1994

Scheduling in a generalized transaction/thread model

Bernhard G. Humm
University of Wollongong

Scheduling in a Generalized Transaction/Thread Model

A thesis submitted in fulfilment of the
requirements for the award of the degree of

Doctor of Philosophy
(Computer Science)

from

THE UNIVERSITY OF WOLLONGONG

by

Bernhard G. Humm, Dipl.-Inform. (Kaiserslautern)

Department of Computer Science
1994

Declaration

I hereby declare that I am the sole author of this thesis. I also declare that the material presented within is my own work, except where duly acknowledged, and that I am not aware of any similar work either prior to this thesis or currently being pursued.

Bernhard G. Humm

Abstract

This thesis is about scheduling in object-oriented distributed systems that support nested transactions. Novel linguistic constructs are introduced that allow the specification of transaction and thread semantics over messages independently. This so-called "generalized message scheme" provides a richer set of useful programming abstractions than does the traditional nested transaction model. For this reason, the scheduling semantics of the traditional nested transaction model are extended to cover all abstractions provided by the generalized message scheme. An implementation-independent scheduling mechanism is presented that satisfies these scheduling semantics. Also, an efficient implementation of this scheduling mechanism is described.

The mechanisms presented in this thesis have a number of advantages over existing approaches. Separation of transaction and thread semantics achieve more flexibility during system development and more efficiency during system execution. Typical features of object-orientation like reusability, extendibility and maintainability are supported. Programmers can fine-tune the performance of their applications without having to change the structure or semantics of the code. It is shown that the proposed mechanisms, though more general than traditional mechanisms, can be implemented as efficiently as traditional mechanisms.

Acknowledgements

I would like to thank a number of people who have been helpful throughout the years that I have worked on this thesis. To Greg Doherty I owe the fact that I ended up in Wollongong. The Telecommunications Software Research Centre (TSRC) has been an excellent work environment and I am grateful to the director Fergus O'Brien for employing me as the first member of TSRC. Thanks go to the Australian Government, the University of Wollongong and Telecom Australia for financial support. Fergus has also been helpful as a thesis supervisor, together with Neil A. B. Gray. Neil's fundamental knowledge in object-oriented concepts was an important source.

Special thanks go to my co-workers at TSRC, Michael Fazzolare and R. David Ranson. Through many long days and nights of discussions we acquired together the level of expertise in the field which we have now. This thesis would certainly not be the same if I hadn't worked in this great group.

I thank Marilyn Cross and all the above mentioned persons for commenting on a draft version of this thesis. Finally, I must thank my wife Anke. She has always reminded me not to neglect my family even more than I already have. Thank you for putting up with all this!

Contents

List of Figures viii

List of Tables ix

1 Introduction 1

2 Transactions and Objects in Distributed Systems 3
- 2.1 Issues in Distributed Systems . . . 3
- 2.2 Transactions . . . 5
  - 2.2.1 Concurrency Control . . . 6
  - 2.2.2 Recovery . . . 9
- 2.3 Nested Transactions . . . 11
- 2.4 Object-Orientation in Distributed Systems . . . 13
  - 2.4.1 The Main Concepts of Object-Orientation . . . 13
  - 2.4.2 Advantages of Object-Orientation . . . 16
- 2.5 Distributed Systems Supporting Nested Transactions and Objects . . . 18

3 The Hermes/ST Distributed Programming Environment 20
- 3.1 The Distributed Bank Example . . . 20
- 3.2 The Hermes/ST Object Model . . . 21
  - 3.2.1 Development Advantages . . . 23
- 3.3 The Generalized Message Scheme . . . 23
  - 3.3.1 Message Kind and Transaction Parameters . . . 23
  - 3.3.2 Specification of Message Parameters . . . 24
  - 3.3.3 The Weighted Voting Example . . . 25
  - 3.3.4 Additional Message Parameters . . . 25
  - 3.3.5 Specifying Invocation Parameters in Method Interfaces . . . 26
  - 3.3.6 Development Advantages . . . 27
- 3.4 Concurrency Control . . . 28
  - 3.4.1 Implicit Concurrency Control . . . 28
  - 3.4.2 Explicit Concurrency Control . . . 29
  - 3.4.3 Development Advantages . . . 31
- 3.5 Hermes/ST Implementation of the Distributed Bank . . . 32
- 3.6 Evaluation and Comparison to Other Approaches . . . 33
  - 3.6.1 Evaluation . . . 33
  - 3.6.2 Argus . . . 34
  - 3.6.3 Avalon/C++ . . . 35
  - 3.6.4 Venari/ML . . . 36

