t_j \in T\) are related \(t_i \prec_T t_j\) if \(t_i\) starts execution only after \(t_j\) has been completed. Intuitively, the order \(\prec_T\) represents the flow of control within a transaction \(T\).

**Definition 3.1:** Let \(L = (l_1, \cdots, l_N)\) where each \(l_i = (T, s, A_k)\) consists of a transaction name \(T\), a suboperation \(s \in T\), and the arguments \(A_k\) of \(s\), and let \(L\) be totally ordered by \(\prec_L\) with \(l_i \prec_L l_{i+1}\) for \(i = 1, \cdots, N - 1\).

\(L\) is an execution log, if \(\prec_L\) is consistent with the partial orders \(\prec_T\) for all transactions \(T\) appearing in \(L\). \(\square\)

For convenience we sometimes write the transaction name as additional index of the step, as in \((t_{1,2}, a)\) for \((T_1, t_2, a)\). In the example below the arguments are omitted.

**Example 3.3:** Let \(T_1 = \{t_{1,1}, t_{1,2}, t_{1,3}\}\) with \(\prec_{T_1} = \{(t_{1,1}, t_{1,3}), (t_{1,2}, t_{1,3})\}\), and let \(T_2 = \{t_{2,1}, t_{2,2}\}\) with \(\prec_{T_2} = \{(t_{2,1}, t_{2,2})\}\). Then \(L_1 = (t_{1,2}, t_{2,1}, t_{1,1}, t_{2,2}, t_{1,3})\) is an execution log for \(\{T_1, T_2\}\). \(L_2 = (t_{1,2}, t_{1,3}, t_{2,1}, t_{1,1}, t_{2,2})\) is not an execution log, since \(t_{1,3} \prec_{L_2} t_{1,1}\), but \(t_{1,1} \prec_{T_1} t_{1,3}\) and thus \(\prec_{L_2}\) and \(\prec_{T_1}\) are inconsistent. \(\square\)

In addition to indicating the order of execution, execution logs for nested transactions also reflect the nesting structure of operations. Nested operations are
represented by transaction templates.

Definition 3.2: A transaction template TP for a nested operation T is a labeled tree with a partial order on its nodes. Each node is labeled with a pair (name, arguments) as follows:

1. The root is labeled (T, A), where A stands for the arguments of T.
2. The children of a node t are labeled with (t_j, a_j) where t_j \in Steps(t) \cup Sub(t) is the name of a suboperation of t and a_j are the arguments of t_j.

The children of t are partially ordered by the relation <_t indicating the flow of control within t. The partial order <_{TP} of TP is the union of the partial orders <_t for all t in TP. □

In the template, an order <_t induces an order downward in the hierarchy, i.e., the order t_1 < t_2 for nodes t_1 and t_2 induces an order u < v between the children u of t_1 and v of t_2. Intuitively the induced order expresses the fact that executing t_1 before t_2 means executing all suboperations of t_1 before those of t_2. To avoid cluttering up the examples we usually omit these induced orderings.

Example 3.4: Let T_1 = \{t_{1,1}, t_{1,2}, t_{1,3}\} with t_{1,1} = \{s_1, s_2\} and t_{1,3} = \{u_1, u_2\}, and let T_2 = \{t_{2,1}, t_{2,2}\} with t_{2,1} = \{v_1, v_2, v_3\} and t_{2,2} = \{z_1, z_2\}. Let the orderings of the children be as follows:

\[
\begin{align*}
<_t &= \{(s_1, s_2)\} \\
<_t &= \{t_{1,1}, t_{1,2}, t_{1,3}\} \\
<_t &= <_t \cup <_{t_{1,1}} \\
<T_{P_1} &= <_t \cup <_{t_{1,1}} \\
<T_{P_1} &= <_t \cup <_{t_{1,1}} \\
<_t &= \{(v_1, v_2), (v_2, v_3)\} \\
<_t &= \{z_1, z_2\} \\
<_t &= \emptyset \\
<T_{P_2} &= <_{t_{2,1}} \cup <_{t_{2,2}}
\end{align*}
\]
$L$ is a nested execution log for \{\(T_1, \cdots, T_n\)\}, if \(<_L \) is consistent with \(<_{TP} \) for all \(i\). □

**Example 3.5:** The execution log in Figure 3.3 is a valid execution log for \(T_1\) and \(T_2\). Figure 3.4 is not a valid execution log, since \(u_1 <_L s_2\) is inconsistent with the order \(s_2 <_{T_1} u_1\) induced by \(t_{1,1} <_{T_1} t_{1,3}\). Switching the order of \(s_2\) and \(u_1\) makes Fig. 3.4 a valid log. □

### 3.3.2. Correctness of Executions

In this section we use execution logs to determine whether a particular execution is correct. The framework is based on the premise that execution of a transaction in isolation with no concurrent interleavings is correct, i.e. it leaves all objects in a consistent state, and all results returned are correct. Our framework is similar to the models used by Beeri [3] and Lynch [35]. The state of an object is defined by the values of all data items that can be observed using the operations of the object. It does not include values of items that cannot be observed, e.g., for an object of type set in general only the content of the set can be observed, but not the internal data structure that is used to implement the set abstraction. The final state of an object as well as the results of transactions can be deduced from a valid execution log, given the initial state of the object.

Under the assumption that non-interleaved or serial execution is correct, and under some suitable definition of equivalence, executions are then considered correct if they are equivalent to a serial execution. Intuitively, executions are equivalent if every transaction produces the same result in both executions and the result of all future transactions will be the same, regardless which execution actually
happened. While this general type of equivalence is the least restrictive and thus allows the most amount of concurrent interleavings, it has been shown that determining equivalent executions is NP-complete under this definition [39].

This complexity problem motivates the definition of less general forms of equivalence that can be determined through feasible algorithms from properties of operations in the log. Our definition is based on interference among operations as defined below.

3.3.2.1. Interference

Intuitively, operations interfere if different orders of execution result in a different state of an object or in a different result returned. For example, consider a data item $x$ that is accessed by two transactions $T_i = \{ t_{i,1}, \ldots, t_{i,n} \}$ and $T_j = \{ t_{j,1}, \ldots, t_{j,n} \}$. If $T_i$ contains a step $t_{i,k} = \text{read}(x)$ and $T_j$ contains a step $t_{j,l} = \text{write}(x, \text{val})$, the value read depends on the order of $t_{i,k}$ and $t_{j,l}$. We say that read-write interference exists between $t_{i,k}$ and $t_{j,l}$. Similarly, if both transactions contain steps $t_{i,k} = \text{write}(x, \text{val}_1)$ and $t_{j,l} = \text{write}(x, \text{val}_2)$, the final value of $x$ depends on the order of the write steps. We say that write-write interference exists between $t_{i,k}$ and $t_{j,l}$.

Transactions interfere if they contain interfering steps. The obvious generalization for nested transactions is to say that nested transactions interfere if they contain interfering subtransactions. For a formal definition of interference see [4].

We call read-write and write-write interference syntactic interference since they are based solely on the syntactic structure of a transaction and ignore the values
actually written or read.

Syntactic interference does not necessarily imply that the order of execution is reflected in the state of objects or in results returned. Consider, for example, transactions $T_1$ and $T_2$ with $T_1 = \{\text{write}_1(x, 7)\}$ and $T_2 = \{\text{write}_2(x, 7), \text{read}_2(x)\}$ with $\text{write}_2 <_2 \text{read}_2$. $T_1$ and $T_2$ interfere since suboperations $\text{read}_2(x)$ and $\text{write}_1(x, 7)$ interfere. However since both transactions write the same value the final value of $x$ is the same and all operations return the same result regardless of the order of execution of $T_1$ and $T_2$.

This observation lead us to define semantic interference.

**Definition 3.4:** Let $T_1 = \{t_{1,1}, \ldots, t_{1,n}\}$ and $T_2 = \{T_{2,1}, \ldots, T_{2,m}\}$. Two operations $t_{1,i} \in T_1$ and $t_{2,j} \in T_2$ semantically interfere, if the state of an object or the results returned by $T_1$ or $T_2$ reflect the order of execution of $t_{1,i}$ and $t_{2,j}$. We say that transactions semantically interfere if they contain semantically interfering suboperations. □

Notice that semantic interference takes into account the context in which an operation is executed. It is more general than syntactic interference: semantic interference between two transactions implies syntactic interference, but not vice versa, as the above example demonstrates. From now on we will use the word interference to mean semantic interference.

Interference between transactions creates dependencies indicating a flow of information. **Precedence dependencies** are a generalization of the read-write dependencies introduced in [4] and are used to describe dependencies introduced by semantic interference.
Definition 3.5: A pair of transactions $T_i, T_j$ in an execution log $L$ is related by a precedence dependency, written $T_i \rightarrow_p T_j$, if $T_i <_L T_j$ and $T_i$ and $T_j$ interfere. □

Similar to the equivalence conditions in [4], equivalence of execution logs can now be expressed in terms of dependencies.

Definition 3.6: Two execution logs $L_1$ and $L_2$ are $\delta$-equivalent, if

(i) $L_1$ and $L_2$ contain the same set of operations.

(ii) $L_1$ and $L_2$ contain the same set of precedence dependencies $\delta$. □

As discussed above a serial execution log represents a correct computation. Since equivalent logs are indistinguishable, any log equivalent to a serial log also represents a correct computation.

Definition 3.7: An execution log $L = (l_1, \cdots, l_N)$ is serial, if there is no interleaving between the steps of any two transactions. □

Definition 3.8: A nested execution log $L = (U, l_1, \cdots, l_N)$ is serial, if there is no interleaving between the steps of any two transactions $T_i$ and $T_j$ in the log, i.e.

$$T_i < T_j \iff \forall c_i \in \text{children}(T_i), c_j \in \text{children}(T_j), c_i < c_j.$$ □

Definition 3.9: An execution log is $\delta$-serializable, if it is $\delta$-equivalent to a serial log. □

Thus an execution is correct if it is $\delta$-serializable. In the following chapters we will prove the correctness of our concurrency control algorithms by showing that they allow only $\delta$-serializable execution logs.
3.4. Summary

In this chapter we presented a formal model of execution for operations on atomic objects. The model extends earlier work by Bernstein and Goodman [4] and Papadimitriou [40] to include nested operations.

Execution of operations is modeled by execution logs, and the notion of log equivalence is fundamental to determining correctness of an execution. We introduced log equivalence based on interference between operations. Two forms of interference are of interest:

(i) **Syntactic interference** is easy to determine from the syntactic structure of an operation. It is the basis for the most popular concurrency control methods, e.g. two phase locking [20]. However, we observed that syntactic interference does not fully capture our intuitive understanding of interference, and thus introduced

(ii) **Semantic interference**, which is based on properties of operations as well as the results of transactions in the logs and the state of objects upon completion of a transactions.

Semantic interference derives its generality from knowledge of the observable effects of a transaction, and not just the items on which the transaction acts.

We then defined $\delta$-serializability in terms of precedence dependencies which are introduced by interfering transactions in an execution log. $\delta$-serializability covers an entire spectrum of serializability. If interference is interpreted as read-write interference, $\delta$-serializability reduces to the familiar read-write serializability [4]. The most general interpretation of interference leads to the most general form of
serializability as computational equivalence to a serial execution.

While we would like concurrency control algorithms to enforce the latter, in many cases complexity arguments force us to settle for read-write serializability.