4 Scheduling in a Generalized Transaction/Thread Model 37
4.1 Definitions . . . 37
4.1.1 Messages and Message Trees . . . 37
4.1.2 Relationships Between Messages . . . 39
4.1.3 Message Paths and Message Path Elements . . . 40
4.1.4 Regular Expressions for Message Paths . . . 40
4.1.5 Transactions . . . 41
4.1.6 Threads . . . 42
4.1.7 Partial Threads Under Transactions . . . 43
4.1.8 Schedules . . . 44
4.1.9 Cascading Aborts . . . 47
4.1.10 Return Dependencies . . . 47
4.2 The Scheduling Properties . . . 48
4.2.1 Examples for the Scheduling Properties . . . 49
4.2.2 Discussion of the Scheduling Properties . . . 51
4.3 The Schedulability Predicate . . . 53
4.4 Correctness of the Schedulability Predicate . . . 54
4.4.1 Return Dependencies . . . 54
4.4.2 Cascading Aborts . . . 57
4.4.3 The Partition of Cases . . . 58
4.4.4 $s_1 = s_2$ . . . 60
4.4.5 m_1 *retDep* m_2 . . . 62
4.4.6 $s_1 \neq s_2$ and not m_1 *retDep* m_2 . . . 63
4.4.7 m_1 and m_2 Both Non-Transactional . . . 64
4.4.8 m_1 Transactional and m_2 Non-Transactional . . . 64
4.4.9 m_1 Non-Transactional and m_2 Transactional . . . 66
4.4.10 m_1 and m_2 Both Transactional . . . 66
4.4.11 $tl_1 \neq tl_2$ . . . 66
4.4.12 $tl_1 = tl_2$ . . . 68
4.4.13 $t_1 = t_2$ . . . 68
4.4.14 $t_1 < t_2$ . . . 69
4.4.15 $t_1 > t_2$ . . . 71
4.4.16 $t_1 <> t_2$ . . . 72
4.5 Implementation of the Scheduling Mechanism . . . 74
4.5.1 System Objects for Scheduling . . . 75
4.5.2 Interaction of System Objects . . . 77
4.6 Implementation of the Schedulability Predicate . . . 82
4.6.1 The Algorithms . . . 83
4.6.2 Correctness of the Schedulability Algorithm . . . 84
4.7 The Wait-By-Necessity Extension . . . 92
4.7.1 Scheduling and Return Dependencies . . . 92
4.7.2 A General Form of Wait-By-Necessity . . . 94
4.7.3 A Less General Form of Wait-By-Necessity . . . 96
4.8 The Non-Serialized Transactional Thread Extension . . . 97
4.9 The Top-Level Extension . . . 98

5 Discussion **99**
5.1 Moss' Model . . . 100
5.1.1 Transactions . . . 100
5.1.2 Scheduling . . . 100
5.1.3 Comparison . . . 102
5.2 Argus . . . 107
5.2.1 The Model . . . 107
5.2.2 Generality of the Model . . . 108
5.2.3 Scheduling . . . 108
5.2.4 Serializability of Ancestor and Descendant Transactions . . . 108
5.2.5 Level of Concurrency . . . 109
5.3 Eden . . . 111
5.3.1 The Model . . . 111
5.3.2 Scheduling . . . 112
5.3.3 Comparison . . . 113
5.4 Downward Lock Inheritance . . . 113
5.4.1 Simple Downward Lock Inheritance . . . 113
5.4.2 Controlled Downward Lock Inheritance . . . 113
5.4.3 Analysis . . . 114
5.4.4 Comparison . . . 114
5.5 Venari/ML . . . 115
5.5.1 Generality of the Model . . . 116
5.5.2 Scheduling . . . 116
5.5.3 Serializability of Ancestor and Descendant Transactions . . . 116
5.5.4 Level of Concurrency . . . 117
5.6 KAROS . . . 117
5.6.1 The Transaction Model . . . 117
5.6.2 Scheduling . . . 118
5.6.3 Serializability . . . 118
5.6.4 Efficiency and Concurrency . . . 118
5.7 Performance Analysis . . . 119
5.7.1 Modifying Message Parameters . . . 120
5.7.2 Performance of Schedulability Testing . . . 121
5.7.3 Schedulability Testing versus Overall Transaction Cost . . . 121
5.7.4 Caching versus Asking Scheduling Information . . . 122