In the next chapter we develop a skeleton for a concurrency control algorithm that enforces \( \delta \)-serializability. It can be parameterized, reflecting different degrees of knowledge about semantic interference between operations. In Chapter 5 we present techniques to derive parameters for the algorithm. Some of those techniques are based on ideas relating to semantic equivalence, bringing the resulting executions closer to the general form of serializability. Moreover, these techniques are interesting because of their practicality. They utilize information that can be obtained from an object specification, using an interactive program development environment or simple data-flow analysis methods.
CHAPTER 4

Concurrency Control

The subject of this chapter is a new concurrency control algorithm for nested transactions on typed objects. The algorithm extends earlier approaches of incorporating operation semantics into concurrency control methods by Garcia-Molina and Schwarz [22, 49]. The novelty of the algorithm lies in its flexible conflict detection strategy. Unlike previous algorithms, it does not depend on a single built-in definition of conflict between operations, but detects conflicts by evaluating user supplied conflict predicates. The predicates are part of the object specification and provide a convenient way for the object designer to incorporate semantic knowledge about the operations into the concurrency control decision.

The algorithm maintains conflict information until the operations involved have committed. The additional bookkeeping overhead is offset by the potential increase in concurrency that can be realized using this information. Conventional concurrency control methods, e.g. locking [20] or timestamping [44] do not explicitly maintain dependency information and thus are limited in their potential to exploit semantic information.

In this chapter we will focus on the general algorithm. A discussion of the implications of our approach together with techniques for utilizing semantic information to increase concurrency is the subject of the next chapter.
The chapter is organized as follows. In Section 1 we introduce interference predicates to express conflicts between operations. In Section 2 we present the data structure maintained by the algorithm. In Section 3 we develop the concurrency control algorithm itself. Its correctness is proved in Section 4, and Section 5 summarizes the chapter.

4.1. Interference Predicates

Concurrency control algorithms enforce serializability by detecting conflicting operations and then imposing an order on their execution. Conflicts are detected through a conflict detection mechanism built into the algorithm, which in practice varies with the concurrency control algorithm. For example, algorithms based on locking [20] detect conflicts when two transactions attempt to acquire incompatible locks. Timestamp based algorithms [44] determine conflicts by comparing the times of the latest read and write access to an item with the time of the new operation.

The cost of detecting conflicts is a major obstacle to implementing concurrency control algorithms that enforce general forms of serializability. Loosely speaking, the cruder the detection strategy, the cheaper it is, but the less accurate it will be - signaling conflicts when none are present. Thus, most algorithms adopt a rather crude definition of conflict. While this results in inexpensive algorithms for conflict detection, it also reduces the potentially allowed interleavings of transactions and thus the amount of concurrency. Rather than settling for either read-write serializability or an inefficient algorithm to determine conflicts, our algorithm relies on the object designer to supply predicates expressing conflicts between operations.
These predicates are supplied as arguments to the algorithm, and thus the cost of the concurrency control method, i.e. the cost of evaluating the predicates, and the potential for concurrency is determined by the object designer.

For the purpose of developing the algorithm, we will adopt the traditional definition of conflict, namely interference as defined in the previous chapter. However, the flexibility of the predicates allows for much more specific definitions of conflict which are the subject of Chapter 5.

**Definition 4.1**: An interference predicate for operations \( o_i \) and \( o_j \) is a predicate 
CONF \([o_i(a_1, \ldots, a_n), o_j(b_1, \ldots, b_m), s]\) such that CONF evaluates to true if and only if \( o_i \) and \( o_j \) interfere. The parameters \( a_k \) and \( b_l \) are the arguments of the operations and \( s \) is a state parameter that denotes information about the state of the object.

Interference predicates are evaluated by the serializer when an operation is submitted for execution. At that time, the arguments to the operation are known and their values can be substituted for the parameters.

We expect that interference predicates will normally depend only on the parameters \( a_i \) and \( b_j \). Although the object state can be accessed, the utility of this information is limited by several constraints. The first is the cost of acquiring state information. Nothing is gained if determining the state requires more time than is saved by the potential increase in concurrency. Consequently \( s \) must be easy to determine.

Other factors are the consistency and timeliness of the information. To guarantee that the state information is consistent, it must be determined in one atomic step. This suggests executing a short read-only transaction in order to read the
state. Such a transaction will itself require concurrency control, however, raising questions of recursive termination and correctness which we prefer not to address here (for example, is the serialization order for such an imbedded transaction constrained by the serialization order for the transactions that issued it, or is it completely independent?) This problem seems to be a difficult one, and it is unlikely that the expected increase in concurrency warrants any substantial computational investment in concurrency control. However, $s$ is included in the model because there are special cases in which it can be useful.

Another concern is the timeliness (accuracy) of the state information. Since interference predicates are evaluated at the time an operation is scheduled the state might not be the same by the time the transaction is executed. For example, at scheduling time it might be determined that two insert operation access different branches of a search tree. However, by the time the operations are executed the tree could have been rebalanced so that the operations no longer access distinct branches. If they had been scheduled concurrently, the tree might be left in an inconsistent state. In general, $s$ will only remain accurate if it is limited to monotonic properties of the object state, i.e. to properties that remain true once they become true. In a banking environment, for example, the property that the time is after midnight on April 15 is monotonic. Once this property becomes true, new customer transactions will access fiscal records for the new fiscal year, permitting optimized concurrency control when those transactions are run concurrently with auditing transactions for the prior fiscal year.
The interference predicates in the example below express read-write interference for nested transactions.

**Example 4.1:** Read and write operations interfere, if they access the same data item. Read operations do not interfere with each other. Nested operations interfere if they contain interfering suboperations.

\[
\begin{align*}
\text{CONF}[\text{read}(a_r), \text{write}(a_w)] & : a_r \neq a_w \\
\text{CONF}[\text{write}(a_{w_1}), \text{write}(a_{w_2})] & : a_{w_1} \neq a_{w_2} \\
\text{CONF}[\text{read}(a_{r_1}), \text{read}(a_{r_2})] & : \text{false}
\end{align*}
\]

For nested operations \(\text{OP}_1(a_1)\) and \(\text{OP}_2(a_2)\) the interference predicate is:

\[
\text{CONF}[\text{OP}_1(a_1), \text{OP}_2(a_2)] = \bigvee_{\text{o}_i \in S_i, \text{o}_j \in S_j} \text{CONF}[\text{o}_i(a_i), \text{o}_j(a_j)]
\]

where \(S_i = \text{SUB} (\text{OP}_i) \cup \text{STEPS}(\text{OP}_i)\) and \(S_j = \text{SUB} (\text{OP}_j) \cup \text{STEPS}(\text{OP}_j)\).

4.2. Execution Snapshots

In this section we develop the data structure that is used by the concurrency control algorithm to maintain dependency information.

4.2.1. Precedence Dependencies

A precedence dependency between two operations indicates a serialization order between the operations. The order arises from two sources: Static ordering is inherent in the object specification and is expressed by the sequential composition operator ",," in the specification language. A static order is to be followed in every execution of the operation. Dynamic ordering is introduced by the serializer when deciding on a serialization order for operations.
Definition 4.2: Two operations $o_i$ and $o_j$ are related by a static precedence dependency if $o_i < o_j$ in every transaction involving both $o_i$ and $o_j$. □

Definition 4.3: Two operations $o_i$ and $o_j$ are related by a dynamic precedence dependency if $o_i < o_j$ in some execution involving both $o_i$ and $o_j$. □

A precedence dependency between nested operations induces dependencies between other operations. Static precedence dependencies introduced by the object specification imply a serial order between siblings in the transaction tree and consequently induce a serial order between the children of the operations they relate.

Dynamic precedence dependencies imply a serialization order between operations and induce the following dependencies: Firstly, for $o_i$ to precede $o_j$, all suboperations of $o_j$ must precede conflicting suboperations of $o_j$, thus introducing precedence dependencies between conflicting descendants. Secondly, since operations are atomic, a dependency between $o_i$ and $o_j$ implies a dependency between the enclosing transactions $parent(o_i)$ and $parent(o_j)$. Consequently, the order propagates to the highest distinct ancestors of $o_i$ and $o_j$.

4.2.2. Execution Snapshots

In order to determine a serialization order for new operations, the serializers maintain an snapshot of the execution history. The execution snapshot consists of the execution log of all uncommitted transactions, together with additional information to enforce precedence dependencies between the operations in the log.
The serializer keeps track of precedence dependencies by inserting appropriate precedence edges and wait-for edges into the execution snapshot. Static precedence dependencies between operations $o_i$ and $o_j$ are indicated by edges $o_i \rightarrow_{P_s} o_j$. Dynamic precedence dependencies are written $o_i \rightarrow_{P_d} o_j$. We sometimes write $\rightarrow_P$ to refer to a dependency of either sort.

Precedence dependencies are enforced by delaying appropriate operations until their conflicting counterpart has completed. The serializer uses wait-for edges in the snapshot to indicate the order of execution for conflicting operations. Wait-for edges are then used by the driver module to ensure that the operation at the head of the edge waits until the operation at the tail of the edge completes.

Their use for excluding certain operations from executing makes wait-for edges similar to locks. However the fact that they also indicate an order between operations makes them more general than locks. Their generality will become apparent in their use within the algorithm.

The following notation will be used in the algorithm. Operations that have been scheduled for execution are marked "x". We call an operation closed if all its suboperations are marked "x". Otherwise it is called open.

**Example 4.2:** Figure 4.1 shows a snapshot of the file system create operation. All transactions have been scheduled for execution and are marked "x", and the subtransaction with TID t.3 is closed. □
4.3. Algorithm 1

This section introduces the basic concurrency control algorithm. It presented in two parts: The first part relates to the invocation of a new operation and describes the insertion of new precedence and wait-for edges into the execution snapshot. The second part relates to the completion of operations and describes the promotion and deletion of wait-for and precedence edges. Execution of operations on atomic objects is discussed after Part 1.

4.3.1. Part 1: Operation Invocation

To simplify the description we have divided the algorithm into two phases. In an implementation the phases can be merged to obtain greater efficiency. In Phase 1 conflicting operations are identified, and in Phase 2 scheduling decisions for a new operation are made. Both phases are now described in detail. The detailed algorithm is shown in Figure 4.2.
Let $S$ be the current snapshot, and let $TP$ be the transaction template of the new operation $OP = \{o_1, \ldots, o_n\}$ to be serialized. The template is generally a tree, partially ordered by static precedence dependencies.

**Phase 1:**

For each $o_i$ in $TP$, bottom up

(1.1) Identify the set of conflicting operations $C_i$ in $S$ and $TP$.

(1.2) Insert the pair $(o_i, C_i)$ into a conflict queue.

**Phase 2:**

(2.1) Remove a pair $(o_i, C_i)$ from the conflict queue.

For each $o_j \in C_i$,

(2.2) determine highest distinct ancestors $(X, Y)$ of $o_j$ and $o_i$.

(2.3) If there is no precedence edge between $X$ and $Y$, then introduce $\rightarrow_P$ between $X$ and $Y$ such that the following conditions hold:

   (a) $\rightarrow_P$ does not introduce a cycle of precedence edges.

   (b) If the new edge is $X \rightarrow_P Y$, then $o_j$ is open, else $o_j$ is open.

(2.4) otherwise, reject $o_i$ and abort it.

(2.5) add edges $o_j \rightarrow_W o_i$ between all conflicting $o_j, o_i \in \text{Steps}(o_j)$ and $o_i \in \text{Steps}(o_j)$ such that $\rightarrow_W$ is consistent with $\rightarrow_P$.