6 Conclusions **123**

A Hermes/ST Code Examples **125**
A.1 The Binary Search Tree . . . 125
A.1.1 The `Tree` Class . . . 125
A.1.2 The `TreeNode` Class . . . 127
A.2 Weighted Voting for Replicated Objects . . . 128
A.2.1 Methods for Concurrent Collection Enumeration . . . 128
A.2.2 The `ReplicatedObject` Class . . . 129
A.2.3 The `ReplicaInfo` Class . . . 131
A.3 Specification and Overriding of Message Parameters . . . 132
A.3.1 Transfer Method in Class `Teller` . . . 132
A.3.2 Deposit And Withdraw Methods in Class `Branch` . . . 132
A.3.3 Transfer Method in Class `AutomaticTellerMachine` . . . 133

A.4 Programmable Lock Definition and Usage 133
A.4.1 The `ProgrammableLock` Class 133
A.4.2 The `AccountWriteLock` Class 134
A.4.3 Deposit Method of Class `Branch` 135
A.4.4 The `SavingsAccountsWriteLock` Class 135
A.4.5 Method `addInterest` in Class `Branch` 136
A.5 The Distributed Bank Implementation 136
A.5.1 The `Teller` Class 136
A.5.2 The `HeadOffice` Class 137
A.5.3 The `AutomaticTellerMachine` Class 138
A.5.4 The `BankClerk` Class 139
A.5.5 The `Branch` Class 139
A.5.6 The `Account` Class 141

Bibliography **143**

Index **149**

List of Figures

2.1 A single inheritance hierarchy for polygon classes 15

3.1 An example binary search tree. 29

4.1 An example message tree. 38
4.2 Messages belonging to thread S_1. 42
4.3 Messages belonging to partial thread S_8. 43
4.4 S_1/T_8: S_1 enters T_8. 44
4.5 S_{10}/T_8. S_{10} is created within T_8. 45
4.6 Synchronization of messages. 46
4.7 Synchronization of messages and transactions. 47
4.8 Return dependency between messages M_1 and M_3. 48
4.9 Avoidance of cascading aborts. 57
4.10 Partition of cases. 58
4.11 $s_1 = s_2$, $m_1 <> m_2$. 60
4.12 $s_1 = s_2$. 62
4.13 m_1 *retDep* m_2. 63
4.14 One of the two messages is transactional, the other one is non-transactional. 65
4.15 $tl_1 \neq tl_2$. 67
4.16 $t_1 = t_2$. 68
4.17 $t_1 < t_2$. 70
4.18 $t_1 > t_2$. 71
4.19 $t_1 <> t_2$. 73
4.20 Scenario of sending and executing a message. 78
4.21 Return dependency with long and short message paths. 85
4.22 Phases of the schedulability algorithm. 87
4.23 $m_1 \prec m_2$. 92
4.24 A scenario of wait-by-necessity messages. 95

5.1 Accesses are turned into subtransactions. 105
5.2 Ancestor/descendant synchronization in Argus. 108
5.3 Linked list example. 109
5.4 Lock downward inheritance. 114
5.5 Scheduling in KAROS. 119

List of Tables

2.1 Example for two interleaving deposit operations. 4
2.2 Example for cascading aborts. 7
2.3 Lock compatibility matrix for read/write locking. 8
2.4 Example for deadlock. 9

5.1 Transactional bank transfer with varying message parameters for deposit and withdraw. 120
5.2 Comparison of the performance of schedulability testing. 121
5.3 Cost for schedulability testing in comparison to overall transaction costs. . 122
5.4 Obtaining scheduling information remotely and locally. 122

Chapter 1

Introduction

This thesis is about scheduling in object-oriented distributed systems that support nested transactions. Within the last decade, *distributed systems* have become increasingly important. Programming distributed systems is inherently more complex than programming single-node, sequential systems. It is therefore a goal of distributed systems research to investigate linguistic mechanisms that allow the construction of reliable and efficient distributed systems in a convenient and cost-effective manner. One convenient abstraction for reliable computing is that of *(nested) transactions* (e.g. [GR93]). Transactions were originally developed in the database area and have since been successfully applied to distributed systems.

Object-orientation (e.g. [Mey88]) is a programming paradigm that was originally developed in the simulation area. Its advantages in terms of rapid prototyping, reusability, extensibility, and maintainability of systems have been widely acknowledged. Today, object-orientation is used by many computing communities, including the distributed systems community.

Both technologies, object-orientation and nested transactions, have been integrated with distributed systems. This research started in the early eighties with the *Argus* project [Lis82] (Massachusetts Institute of Technology). The Argus project demonstrated successfully that both technologies make the development of distributed systems easier. However, Argus has performance drawbacks. Later projects, like *Camelot/Avalon* [EME91] (Carnegie Mellon University), were able to overcome these shortcomings. Today, research into this technology has matured enough so that it has been adopted in large-scale commercial products. Many such systems are currently available and their number is increasing rapidly.