Figure 4.2. Edge Insertion Algorithm

### 4.3.1.1. Phase 1: Identification of Conflicts

Upon receipt of the template $TP$ of a new operation $OP$, first determine the set of conflicting operations for $OP$ and all its suboperations by evaluating conflict predicates $CONF[o_j(a_j), o_i(a_i)]$ for each node $o_j(a_j)$ in the template and each operation $o_i(a_i)$ in the snapshot (step 1.1). Conflicting operations in the template are identified to handle concurrency within a transaction.
For the serialization algorithm to be correct, it is necessary that child transactions be processed before their parents. To ensure this, the template of the new operation is scanned bottom-up. For each node in the template, conflicting operations in the snapshot and the template are determined by a bottom-up search, starting at the leaves. This order is preserved for the second phase by inserting the operations into a FIFO conflict queue (step 1.2).

4.3.1.2. Phase 2: Deciding on a Serialization Order

After removing conflicting operations \( o_j \) and \( o_i \) from the conflict queue a serialization order for the highest distinct ancestors \( X \) and \( Y \) of \( o_j \) and \( o_i \) is determined and indicated by an appropriate precedence edge in the execution snapshot (steps 2.2, 2.3).

The selection of an order in which to execute conflicting operations is a database scheduling decision and closely related to query optimization problems [58]. Factors to be considered include the order in which operations arrive and the weight of an operation, i.e. an estimate of the amount of resources required to complete an operation. A discussion of scheduling policies is beyond the scope of this dissertation.

After the serialization order has been determined, the serializer inserts wait-for edges between conflicting suboperations of \( o_j \) and \( o_i \) to indicate the order of execution for these operations (step 2.5). In all cases the direction of the wait-for edges must be consistent with the precedence edge in step 2.3.
4.3.2. Executing Operations

Once operations have been serialized and inserted into the execution snapshot, the driver searches the snapshot for executable operations.

Definition 4.4: A single level operation $o_j$ in the snapshot is executable if there are neither incoming wait-for edges $o_i \rightarrow_w o_j$ nor incoming precedence edges $o_i \rightarrow_{P_k^*} o_j$ for any operation $o_i$. A nested operation is executable if at least one of its suboperations is executable. □

Once an executable operation has been scheduled, it is marked “x” to prevent the serializer from inserting inconsistent precedence edges. It is then translated into a sequence of physical operations which are relayed to the appropriate data managers for execution. Once the operation is completed, edges relating to that operation are rearranged as described below.

Definition 4.5: A single level operation is completed once its execution terminates. A nested operation is completed once all of its suboperations are completed. □

For now we assume that there are no failures (The effect of failures will be discussed in Chapter 6). Furthermore, since the algorithm does not introduce cycles of precedence edges (step 2.3a), it does not lead to deadlock. Thus operations can always be completed.

If operations become known dynamically, it might not always be possible to insert an operation into the graph without introducing a cycle. If an operation cannot be inserted the corresponding transaction must be aborted.
Note that completion of an operation does not imply that the effect of the operation will become permanent in the object data nor does it imply that its results are visible to other operations. After an operation is completed, the transaction manager decides to either commit or abort it, depending on its results. Only when the operation is committed are its results visible to other transactions within its scope, and only when the enclosing top-level transaction commits are changes to object data made permanent. The conditions under which operations commit are dependent on the object type.

4.3.3. Part 2: Operation Commitment

Part 2 of the algorithm handles commitment of operations. It is depicted in Figure 4.3 and described below.

Once an operation \( o_i \) commits, all precedence dependencies involving \( o_i \) have been enforced and the corresponding edges can be deleted from the snapshot. Wait-for edges are promoted to higher level operations as follows.

Let \( \mathcal{X} \) be a node corresponding to an operation that commits.

**Wait-for Edges**

For each edge \( X \rightarrow_{w} Y \) do

(3.1) if \( X \) is the tail of a precedence edge \( X \rightarrow_{p} Z \), delete \( X \rightarrow_{w} Y \)

(3.2) otherwise, replace \( X \rightarrow_{w} Y \) by \( \text{parent}(X) \rightarrow_{w} Y \)

**Precedence Edges**

(3.3) Delete all precedence edges \( X \rightarrow_{p} Y \).

Figure 4.3. Edge Deletion Algorithm
When the operation at the tail of a wait-for edge commits, the tail of the edge is promoted to the parent, since the operation at the head must now wait for the parent to commit. When the operation at the tail of a precedence edge commits, the precedence dependency has been enforced and the tail of the wait-for edge need not be promoted further. In this case the edge can be deleted.

Once a top-level transaction \( T \) commits, the template corresponding to \( T \) can be removed from the snapshot.

4.4. Correctness of Algorithm 1

In this section we prove the correctness of Algorithm 1. As discussed in Chapter 3, executions are correct if they are \( \delta \)-equivalent to a serial execution. A concurrency control algorithm is considered correct if it allows only \( \delta \)-serializable executions. This notion of correctness is based on the assumption that any serial order of execution is correct. Operation on atomic objects impose the additional constraint that the equivalent serial order be consistent with the partial order among suboperations which is prescribed by the object specification.

We now show correctness of the algorithm by showing that the execution logs generated by Algorithm 1 are \( \delta \)-serializable. Lemma 1 establishes that a new operation \( OP \) is ordered with respect to every conflicting operation in the execution snapshot. Lemma 2 shows that the partial order prescribed by the object specification is respected. Finally Lemma 3 establishes that the execution logs produced by Algorithm 1 can be extended to a serial log with the same set of pre-
precedence dependencies.

**Lemma 4.3:** $OP$ is ordered with respect to every conflicting operation.

**Proof:** In Phase 1, all conflicting operations are identified. Phase 1 is executed for all operations submitted to the serializer, hence it will be executed whenever operations that are not known at the time their parent operation is serialized are submitted. Since the conflict relations are symmetric, the order in which operations are submitted does not affect the value of the conflict predicate.

Operations are deleted from the snapshot only after they commit. Thus, every operation in the snapshot is ordered after conflicting operations that have committed. □

**Lemma 4.4:** The serialization order is consistent with the partial order in the specification of an operation.

**Proof:** Since precedence edges are never reversed in the algorithm, we only need to show that once a precedence edge is deleted, no conflicting edge is introduced in the opposite direction.

Precedence edges are only deleted when the operation at the tail of the edge commits(3.3). Since operations commit only when all their suboperations are committed, the operation will be closed at that point. No edges with the operation at the head being closed are ever introduced (2.3). Therefore, the algorithm respects static precedence dependencies and produces execution logs that are consistent with the specifications of the operations. □

**Lemma 4.5:** Algorithm 1 schedules a new operation $OP$ such that the resulting execution log is equivalent to a serial log with the same set of precedence
dependencies.

Proof: Let $L$ be the current execution log with equivalent serial log $L_s$, and let $L'$ be the log that results after scheduling $OP$. We show that $OP$ is ordered with respect to any conflicting operation $C$ in the snapshot, and that $L'$ can be extended to a serial log $L_{s}'$ with the same set of dependencies. Furthermore, we show that $L_{s}'$ is an extension of $L_s$.

The proof proceeds by induction on the depths $d = d_C + d_{OP}$ of nesting of suboperations.

$d = 0$:

In this case, neither $C$ nor $OP$ has any suboperations.

In line 2.3 of the algorithm a serialization order for $C$ and $OP$ is determined. Such an order might already exist due to serialization decisions involving other operations. Otherwise, a new order is introduced such that it does not introduce a cycle of precedence dependencies (2.3, 2.4). It is indicated by inserting a precedence edge between the highest distinct ancestors $X$ and $Y$ of $C$ and $OP$, respectively. Without loss of generality assume the direction of the edge is $X \rightarrow P Y$.

The order is enforced by introducing wait-for edges (2.5) $c_i \rightarrow_w o_{ij}$ for all $c_i \in \text{Steps}(C)$ and $o_{ij} \in \text{Steps}(OP)$ such that $C(c_i, o_{ij}, a_i, a_j) = \text{true}$. The direction of the edges is consistent with $\rightarrow_p$. Thus, all dependencies will be directed from $C$ to $OP$.

The tail of the wait-for edge $c_i \rightarrow_w OP$ is promoted to $C$ when $c_i$ commits. It is
only deleted once he operation at the tail of $\rightarrow_p$ commits. Thus, $OP$ cannot start executing until $C$ commits.

Steps $o_i$ with no dependencies do not conflict with any suboperation $c_i$ and thus can be reordered such that $c_i$ is executed before $o_i$ for all $c_i \in Steps(C)$. Hence the resulting execution has the same set of dependencies as a serial log where $OP$ is executed only after $C$ committed.

For the induction step we assume that the hypothesis is true for all pairs of transactions with $d$ or fewer levels of nesting.

$d \to d+1$

As before, a serialization order between $C$ and $OP$ is determined in line (2.3) of the algorithm. We assume again that the order is $C \rightarrow_p O$.

Since operations are serialized bottom up, it follows from the induction hypothesis that the set of dependencies between suboperations of $C$ and $OP$ is the same as for the serial order where each $c \in SUB(C)$ is followed by $OP$. Similarly, $SUB(OP)$ can be ordered serially with respect to $C$. It thus remains to be shown that $C$ and $OP$ can be ordered with respect to each other, and that the order is consistent with the order between their suboperations.

Since wait-for dependencies between suboperations $c_i \in OPS(C)$ and $o_i \in OPS(OP)$ are introduced consistent with $\rightarrow_p$ (line 2.5), new serialization decisions are consistent with any previous ones.

Since the tails of wait-for edges are promoted (3.2) until the operation at the tail of $\rightarrow_p$ commits, the conflicting operations in $OP$ cannot begin until $C$ com-
mits. Consequently, $OP$ can be ordered after $C$ without changing the set of dependencies.

The claim that the equivalent serial log $L'_S$ is an extension of $L_S$ follows since precedence edges are never directed towards closed operations (step 2.3); thus, any order in $L_S$ will be respected for subsequent serialization decisions. 

Thus we established the following theorem:

**Theorem 4.6:** The execution logs produced by Algorithm 1 are δ-serializable.

□

**Corollary 4.7:** If dependencies are interpreted as read-write or write-write dependencies, Algorithm 1 enforces read-write serializable schedules. □

4.5. Summary

This chapter introduced a new concurrency control algorithm for nested transactions. The algorithm enforces ordering relations between conflicting operations that are expressed as precedence dependencies between operations. We proved that execution logs generated by the algorithm have the same set of dependencies as a serial log.

The novelty of our algorithm lies in its generality. It gives the system designer the freedom to include semantic information into concurrency control decisions, thus increasing the level of concurrency. Semantic information is supplied as part of an object's specification in form of conflict predicates for pairs of operations. The predicates refer to the particular implementation of operations and may incorporate operation parameters as well as information about the state of an object.
Conflict predicates cover a wide spectrum of semantic information. They can be used to express the traditional notion of read-write and write-write conflicts, in which case the algorithm enforces read-write serializability for nested transactions similar to Moss' schema [37]. A more general notion of conflict can be accommodated by formulating predicates that express interference, resulting in an algorithm similar to the one proposed by Schwarz [49].

The use of the algorithm is geared towards atomic objects, where the set of transactions is limited, and their semantics are known a priori. In such an environment the conflict predicates can be supplied together with the object specification. Given a suitable programming environment the object designer can formulate different predicates and experimentally determine the predicate that is most appropriate for a particular application. Object specifications, however, contain predetermined partial orders on suboperations, and, unlike other concurrency control methods, the serialization order produced by our algorithm is consistent with this predefined order.

In the next chapter we will introduce modifications to the algorithm that allow even higher concurrency than the basic algorithm through ideas relating to conflict serializability defined in Chapter 3.
CHAPTER 5

Increasing Concurrency

The goal of this chapter is to develop a collection of techniques for improving the performance of operations on atomic objects. The techniques exploit semantic knowledge about operations in order to increase concurrency among operations. In contrast to general database systems, the semantics of operations on atomic objects are known to its programmer, and our techniques therefore rely on the object designer to supply this semantic information together with the object specification.

Our optimization techniques are based on two observations: Equating conflict between operations with interference unnecessarily restricts the level of concurrency. Interfering operations can be commutative, and the effect of a pair of commutative operations is independent of the order in which they are executed, even if they interfere. Thus, the execution order that was used can be ignored during subsequent serialization decisions. In the context of nested operations, this reduces the constraints on serialization decisions for higher level operations and may allow two transactions to run concurrently that would otherwise have been run serially, leading to higher concurrency within operations.

The second observation relates to the availability of results. In many cases results of an operation might be available before the entire operation completes. In particular, if an operation has side effects, the result it eventually returns might be available even before the suboperations establishing the side effect are started. For example, in the create operation of the file system in Chapter 2, the result is
available and could be returned before the dict.balance operation is invoked. Any operation depending on such a result can then be started earlier, leading to higher concurrency within as well as among operations.

The chapter is divided into three sections. Commutativity of operations is the topic of Section 1, Section 2 discusses early availability of results. Both sections contain appropriate modifications to the concurrency control algorithm and extensions to the specification language. Section 3 concludes the chapter with a discussion of our results.

5.1. Commutativity of Operations

In this section we investigate commutativity of operations and its implications on concurrency control algorithms. The goal is to extend our new concurrency control algorithm to allow commutative operations to be executed in any order without influencing the order of enclosing operations. The following example will illustrate this idea, for now relying on the intuitive meaning of commutativity of operations. A formal definition is given later in this section.

Example 5.1: Consider the execution snapshot in Figure 5.1 (a). The increment operations obviously interfere since they both contain read and write operations on the same counter \( x \). Therefore they have to be serialized with respect to each other. The parent operations \( A \) and \( B \) must be serialized because of interference of the read and write operations on data item \( y \). Under the traditional definition of conflict as interference it is required that \( A \) and \( B \) be serialized in the same order as the increment operations. These constraints would not allow the execution log depicted in Figure 5.1 (b), since there \( A \rightarrow B \) and \( B \rightarrow A \).
Notice however that the results returned by A and B are independent of the order in which the increment operations are executed, as long as they are executed atomically. The event ordering that determines the serialization order for A and B is the order in which the operations read(y) and write(y) are executed. □

Our goal is to limit the scope of conflicts and extend the skeleton algorithm of Chapter 4 to allow interleavings such as Figure 5.1 (b), when supplied with appropriate information.

5.1.1. Commutativity and Conflicts

Commutativity of operations is not an entirely new concept in concurrency control. Other algorithms, including Algorithm 1, enforce a serialization order only between conflicting operations; no order needs to be enforced among non-conflicting operations. Clearly, non-conflicting operations commute. However, in this view conflicts and commutativity are interpreted in a rather narrow sense. Conflict is equated with interference, and consequently operations only commute if
they do not interfere. Note however that in the above example the increment operations are commutative, despite interference of the read and write suboperations.

This observation results in a new interpretation of conflict and commutativity: Conflicts between operations no longer depend only on interference among suboperations, but also on the state of the object and the semantics of the operations.

**Definition 5.1:** Two operations \( o_i \) and \( o_j \) are **commutative**, if

(i) executing them in either order leaves the object in the same state and,

(ii) they return the same result in both orders of execution. □

In this new view of commutativity, a pair of non-interfering operations still satisfies the definition of commutativity. However, commutative operations might interfere and thus have to be serialized with respect to each other. Only *after* they have been executed is the order irrelevant and can be ignored when determining a serialization order for ancestors or siblings in the transaction tree.

Commutativity of operations is expressed through **commutativity predicates**.

**Definition 5.2:** A commutativity predicate for two operations \( O_1 \) and \( O_2 \) is a predicate \( COMM[O_1(a_1, \cdots, a_n), O_2(a_1, \cdots, a_k), s] \) such that \( COMM \) evaluates to true if and only if \( O_1 \) and \( O_2 \) commute. The parameters \( a_j \) refer to arguments of the operations and \( s \) to the state of the object (see discussion in Section 4.1). □

Commutativity information leads to a tighter definition of conflict, since conflicting suboperations no longer unconditionally propagate their conflict to parent operations. Commutativity predicates are a means to evaluate the relevance of a particular conflict for higher level operations. They allow to restrict the propagation
of conflicts to higher level operations, effectively confining conflicts to the smallest scope in which they are relevant. For the sake of brevity we will omit the state parameter from the predicates.

The conflict predicate for a pair of operations is constructed from the conflict and commutativity predicates of their suboperations as follows.

Let \( OP_1 = \{ o_1, \ldots, o_n \} \) and \( OP_2 = \{ p_1, \ldots, p_m \} \), and let \( CONF [o_i, p_j] \) and \( COMM [o_i, p_j] \) be the conflict and commutativity predicates for all pairs of operations \( o_i \) and \( p_j \). The conflict predicate for \( OP_1 \) and \( OP_2 \) is defined as

\[
CONF [OP_1, OP_2] = \bigvee_{i,j} (CONF [o_i, p_j] \land \neg COMM [o_i, p_j]).
\]

In the following example we assume that the conflict predicates for the write operations express the traditional definition of write-write conflicts, i.e. two writes conflict if they access the same data item. For example, assuming that \( inc_1 \) and \( inc_2 \) commute with each other and with the write operations, the conflict predicate for operations \( A \) and \( B \) above is

\[
CONF [A, B] = CONF [w_1, inc_2] \land false
\]

\[
\bigvee CONF [w_1, w_2] \land \neg COMM [w_1, w_2]
\]

\[
\bigvee CONF [inc_1, w_2] \land false
\]

\[
\bigvee CONF [inc_1, inc_2] \land false
\]

which reduces to

\[
CONF [w_1, w_2] \land \neg COMM [w_1, w_2]
\]

Since \( CONF [w_1, w_2] = true \), \( CONF [A, B] = \neg COMM [w_1, w_2] \) and \( A \) and \( B \) conflict if the write operations are not commutative.
For the remainder of this chapter we assume that conflict predicates are constructed in this fashion, and that the predicates for primitive operations express read-write and write-write conflicts between these operations.

5.1.2. Extending Algorithm 1

We now describe two extensions to Algorithm 1, both of which increase the level of concurrency by exploiting the fact that operations can commute despite interfering with each other. The term "commutative" in the following two sections is to be interpreted as "commutative but interfering". Obviously non-interfering operations are commutative and their order of execution is unimportant.

5.1.3. Ordering of Commutative Operations

The first extension relaxes ordering constraints by allowing commutative suboperations to be executed in any order. As discussed before, operations have to be serialized with respect to all other operations that they conflict with. However, after they are committed, the order in which commutative operations were executed can be ignored for future serialization decisions, since it is neither reflected in the state of the object nor in the result of the operations. Hence the effects of all future operations will be the same, regardless of this order.

The modified algorithm is shown in Figure 5.2 and 5.3 and described in detail below.

The basic difference to Algorithm 1 is the labeling of edges introduced in steps (2.6) and (2.7). Wait-for edges $o_j \rightarrow_w o_i$ are labeled with the pair of operations between which they enforce a precedence dependency. If there are no commuta-
Let $S$ be the current snapshot, and let $TP$ be the transaction template of the new operation $OP = \{o_1, \cdots, o_n\}$ to be serialized. The template is generally a tree, partially ordered by static precedence dependencies.

**Phase 1:**
For each $o_i$ in $TP$, bottom up

1.1 Identify the set of conflicting operations $C_i$ in $S$ and $TP$.
1.2 Insert the pair $(o_i, C_i)$ into a conflict queue.

**Phase 2:**

2.1 Remove a pair $(o_i, C_i)$ from the conflict queue.
   For each $o_j \in C_i$,
2.2 determine highest distinct ancestors $(X, Y)$ of $o_j$ and $o_i$.

2.3 If there is no precedence edge between $X$ and $Y$, then introduce $\rightarrow_p$ between $X$ and $Y$ such that the following conditions hold:
   (a) $\rightarrow_p$ does not introduce a cycle of precedence edges.
   (b) If the new edge is $X \rightarrow_p Y$, then $o_i$ is open, else $o_j$ is open.

2.4 otherwise, reject $o_i$ and abort it.

2.5 add edges $o_k \rightarrow_w o_i$ between all conflicting $o_k \in \text{Steps}(o_j)$ and $o_i \in \text{Steps}(o_i)$ such that $\rightarrow_w$ is consistent with $\rightarrow_p$.

Figure 5.2. Modified Edge Insertion, Part 1
Let \( X = X_1, \ldots, X_k = c \) be the nodes on the path from \( c \) to \( X \) and let \( Y = Y_1, \ldots, Y_l = o_i \) be the nodes on the path between \( o_i \) and \( Y \).

(2.6) If for all operations \( X_m \) and \( Y_n \) the predicate \( \text{COMM}_{X_m, Y_n} = \text{false} \):

(2.6.1) Label \( \rightarrow_W \) and \( \rightarrow_P \) with \( (X, Y) \) or \( (Y, X) \) respectively.

(2.7) If there are operations \( X_m \) and \( Y_n \) such that \( \text{COMM}_{X_m, Y_n} = \text{true} \), let \( m \) and \( n \) be maximal in lexicographical order.

(2.7.1) Label \( \rightarrow_W \) and \( \rightarrow_P \) with \( (X_m, Y_n) \) or \( (Y_n, X_m) \) respectively.

Figure 5.3. Modified Edge Insertion, Part 2

tive operations among the ancestors of \( o_i \) and \( o_j \), the edge \( \rightarrow_W \) enforces a precedence dependency between the highest distinct ancestors \( X \) of \( o_j \) and \( Y \) of \( o_i \), and the edge is labeled \( (X, Y) \). If commutative ancestors exist, the edge is labeled with the lowest pair of commutative ancestors \( (X_i, Y_j) \) of \( o_j \) and \( o_i \) respectively.

Similarly, the precedence edge between highest distinct ancestors of conflicting operations is labeled with the lowest pair \( (A_i, B_j) \) of commutative suboperations, indicating that the ordering constraint must be observed while \( A_i \) and \( B_j \) are executing.

A wait-for edge is deleted once the precedence dependency it enforces has been established. This is the case when the operation at the tail of the precedence edge commits (step 3.3). Precedence edges are deleted, once one of the operations between they enforce an order commits (step 3.1).

We now prove the correctness of the extended algorithm. Since in this version of the algorithm precedence edges can be deleted before the operation at the tail of the edge commits, it is possible that cycles of precedence dependencies are introduced. However, these cycles never consist entirely of dependencies caused by
**Precedence Edges**

Let $X$ be a node corresponding to an operation that commits.

(3.1) Delete all edges $X \rightarrow_{P} Y$ where $Y$ is any operation.

(3.2) Delete all precedence edges labeled $(X, Y)$, where $Y$ is any operation.

**Wait-for Edges**

Let $X$ be a node corresponding to an operation that commits and let $(A_1, B_1), \ldots, (A_m, B_m)$ be the labels on the $\rightarrow_{W}$ edges originating in $X$.

For each edge $X \rightarrow_{W} Y$ with label $(A_i, B_i)$

(3.3) if $A_i = X$ delete $X \rightarrow_{W} Y$.

(3.4) if $A_i$ is a predecessor of $X$ replace edge $X \rightarrow_{W} Y$ by $parent(X) \rightarrow_{W} Y$.

Figure 5.4. Modified Edge deletion.

---

non-commutative operations. The following lemma shows that despite these cycles there exists a serial order of execution that yields the same results.

**Lemma 5.2:** Let $OP$ be a new operation to be serialized. The execution log generated by the extended algorithm is conflict equivalent to a serial log.

**Proof:** We distinguish two cases:

(a) If there are no cycles of precedence dependencies, the statement follows from the proof of Lemma 4.6.

(b) Assume that there exists a cycle of precedence dependencies including $OP$.

Let

$$A_1 \rightarrow A_2 \rightarrow \cdots \rightarrow A_n \rightarrow OP$$

be the smallest such cycle.

Since the algorithm never generates a cycle of precedence edges (2.3), at
least one of the edges representing the dependencies must have been deleted
prior to scheduling $OP$. Without loss of generality we assume that $A_1 \rightarrow_p A_2$
has been removed. $A_1$ is open and thus not committed, since otherwise $OP$
would not have been scheduled before $A_1$. Hence the edge has been deleted
because one of two commutative suboperations $a_1$ of $A_1$ and $a_2$ of $A_2$ committed (step 3.2). Assume that $a_1$ is committed. Since $A_1$ is open, we can write
$A_1$ as $A_1 = (A_1', a_1, A_1'')$, where $A' a$ are the committed suboperations in $A$, and $A_1''$ are the unexecuted suboperations of $A_1$.

We now show that the execution $(A_1', a_1), A_2, \ldots, A_n, OP, A_1''$ which is generated by the algorithm is equivalent to the serial execution $A_2, \ldots, A_n, OP, A_1$.

The suboperation $a_1$ does not conflict with any operation $A_i$, $i = 2, \ldots, n$, since otherwise there would be a dependency between $a_1$ and the $A_i$ it conflicts with and thus a shorter cycle exists. It does not conflict with $OP$, since $OP$ is not scheduled before a closed operation (step 2.3).

Consequently, $a_1$ commutes with $A_2, \ldots, A_n, OP$, and we can "bubble" it to the right without changing the result of the computation. The same argument holds for all suboperations in $A_1'$, and thus they can be executed after $OP$ as well without changing any result.

Hence, $(A_1', a_1), A_2, \ldots, A_n, OP, A_1''$ is equivalent to $A_2, \ldots, A_n, OP, A_1$. $\square$
5.1.3.1. Early Promotion of Wait-for Edges

The second extension to Algorithm 1 allows the promotion of wait-for edges before the operation at the tail of the edge commits. The following example illustrates the idea.

**Example 5.3**: Consider the execution snapshot in Figure 5.5. Following Algorithm 1, the tail of the wait-for edge is promoted to op once \( \alpha_1 \) commits. It is deleted only when \( op \) commits and thus \( \alpha_3 \) is executable only after \( op \) committed. However, if \( \alpha_2 \) and \( \alpha_3 \) are commutative, \( \alpha_3 \) could be executed before \( \alpha_2 \) without influencing the result, thus reducing overall latency incurred by \( OP \).

Figure 5.6 shows the modification to the edge deletion algorithm that implements this idea. Once an operation commits, the tail of all wait-for edges is promoted to the parent node. The tail is propagated further if the unexecuted siblings of the operation at the tail commute with the operation at the head.

---

![Figure 5.5. Promote Example](image-url)
Wait-for Edges

Let $X$ be a node corresponding to an operation that commits and let $(A_1,B_1), \ldots,(A_m,B_m)$ be the labels on the $\rightarrow_w$ edges originating in $X$.

For each edge $X \rightarrow_w Y$ with label $(A_i,B_i)$ do

1. If $A_i = X$ delete $X \rightarrow_w Y$.
2. If $A_i$ is a predecessor of $X$ replace edge $X \rightarrow_w Y$ by $\text{parent}(X) \rightarrow_w Y$.

   For each edge $Y \rightarrow_w Z$ with label $(A, B)$ do

   If $\text{PROMOTE}(Y, Z)$ is TRUE /* PROMOTE described below */

3. If $Y = A$, delete $Y \rightarrow_w Z$.
4. If $Y \neq A$, replace $Y \rightarrow_w Z$ by $\text{parent}(Y) \rightarrow_w Z$.

\text{PROMOTE}(X, Y)

\text{PROMOTE}(X, Y) \text{ evaluates to TRUE, if } X \rightarrow_w Y \text{ can be promoted to } \text{parent}(X) \rightarrow_w Y. \text{ Let } S_1, \ldots,S_i \text{ be the unexecuted siblings of } X.

In a conservative setting, the definition of \text{PROMOTE} is as follows:

\text{PROMOTE}(X, Y) = \text{TRUE, if for all } i, \text{ COMM}_{S_i} \text{ Y } = \text{TRUE}

In an optimistic setting, \text{PROMOTE}(X, Y) is \text{TRUE if the above condition is "expected" to be true.}

Figure 5.6. Promote

\textbf{Lemma 5.4}: Early promotion of wait-for edges does not introduce non-serializable executions.

\textbf{Proof}: Since early promotion of wait for edges does not involve any precedence edges, it does not influence the edge insertion part of the algorithm. The execution of $o_j$ before the unexecuted siblings of $o_i$ yields the same result as first executing the siblings and then $o_i$, since they are pairwise commutative. $\Box$
5.1.4. Specifying Commutativity

Internally, commutativity information is stored as a list of predicates associated with each operation. The specification language for atomic objects has been extended to allow the object designer to provide this list with the object specification.

Commutativity is expressed in a commutativity clause, started by the reserved word `commute`, followed by a list of commutativity predicates, and closed by `end`. A commutativity clause for operations \( \textit{op}_1 \) and \( \textit{op}_2 \) is expressed as follows:

\[
\text{commute} \\
\quad \textit{op}_1, \textit{op}_2 : \text{COMM} \left[ \textit{op}_1 \left( a_1, \ldots, a_k \right), \textit{op}_2 \left( a_1, \ldots, a_l \right), s \right]; \\
\text{end}
\]

where \( \text{COMM} \left[ \textit{op}_1 \left( a_1, \ldots, a_k \right), \textit{op}_2 \left( a_1, \ldots, a_l \right), s \right] \) is the commutativity predicate for \( \textit{op}_1 \) and \( \textit{op}_2 \).

**Example 5.5**: Figure 5.7. shows a possible commutativity clause for the file system. □

---

\[
\text{commute} \\
\quad \text{create, delete} : fn_1 \neq fn_2 \\
\quad \text{read, write} : fn_1 \neq fn_2 \\
\quad \text{write, write} : fn_1 \neq fn_2 \lor \text{buffer}_1 = \text{buffer}_2 \\
\quad \text{read, read} : \text{true} \\
\text{end}
\]

Figure 5.7. Commutativity Clause for File System
5.2. Early Availability of Data

In this section we examine the function of static precedence dependencies and explore a technique to reduce latency caused by these dependencies.

A static precedence dependency \( o_i \rightarrow_{P} o_j \) is introduced by the sequential ordering of operations \( o_i \) and \( o_j \) in the object specification. Sequential execution is essential if one of the operations needs the result of the other as an argument. Note that in this situation the nested transaction mechanism is not to isolate the user from the effects of concurrency, but to enforce smaller units of recovery through the use of subtransactions.

The result of an operation is determined by the primitive steps used to implement the operation and possibly by some of its suboperations. Other primitive steps and suboperations might be used to establish side effects.

**Example 5.6:** Consider the create operation of the file system depicted in Figure 5.8. The operations necessary to establish the result (ERROR or OK) of create are the allocate and dict.search operations in lines (1) and (2), and the test in line (3). The deallocate operation in line (4) and the insert and balance operations in lines (5) and (6) establish side effects. □

In the above example, the result of the operation is returned only after either the deallocate or the insert and balance operations are completed, depending on the outcome of the test. Thus, any operation with a static precedence dependency on the create operation is forced to wait until the side effects of the create are established.
\texttt{create}\texttt{(string fn, int n);} \\
{ 
\begin{enumerate}
\item \texttt{cobegin}
\begin{enumerate}
\item \texttt{ptr = allocate(n);} \\
//
\item \texttt{loc1 = dict.search(fn);} \\
\end{enumerate}
\texttt{coend}
\item \texttt{if( loc1 != NIL)}
\begin{enumerate}
\item \texttt{deallocate(ptr, n);} \\
\texttt{return(ERROR);} \\
\end{enumerate}
\item \texttt{else}
\begin{enumerate}
\item \texttt{dict.insert(fn, "loc", loc1, "size", n, "offset", 0);} \\
\item \texttt{dict.balance();} \\
\texttt{return(OK);} \\
\end{enumerate}
\end{enumerate}
\}

Figure 5.8. Order of suboperations of create

Our latency reduction technique is based on the distinction between returning the result of a suboperation and completion of the suboperation. Results are returned as soon as they are available, and static precedence dependencies are consequently removed as soon as the results of the operation at the tail have been returned.

Before we describe the changes to the algorithm we must discuss some of the implications on the transaction system. If results of operations are made available before an operation actually commits, cascading aborts might occur if an operation aborts after its results have been revealed. In a general transaction system the possibility of cascading aborts requires extensive overhead to ensure consistency if cascading aborts indeed happen. In many cases the gain in efficiency does not
justify the overhead, and many systems simply do not allow cascading aborts to occur. In a nested transaction system the additional overhead is much less severe. Since static precedence dependencies never exist between top-level transactions, a chain of aborts never spans multiple transactions and thus is always confined to a single transaction tree. Instead of initiating an abort for every transaction in the chain, the version store mechanism can be used to abort all of the subtransactions in the chain simultaneously by restoring the version of the transaction that contains the entire chain. In any case, it would not be a good choice to use this technique unless the probability of aborts is known to be fairly low.

5.2.1. Algorithm

The new algorithm entails only a simple extension to the previous one. It is shown in Figure 5.9. Precedence dependencies are removed once the result of the operation at the tail of the dependency has been returned.

5.2.2. Specifying Early Availability of Results

Early availability of data is expressed in the object specification by separating the return of results from the return of a procedure call. Figure 5.10 shows the specification of the create operation revised to include the ideas from this section.

---

**Static Precedence Dependencies:**

Let $X$ and $Y$ be operations with $X \rightarrow_{P_s} Y$

- remove $\rightarrow_{P_s}$ as soon as the result of $X$ is returned

Figure 5.9. Removing Static Precedence Dependencies
create(string fn, int n);
{
    cobegin
        ptr = allocate(n);
        //
        loc1 = dict.search(fn);
    coend
    if( loc1 != NIL)
    {
        return(ERROR);
        deallocate(ptr, n);
    }
    else
    {
        return(OK);
        dict.insert(fn, "loc", loc1, "size", n, "offset", 0);
        dict.balance();
    }
}

Figure 5.10. Create with Early Results

5.3. Summary

This chapter introduced a technique to confine the scope of conflicts using information about commutativity of operations. The technique is an extension of [49], where a method is proposed to distinguish between essential and non-essential dependencies for single level transactions.

We extended the concurrency control algorithm developed in Chapter 4 to take advantage of the scope information. The first extension concerned the propagation of the serialization order of suboperations to enclosing operations. The new algorithm does not propagate the order of commutative suboperations once the operations are completed. This confinement of the scope of conflicts leaves a greater
degree of freedom for serialization decisions for higher level operations and results in executions logs with increased concurrency.

The second extension concerned the early propagation of wait-for dependen-
cies. It is similar to Kung's optimistic concurrency control techniques [31]. The advantage of our algorithm over optimistic techniques is its ability to take into account information about future operations. This information is again supplied in the object specification in terms of predicates. As a result, our algorithm is less prone to cascading aborts than methods that do not use such information. Even if cascading aborts should occur, they are confined within a single nested transaction tree and thus can be handled efficiently by the transaction mechanism.

The last extension reduced latency of transactions by making results available as soon as possible. The reduction was based on decoupling the return of results from the termination of a procedure.
CHAPTER 6

Resilient Objects

In this chapter we show how to implement atomic objects in a distributed system. One of the main advantages of distributed systems over the traditional single site processor is that a single processor failure does not render the entire system inoperative. In such an environment, replication of services and data at several sites can provide the basis for highly fault-tolerant applications. However, the complex interactions necessary to keep such a system consistent and the lack of tools to manage this complexity have been a major obstacle for the realization of fault-tolerant applications. The extension of atomic objects to a distributed environment is intended to fill this gap. To do so, extended atomic objects must accommodate properties peculiar to distributed systems, e.g. site failures, communication failures, and the lack of a global clock in the system.

Birman and Joseph [5] discuss an extension of atomic objects that accommodates most of these conditions. It is based on a small set of communication primitives, and does concurrency control using a naive extension of 2-phase locking. A consequence is that although this system supports nesting, nested objects incur such high overhead as to be almost useless. Below we will use similar primitives to implement the concurrency control algorithms defined in this dissertation in a distributed environment. The additional concurrency gained through our basic algorithm should enhance the viability of atomic objects as a design tool for fault-tolerant distributed applications. Following ISIS terminology [8,5] we will call the resulting
fault-tolerant objects resilient objects.

The chapter is divided into three sections. In Section 1 we discuss distributed systems and illustrate some of the problems the implementation of resilient objects must deal with. In Section 2 we define resilient objects, sketch the communication primitives and show how to use them to implement our concurrency control methods in a distributed system. Section 3 summarizes the chapter.

6.1. Distributed Systems

A distributed system consists of a collection of processors, called sites, connected by a communication network. Processors do not share memory, and processes running at different sites communicate solely by sending messages over the communication system. In the absence of failures, the network provides an addressing mechanism that allows a process to communicate with any other process in the system. We assume that the communication system provides mechanisms to mask communication failures. For example, checksums can be used to detect when a message has been corrupted by noise in a link, sequence numbers together with retransmission algorithms [53] prevent messages from getting lost or duplicated.

An important parameter in a distributed system is the upper bound on message transmission time. If no such bound exists, the system is called asynchronous. Fisher et al. [21] have shown that it is impossible to design deterministic protocols to reach agreement among processes in such a system. A number of probabilistic agreement methods have been developed for such systems [43, 55]. Most practical systems assume an upper bound on message transmission time and we restrict our
attention to systems in which such a bound is known.

The failure behavior of sites leads to a second classification of distributed systems. *Halting failures*, where sites fail by simply stopping, are the most benign failure mode. If other sites have the capability to detect such failures, e.g. through timing out on a message, the failures are called *fail-stop* [47]. Other failure classes include *omission* failures [41], where sites occasionally fail to send or to receive a message, and *malicious* or *Byzantine* failures, where sites may even conspire against achieving a common goal. For the purpose of this dissertation, we consider a distributed system where sites incur only fail-stop failures. We also assume that message loss and message corruption is handled by a low level communication layer. Thus our protocols are based on reliable communication between operational sites. The final assumption concerns *network partitioning*. We assume that all operational sites in the system can communicate with each other. Protocols to handle partitioned systems are beyond the scope of this dissertation and are discussed by ElAbbadi et al. [19, 18].

6.2. Resilient Objects

In this section we describe fault-tolerance properties of resilient objects, and then show how to enforce these properties in a distributed system.

6.2.1. Properties of Resilient Objects

A k-resilient object (or just resilient object when k is implicit) consists of k + 1 or more *components* which may be distributed across the sites in a distributed system. The components cooperate to present the abstraction of a single site atomic
object to the user.

In addition to the properties of an atomic data type, e.g. data abstraction and protection, $k$-resilient objects have several additional properties:

Consistency

The object behaves like a non-distributed object that executes requests serially.

Availability

If there are no more than $k$ site failures, the object continues to accept and process requests.

Progress

If there are no more than $k$ site failures, then operations are executed to completion, despite failures.

Recovery

(i) Partial. If the number of failures does not exceed $k$, a failed component restarts automatically when it recovers.

(ii) Total. If the number of failures is greater than $k$, failed components restart automatically when all the failed sites recover.

Below, we describe in detail how the consistency constraints are met. Availability is guaranteed by replicating objects at more than $k$ sites. Each instance of an object at a site is called a copy. A checkpoint/restart mechanism among the copies ensures progress despite failures of individual copies. Such a mechanism is described by Birman and Joseph [6]. The recovery problem is discussed in detail in [49]; we will discuss it briefly in Section 6.2.3.
6.2.2. Implementing Resilient Objects

In this section we show how to meet the consistency constraints in a distributed system. In particular, we show how to implement our concurrency control algorithm in an environment where sites can fail and recover independently.

To guarantee that the algorithm enforces consistency, i.e. serializable execution of operations, a number of properties of distributed systems have to be dealt with. Most importantly, the algorithm must be tolerant to site failures during its execution. It also must function correctly if sites recover. This raises two concerns: Once a site fails, all other sites involved in the algorithm must take action based on the same perception of the status of sites (operational or failed) throughout the system. In particular, the action must be based on either all sites knowing that the failure occurred or none of them knowing about it. Otherwise, if some of the sites base their action on the failure having occurred and others on it not having occurred, inconsistent actions might result. The second concern is that all sites affected must actually be involved in the execution of the algorithm. In particular, after recovery a site must receive the information needed to become reintegrated into subsequent execution of the algorithm.

Another issue that must be addressed concerns the ordering of events in a distributed system. While most systems guarantee that messages on the same link are received in the order in which they were send, this guarantee does not generalize to the entire network. It thus might happen that if two messages are each sent to a pair of sites, they could be received in different order at those sites, resulting in possibly inconsistent actions.
The basic concept underlying the distributed implementation of our algorithm will be to deliver each operation, in a globally fixed order, to all local serializers that need to know about that operation. To this effect we extend the transaction manager to include a fault-tolerant replica control protocol.

The replica control algorithm is developed in three stages: The first version assumes that information is completely replicated everywhere in the system, and consequently requires that all operations be broadcast to all sites in the system. The second version provides an optimization for read-only transactions and allows them to be serialized only locally. The third version will relax the assumption of complete information and distribute operations only to the subset of the sites where serialization decisions may be affected.

Our approach is motivated by the fault-tolerant process groups proposed in [5]. In this paper a number of broadcast primitives are developed that address most of the issues discussed above. These primitives however are geared towards support for atomic actions, and do not address the problem of maintaining data structures that have dynamically varying distribution in a fault prone environment. The same holds true for other atomic broadcast primitives proposed by Chang [13] and Christian et al. [14]. We thus use these primitives only as a starting point for our implementation and develop additional techniques as the need arises.

Two primitives, BCAST and FBCAST are used. The BCAST primitive is used to distribute operations to the serializers at the appropriate sites. It has the following properties:
BCAST(msg, dests)

(1) message delivery is atomic, i.e. either all operational sites in dests receive the message or none does.

(2) BCAST is totally ordered with respect to other BCAST's with overlapping destinations.

(3) BCAST preserves causality, i.e.

   (i) if msg is BCAST before msg' from the same site, then msg is received before msg' at all overlapping destinations.

   (ii) if msg is delivered at sender(msg') before msg' is BCAST, then msg is received before msg' at all overlapping destinations.

The FBCAST primitive is used to notify processes when a failure or recovery occurs, and is based on GBCAST in [5]. It has the following properties:

FBCAST

(1) FBCAST is totally ordered with respect to all other broadcasts

(2) It flushes all BCAST's from failed sites first and guarantees that no message is received from a site after it fails.

We will also use the concept of a coordinator-cohort algorithm from the ISIS system [6]. Coordinator-cohort algorithms are used to ensure completion of a distributed computation despite failures of some of the participants. Coordinator-cohort algorithms are derived from multi phase commit protocols [25, 48]: The coordinator initiates an action and notifies the cohorts when the action is finished. The cohorts either receive the "finished" message or an FBCAST notifying them that the coordinator failed. In this case one of the cohorts takes over and completes the action.
The order in which cohorts take over is determined statically according to site ids.

6.2.2.1. The Naive Algorithm

The naive version of the algorithm maintains a complete serialization snapshot at every site in the system. To do this, a serializer BCAST's every operation to all sites, including itself. After receiving a BCAST message with a new operation, the operation is inserted into the local snapshot as described in Chapter 5. The basic algorithm is shown in Figure 6.1. Recovery is addressed later in the chapter.

Since BCAST's are ordered with respect to each other, they are received in the same order everywhere. Thus, operations are inserted into the serialization snapshots in a globally fixed order and the snapshots that result will be identical at all sites in the system. Furthermore, since FBCAST's order BCAST's with respect to failures, either every site sees a BCAST or none of the sites does, hence site failures do not compromise correctness of the algorithm.

Let $S$ be the set of sites in the network.

**Notification Phase**

- BCAST(“INSERT, $o_{new}$”, $S$)

**Serialization Phase**

Upon receipt of a message “INSERT, $o_{new}$”,

- serialize according to the rules of the centralized algorithm.

Figure 6.1. The naive distributed algorithm
Use of the BCAST primitive, which involves a two-phase protocol, may seem expensive. However, the reader should keep in mind that BCAST is effectively generating a serialization order for previously unordered operations. This cost is consequently unavoidable. Moreover, several optimizations can be applied, and in the next two sections we will describe two optimizations regarding read-only operations and restrictions on the set of sites to which an operation is BCAST. Finally, the primitives can perform some optimizations, based on the use of piggybacking to overlap messages pertaining to one BCAST with those for another, and the algorithm itself could be implemented to send several operations in each BCAST message. Thus, cost should actually be reasonably low in any careful implementation of the approach we are advocating.

6.2.2.2. Read-only Operations

A number of replica control algorithms [54, 23] support the possibility of performing read operations locally. This strategy avoids broadcast operations for read-only transactions, but is very sensitive to failures. In case the site at which the read is performed fails, the information about the read is lost and subsequent operations could be serialized in an inconsistent manner. One solution is to abort all active operations with subgraphs that depend on a read operation at a site that failed. If aborts due to failures are to be avoided, read locks have to be verified at the time a transaction commits, and reacquired in case the site at which the lock was held failed.

Another solution, used for lock based concurrency control in ISIS, was described in [7] It uses a piggybacking scheme to ensure that read-locks are re
gistered of read locks in case of failures. This eliminates the need for aborts after failure, and was important in ISIS because the system attempts to guarantee roll-forward execution whenever possible. We believe that this technique can be applied to our graph based algorithm. However, the technique is complex and depends on other details of replicated data management in ISIS, which would be difficult for us to review here without a significant digression. Therefore, in the interest of brevity we omit a detailed description of this extension.

6.2.2.3. Fragment Based Scheme

In this section we describe an optimization based on the observation that a new operation need not be broadcast to sites executing only operations whose serialization order cannot be affected by the new operation. The idea is then to divide the execution snapshot into disjoint fragments, where each fragment contains the execution snapshots for a subset of the objects in the system. A new operation on object \( O \) is then distributed only to the sites at which the appropriate fragment is located.

Fragments are dynamic entities, i.e. they change as new operations are inserted and as operations commit. The problem to be solved is then to maintain dynamically growing and shrinking fragments across a set of sites and to ensure that the growing and shrinking of the fragments is properly synchronized with insertions and deletions of new operations. We note that the technique is one that was not considered by Birman and Joseph [5] and which could be applied to the management of other sorts of dynamic data structures in addition to the fragment management discussed here.
Consider for example the distribution of objects and operations depicted in Figure 6.2.

**Example 6.1:**

Objects $o_1$, $o_2$, and $o_3$ are located at sites $s_1$, $s_2$, and $s_3$ respectively. Every operation updating an object must be known at least at the set of sites at which the object is located. Thus, any operation updating $o_1$, denoted $upd(o_1)$, must be known at site $s_1$, and $upd(o_2, o_3)$ which updates $o_1$ and $o_3$ must be known at sites $s_2$ and $s_3$. Consequently, the fragment containing $o_1$ must be located at $s_1$, and the fragment containing $o_2$ and $o_3$ at $s_2$ and $s_3$. Any new operation $upd(o_2)$ must be broadcast to all sites where the fragment of $o_2$ is located, i.e. $s_2$ and $s_3$. Before an operation updating $o_1$ and $o_2$ can be broadcast, the fragments of $o_1$ and $o_2$ must be combined and the new fragment distributed to sites $s_1$, $s_2$, and $s_3$. □

---

<table>
<thead>
<tr>
<th>$s_1$</th>
<th>$s_2$</th>
<th>$s_3$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$o_1$</td>
<td>$o_2$</td>
<td>$o_3$</td>
</tr>
<tr>
<td>$upd(1,2)$</td>
<td>$upd(1,2)$</td>
<td>$upd(3)$</td>
</tr>
</tbody>
</table>

**merge fragments**

**distribute new fragment**

<table>
<thead>
<tr>
<th>$s_1$</th>
<th>$s_2$</th>
<th>$s_3$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$upd(2,3)$</td>
<td>$upd(2,3)$</td>
<td>$upd(2,3)$</td>
</tr>
</tbody>
</table>

**Figure 6.2. Fragments**
Before we describe the complete algorithm we need to introduce some technical definitions. We denote by \( \text{sites}(e) \) the set of sites at which entity \( e \) resides. Here, \( e \) can either be an object or an execution snapshot. \( \text{Objects}(op) \) is the set of objects accessed by operations \( op \). \( \text{Graphs}(O) \) denotes the execution snapshots containing operations on object \( O \), and \( \text{loc}(O) \) denotes the set of sites where snapshots related to object \( O \) are located, i.e. \( \text{sites}(\text{graphs}(O)) \).

The basic idea is to ensure that new operations are inserted atomically into the graphs at the full set of affected sites. We do this using a two-level algorithm. The front end dispatches new operations using an iterative broadcast; it simply BCAST’s an operation to a set of sites until the destination sites accept and perform the operation, which is detected by checking after the BCAST returns to see if the desired effect occurred. Use of BCAST ensures atomicity. The role of the back end is to receive each operation and perform it, discarding any operations that the dispatcher sent to the wrong set of sites.

Operations could be sent to the wrong set of sites for two reasons: Firstly, fragments are changing dynamically as operations are inserted from several sites. By the time a message is received by the sites holding a particular fragment, objects might have been added to or deleted from the fragment. We say that a fragment merge occurs if a new operation is submitted that accesses objects in two different fragments. A fragment split occurs once the last operation accessing two disjoint sets of objects in a fragment commits. A fragment insert occurs if an operation is submitted that touches only objects in one fragment. Secondly, a failure or recovery may have occurred. We model recovery as a type of fragment merge; the same
algorithm as is used for merge is used to transfer the state of the fragment to a recovering site (discussed below). Failure, on the other hand, is ignored: the failed site is deleted from the fragment, but sites do not reject operations that were "sent" to a failed site, nor is the a new fragment formed to exclude the failed site.

To detect changes in fragments, every site \( s \) maintains a view_id for every object \( o \) and every fragment. The view_id for \( o \) which fragment \( s \) thinks \( o \) belongs to. The view_id of a fragment is updated whenever the fragment splits or merges with another fragment. It is broadcast with every message, allowing the receiver of the message to detect whether the message was based on the correct state of the system. A message can then be rejected if its view_id does not correspond to the current fragment.

The minimum set of sites at which a fragment containing an operation on object \( O \) must be known are the sites at which \( O \) is located. Consequently, before any new operation \( op \) can be inserted into a graph, the relation

\[
sites(O) \subseteq sites(graphs(O))
\]

must be established for all objects \( O \) touched by \( op \). The algorithm maintains a graph such that the above relation is true for all objects in the graph. In other words, the graph merges the execution snapshots of all objects in the transitive closure of the above relation. The graph that results resides at the set of sites \( S \) given by:

\[
S = \bigcup_{objects(G)} sites(graphs(O))
\]

We now describe the fragment management algorithm in detail. The front end is depicted in Figure 6.3., the back end in Figure 6.4.. The front end is responsible
for broadcasting insert requests to its "best guess" of the set of destination sites and for initiating fragment merges if necessary. Specifically, the front end checks to see that the fragment resides at a sufficient set of sites; if not, it BCAST's a merge request "merge, mustinclude, doesinclude" to all sites in loc(o) for objects o accessed by op. Then, it BCAST's the op to all sites holding the new fragment. In both cases, it loops until the desired effect has occurred. Implicit in this is an assumption that BCAST does not return to its caller until the BCAST-ed operation has been executed (or discarded).

Let s be the site at which a new operation op is submitted, and let

\[ S(op) = \bigcup_{o \in \text{objects}(op)} \text{sites}(o) \]

be the set of sites at which objects(op) are located. Let

\[ \text{LOC}(op) = \bigcup_{o \in \text{objects}(op)} \text{loc}(o) \]

denote the sites at which objects touched by op are located, and let \( I(op) = \bigcap_{o \in \text{objects}(op)} \text{loc}(o) \) be the set of those sites that host all objects touched by op.

**Front End:**

```
loop
{
  (6.1) while( S \subseteq I(op))
    {
      /* merge fragments */
      BCAST("merge, S, I(op), view_id [o_1, ..., o_n]", LOC(op))
    }
  (6.2) BCAST("insert, op, view_id [o_1, ..., o_n]", LOC(op))
}
```

until insert succeeds.

Figure 6.3. Front End
The back end handles fragment merge requests and insertion of operations into existing fragments. Once it receives a merge request it temporarily stops processing insert requests. One of the receivers of the request assumes the role of coordinator for the fragment merge phase\(^1\) and requests the current fragments from all sites in mustinclude, i.e. \( \bigcup_{o \in \text{objects(op)}} \text{loc}(o) \). It then combines the fragments into a new fragment and updates \( \text{loc} \) such that \( \text{loc}(o_i) = \text{loc}(o_j) = \bigcup_{o_k \in \text{objects(op)}} \text{loc}(o_k) \). If any site returns a wrong vector of view_ids, the merge is aborted. Otherwise, the new fragment is broadcast back to the participants in the merge. These again check the view_id and install the new graph if ids are correct. Then the view_id is incremented for each object in the new fragment. Should the coordinator fail, one of the sites that remained operational takes over on its behalf and re-runs the merge algorithm.

We now argue that the algorithm correctly inserts operations into the right set of fragments. We first show that the fragment merge algorithm produces the right fragment and then show that operations are inserted into the correct fragment.

The fragment merge part (6.1) does not terminate until the correct fragment is known at all sites at which objects accessed by the new operation reside. It only terminates once all all sites at which a fragment must be known have the right fragment. The coordinator cohort scheme makes the algorithm fault-tolerant; a new coordinator takes over in case the old coordinator fails. The back end ensures that the \( \text{loc}(o) \) tables are updated, and thus the fragment merge loop terminates eventu-

\(^1\) We assume a unique numbering of sites, and the site with the lowest site id will be coordinator, assuring that the coordinator is unique.
Back End:
loop forever
{
    receive message
    case (message)
    {
        "merge, mustinclude, doesn't include":
            • stop processing incoming BCAST's of operations
            • initiate fragment update
        "new graph":
            if view_id(graph)=view_id [o_1, ⋯ ,o_n]
                • install new fragment and increment view_id
        "insert, op, view_id":
            if view_id = view_id [o_1, ⋯ ,o_n]
                • serialize op according to the centralized rules
            else
                • discard message
    }
}

Fragment Update:
Coordinator:
• get graphs from all sites in loc("mustinclude")
• for each object, update loc(o)
if any wrong view_id is returned
    • abort
else
    • merge graphs into new graph G
    • BCAST("new graph, G, view_id [o_1, ⋯ ,o_n]", loc("mustinclude"))

Cohorts:
Upon receipt of a message "get graph"
    • respond with local graph and view_id

Figure 6.4. Fragment based algorithm
ally. Moreover, merging is idempotent, hence failures that cause the merge algorithm to be restarted pose no problems. Finally, since BCAST and FBCAST are atomic, and BCAST is totally ordered relative to FBCAST, all sites observe the same sequence of events throughout the execution of the algorithm.

The "insert" BCAST succeeds only if the view_ids of the sender and receiver match, ensuring that the operation is inserted into the right fragment. The view_id also ensures that other merges do not cause operations to be inserted into a wrong fragment. In case the fragment changes between the merge phase and the insert BCAST, the insert fails and the fragment merge is repeated.

6.3. Recovery

To recover, a site issues an FBCAST to the operational sites in the system. An algorithm for doing this is given in [5]; it ensures that even if a site incorrectly guesses at the list of operational sites, eventually the FBCAST is delivered atomically to the entire set of such sites. Once the FBCAST is received, a simple coordinator-cohort algorithm is used to transfer the fragments that should reside at the recovered site to it; this is done just as is the merge algorithm that was described above. As in the case of the merge algorithm, the total ordering of FBCAST relative to BCAST makes it easy to establish that such a transfer can be done correctly. All sites then update the view-id's for fragments that were transferred to the recovering site and execution resumes normally.
6.4. Summary

This chapter introduced resilient objects as a design tool for fault-tolerant systems. Resilient objects are an extension of atomic objects. On top of data abstraction and concurrency control they provide a built in replica control mechanism and present the user with a single-site abstraction of an object.

We showed how to implement our concurrency control algorithm for resilient objects, using a small set of broadcast primitives from [5]. These primitives enabled us to translate the concurrency control methods presented earlier into a distributed setting. In connection with the translation we presented an algorithm to minimize the number of sites to which new operations have to be transmitted for serialization. The algorithm uses an iterative technique which should be applicable to other dynamic data structures in addition to the concurrency control graphs for which we used it here. Correctness is easily established because the primitives are atomic and totally ordered and failures are treated as a special sort of communication event. Although it is risky to speculate on the probable cost of a distributed algorithm as complicated as the one we have proposed here, we believe that the cost of the approach will be low in most normal execution scenarios.
CHAPTER 7

Optimizing Concurrency Control

In this chapter we investigate techniques to reduce the number of concurrency control interactions necessary to serialize an operation, thus offsetting some of the cost introduced by the broadcast primitives in the previous chapter. We suggest techniques to completely avoid concurrency control interactions when possible, and to replace global interactions by local concurrency control interactions when some concurrency control is needed, but global interactions can be avoided.

Our approach is again geared towards object oriented systems where the structure of operations is known in advance from object specifications. The optimization techniques are based on defining a *scope of concurrency* for each operation and determining whether previous concurrency control decisions cover an operation within its scope.

In the first method, coverage is determined dynamically from conflict predicates of active operations in the local concurrency control snapshot. This technique is particularly applicable to interactive environments where transactions are invoked operation by operation (e.g. as a user types commands at a terminal). Instead of broadcasting every operation as it is submitted we use conflict predicates to identify conditions under which operations can be serialized locally without compromising consistency of data objects.

Our second technique applies to a batch oriented environment where transactions are submitted as a whole. It uses data flow information collected by the object
compiler to determine coverage information at compile time. The basic idea here is to represent every operation as a control flow graph and use the graph together with an extension of conflict predicates to paths in the graph to identify covered suboperations.

The chapter is organized as follows. In Section 1 we define the notion of coverage and show how to change the serializers to take advantage of dynamic coverage information. Section 2 is devoted to static analysis techniques that allow to determine coverage information at compile time. Section 3 summarizes the chapter.

7.1. Detecting Coverage Dynamically

In this section we introduce the concept of coverage and show how to modify the serializer module to use coverage information to optimize concurrency control interactions.

Coverage is defined on the tree structure of a nested operation. Intuitively, an operation \( t_i \) is covered if its parent conflicts with all operations \( t_i \) conflicts with. Then, once the parent operation is active, all conflicting operations outside of the scope of the parent are excluded from executing concurrently.

Formally, let \( OP = \{ o_1, \cdots, o_n \} \) be a transaction template with root \( OP \) and suboperations \( o_1, \cdots, o_n \). Coverage for a suboperation \( o_i \) is defined as follows.

**Definition 7.1:** Operation \( o_i \) is **covered** by \( OP \), if

\[
\vee_{op} \; CONF [o_p, op] \Rightarrow \vee_{op} \; CONF [OP, op].
\]

Intuitively, an operation \( o_i \) is covered if its parent conflicts with all operations \( op \) that conflict with \( o_i \). Hence, if the parent operation has been scheduled for execu-
tion, no operation conflicting with \( o_i \) can be active, and concurrency control for \( o_i \) can be avoided.

The relatively high cost of communication leads us to assume that if possible, all subtransactions of a particular operation will be executed at a single site. In systems in which this assumption holds concurrency control for covered operations can avoid broadcasts and limit conflict detection to the local site. If operation \( o_i \) is covered and has no concurrent siblings, no concurrency control is necessary at all. We say that in this case \( o_i \) is globally covered. Alternatively \( o_i \) is partially covered if it is covered by its parent but has siblings that might be executed concurrently. In this case, a local concurrency control interaction is necessary to serialize the concurrent siblings.

The serializer module can now take advantage of coverage information. The modified serializer is shown in Figure 7.1. It is assumed that at compile time each operation is tagged with a flag indicating whether there are potentially concurrent siblings. Operations that are not covered need to be serialized globally as before; partially covered operations can be handled through local concurrency control, and for globally covered operations no concurrency control is necessary.

The following argument shows that the algorithm still guarantees serializability. An operation can be active only if there are no conflicting operations executing concurrently. Furthermore, operations are active only if their parent operation is active. Consequently, for a covered operation all conflicting operations outside the scope of its parent are excluded once its parent is active. Thus it is sufficient to interact with the local concurrency control unit to ensure correct serialization of concurrent
Let $S$ be the current snapshot, and let $OP = \{o_1, \ldots, o_n\}$ be the transaction template of a new operation to be serialized.

Phase 1:

For each $o_i$, bottom up, identify the set $C_i$ of conflicting operations as follows:

(i) if $o_i$ is globally covered
   - $C_i = \emptyset$

(ii) if $o_i$ is partially covered
    - $C_i = \{o_j \in \text{siblings}(o_i) \mid C(o_i, o_j) = true\}$

(iii) otherwise
    - $C_i = \{o_j \in S \cup OP \mid C(o_i, o_j) = true\}$

Push the pair $(C_i, o_i)$ onto the conflict stack.

Figure 7.1. Dynamic Coverage Algorithm

siblings. By definition globally covered operations do not have any concurrent siblings, hence no concurrency control is necessary.

Determining coverage at runtime might involve a considerable amount of computation, which makes it doubtful that the above technique results in significant savings in a very wide range of systems. However, our experience with applying similar techniques in the ISIS system suggests that in systems with replicated data substantial performance improvements are possible. For example, the ISIS concurrency control algorithm is based on Moss' locking scheme for nested transactions. It uses a variant of our technique to avoid global lock acquisition: Before a write lock request is broadcast to all copies of the data item, the serializer inspects the local lock tables to find out whether the lock is already held by an ancestor of the operation. If this is the case, the operation is globally covered, since all other
operations requesting the lock will be blocked until it is released by the parent. Consequently the lock needs to be acquired only locally, saving the cost of the broadcast operation, and more importantly, reducing latency incurred while waiting for the lock request to be granted. The incorporation of coverage related techniques into the concurrency control algorithm is expected to result in considerable performance improvements.

7.2. Determining Coverage Statically

The complexity of computing coverage prevents the application of dynamic coverage detection in all but a limited number of cases. However, it is possible to collect coverage information through compile time analysis of objects. Since this analysis is performed only once its complexity can be amortized over the lifetime of the system and the increase in runtime performance may well justify the increase in compile time cost. In this section we sketch out this approach and show how to use data flow analysis techniques to determine coverage of operations statically.

For the purpose of data flow analysis, programs are represented by control flow graphs [45]. A control flow graph consists of a collection of basic blocks, sequences of straight line code, connected by directed edges introduced by control statements. To illustrate the technique we consider two types of control statements, alternative (if-then-else) and repetition (while), which results in the following definition for control flow graphs.

**Definition 7.2:** For a graph \( G \), \( \text{ENTRY}_G \) (\( \text{EXIT}_G \)) denotes the set of vertices \( v \in G \) such that there are no edges entering (leaving) \( v \). The control flow graph \( G \) for an operation \( OP \) is then defined recursively as follows. If \( OP \) is
(1) a basic block $s_i$,
then the graph consists of a single vertex $s_i$.

(2) formed by alternative if $\text{cond then } s_1 \text{ else } s_2$,
then the graph consists of the graphs for $s_1$ and $s_2$ with two additional nodes $s$ and $s'$ and edges from $s$ to all nodes in $ENTRY_{s_1} \cup ENTRY_{s_2}$, and from all nodes in $EXIT_{s_1} \cup EXIT_{s_2}$ to $s'$.

(3) formed by repetition while $\text{cond do } s_1 \text{ od}$,
then the graph consists of the graph for $s_1$ and two new nodes $s$ and $s'$ and edges from $s$ to $s'$, from $s$ to all nodes in $ENTRY_{s_1}$, and from all vertices in $EXIT_{s_1}$ to $s$.

Figure 7.2 shows control flow graphs for the if-then-else and while statements.

Each control flow graph defines a set of control flow paths. Intuitively, a control flow path represents one particular execution of the operation represented by

---

**basic block $s_1$:**

$$s_1$$

**alternative statement:**

![Diagram of alternative statement]

**repetition:**

![Diagram of repetition]

---

Figure 7.2. Control Flow Graphs
the graph. In particular, for each operation $p_i$ in a control flow path the set of operations executed before $p_i$ is known. We will use this knowledge to compute coverage information.

Formally, a control flow path in graph $G$ is a sequence of operations $P = \{p_1, \ldots, p_n\}$ such that $p_1 \in ENTRY_G$ and $p_n \in EXIT_G$, and there exists an edge $(p_i, p_{i+1})$ for $i = 1, \ldots, n-1$. We can now extend the definition of coverage to basic blocks, paths and graphs:

**Definition 7.3:** Let $B = \{b_1, \ldots, b_n\}$ be a basic block. An operation $b_i \in B$ is partially covered in $B$, if

$$\forall \text{ op, } C(b_i, \text{ op}) \Rightarrow \bigvee_{b_j \preceq b_i} C(b_j, \text{ op})$$

Intuitively $b_i$ is covered in $B$, if the operations preceding $b_i$ conflict with every operation $b_j$ could potentially conflict with.

**Example 7.1:** Consider the following operation $O$.

$$O(x)$$

{  
BEGIN write_1(x) END;  
BEGIN read_2(y); write_3(y) END;  
BEGIN write_4(x) END
}

Here suboperation write_4(x) is covered, since $C_{\text{write}_4, \text{op}} \Rightarrow C_{\text{write}_1, \text{op}}$ for all operations op. However, write_3(y) is not covered, since neither write_1 nor read_2 conflict with all operations write_3 conflicts with. □

An operation op is covered in graph $G$, if it is covered on all paths $P = \{p_1, \ldots, p_k\}$, where $p_1 \in ENTRY_G$ and $p_k = \text{op}$. For acyclic program graphs,
the set of paths is finite and coverage can be determined for each path as before.

For cyclic program graphs, the set of paths is potentially infinite. However, we can eliminate all paths containing operations in a loop body when determining coverage for operations outside of loops. This is because there is always the possibility that a while loop will not be executed at all, in which case there are no concurrency control interactions in the body that could cover other operations. Hence the coverage of subsequent operations is not affected by operations in loop bodies.

To determine coverage for operations within the loop, it is sufficient to unroll the loop once. The modification of the data flow graph for while loops in Figure 7.3 reflects this idea.

Intuitively, the modification equates the while loop with an alternative statement with one empty branch. Thus, there is always a path that does not contain the loop body, reflecting the fact that the body might not be executed, and for operations in the loop body coverage is determined the same way as for operations in the body of alternative statements.

Figure 7.3. Modified Flow Graph for while
7.3. Summary

In this chapter we suggested techniques to restrict or eliminate concurrency control actions. We introduced the notion of operations being covered by previous concurrency control decisions. Coverage can be determined dynamically from the conflict predicates of operations. The applicability of the technique however is limited by its cost. Coverage can also be determined statically at compile time using data flow analysis techniques.

Although neither technique is simple enough for general use, both are valuable in a limited number of situations where they might be applied by hand, or as in ISIS, where the concurrency control interactions narrowly defined and hence it is relatively easy to determine dynamic coverage.
CHAPTER 8

Summary

The main result of this work has been to show that libraries of objects can be effectively used to implement fault-tolerant distributed systems. We demonstrated this by devising a method to overcome some of the more glaring inefficiencies that arise when object libraries are used in a naive straightforward way. We first defined a model of execution for operations on resilient objects and extended the traditional database notion of conflicting operations and serializability. We then introduced a new concurrency control algorithm that is based on this new definition of conflict and used the model of execution to prove the correctness of the algorithm. We extended the algorithm to allow a user to incorporate semantic information into the concurrency control process by means of conflict predicates, which results in increased concurrency over traditional methods. We showed the feasibility of the algorithm in distributed systems by designing a fault-tolerant distributed implementation based on a few well defined atomic broadcast primitives. In addition, we showed how to use data flow analysis techniques to reduce communication overhead introduced by concurrency control in a distributed system.

Our work shows that the object oriented approach to designing fault-tolerant distributed systems is feasible if used in conjunction with techniques that exploit maximum concurrency in such a system and with techniques that reduce communication overhead whenever possible.
8.1. Future Work

There are several ways in which our work can be extended. The logically next step is to implement the full algorithm and experimentally verify its feasibility.

The most popular concurrency control techniques are based on locking data items to prevent conflicting operations from accessing the item concurrently. The problem with this approach is that it does not keep a history of accesses with an object and allows access only on a strictly first-come, first-served basis. The question of how to associate partial history information with locks and then utilize this information to increase concurrency is one area for future research.

Another such area is the further exploration of data flow analysis techniques to reduce concurrency control overhead.

Another question worth further exploration is the applicability of the implementation technique for our concurrency control algorithm in a distributed system to other problems. One example for such a problem is finding a minimum cost spanning tree in a distributed system.
REFERENCES


47. R. Schlichting and F. B. Schneider, "Fail Stop Processors: An Approach to Designing Fault-Tolerant Distributed Systems," *ACM Transactions on Com-


    2nd corrected edition