All existing object-oriented transactional distributed systems, research prototypes and commercial systems, provide only a restricted model for concurrency in nested transactions. This thesis argues that a *generalized transaction/thread*[1] *model* allows higher concurrency, a more natural and convenient programming abstraction and other development advantages which are typical for object-orientation: reusability, extensibility, and maintainability. This generalized model can be implemented as efficiently as the traditional, restricted model. The novel aspects of this thesis can be summarized as follows.

- New *linguistic constructs* are presented that unify transaction creation and thread creation with messages. With this so-called "*generalized message scheme*", synchronous and asynchronous messages can be specified that do or do not create transactions. This scheme allows non-transactional threads, top-level transactions,

[1] A *thread of control* or simply *thread* is a unit of concurrent computation.

synchronous and asynchronous subtransactions and transactional threads that do not create a subtransaction.

- New *scheduling properties* are defined for the generalized message scheme. They represent a natural and useful extension of existing nested transactions scheduling semantics. Serializable threads are distinguished from threads that are not serializable due to so-called "*return dependencies*".
- An implementation-independent *scheduling mechanism* is described which satisfies the scheduling properties.
- An algorithm is presented which implements the scheduling mechanism. It is shown that the algorithm is no more expensive than traditional implementations of the restricted transaction model.

The remainder of this thesis is structured as follows. Chapter 2 gives an overview of issues in transactional object-oriented distributed systems. Chapter 3 presents the generalized message scheme and other linguistic constructs of *Hermes/ST* [FHR94, Faz94, Ran94, Hum93, FHR93c, FHR93a, FHR93b]. Hermes/ST[2] is an object-oriented distributed programming environment that Michael Fazzolare, David Ranson and the author have developed and implemented in Smalltalk/80 [GR89]. An example application, a distributed bank, is used for demonstration throughout Chapter 3. Chapter 4 presents the core of this thesis. The scheduling properties, the scheduling mechanisms and its efficient implementation are described and their correctness is analyzed. Chapter 5 compares the scheduling mechanism with traditional approaches. It also presents some performance figures of the implementation in Hermes/ST. Chapter 6 summarized the results of this thesis and outlines areas of continuing research.

[2] Hermes/ST is not to be confused with IBM's Hermes system. The postfix stands for the implementation language, Smalltalk.

Chapter 2

Transactions and Objects in Distributed Systems

This chapter provides an overview of the use of nested transactions and object-orientation in distributed systems. Section 2.1 describes some issues in developing reliable distributed systems. Section 2.2 introduces transactions as a concept that addresses these issues. Nested transactions go further and are presented in Section 2.3. Section 2.4 describes the main concepts of object-orientation and how they are advantageous in the context of distributed systems. Finally, Section 2.5 gives an overview of relevant existing academic and commercial distributed systems that support nested transactions and objects.

Most of the terminology and some descriptions used in this chapter have been adapted from standard textbooks including [BHG87, GR93, Mey88]. However, this chapter is by no means a textbook-style introduction to the fields of nested transactions and object-orientation. Rather, it introduces concepts and terminology that are important for the understanding of this thesis. For example, concurrency control, and particularly the concept of serializability, are discussed in detail since the core of this thesis deals with concurrency control issues. Recovery, on the other hand, is only mentioned briefly for completeness.

Furthermore, concepts like serializability are not defined formally. Rather, this chapter tries to convey a fundamental but intuitive understanding of these concepts to the reader. No prerequisite knowledge of the reader is assumed except an understanding of the fundamental concepts of computer science.

2.1 Issues in Distributed Systems

A *distributed system* is a collection of programs that execute *concurrently* over a set of computers, in this context called "*nodes*". A node consists of processor(s), local memory, possibly some stable storage like disk(s) and I/O ports to connect it with the environment. Nodes communicate via *networks* that interconnect their I/O ports. Concurrency and distribution pose problems that do not exist or exist in a less complex form in sequential, centralized systems. Some of these problems are discussed below.

Interleaving Operations: Without appropriate *concurrency control*, concurrent operations may *interleave* in a way that leads to incorrect outcomes. Consider the following example from the banking domain. The pseudo code below describes the implementation of an operation that deposits some amount of money into a bank account.

```
deposit(amount, accountNumber)
```

```
{
    tmp := read (Accounts[accountNumber]);
    tmp := tmp + amount;
    write (Accounts[accountNumber], tmp);
}
```

Now consider the following scenario. The initial balance of a bank account is \$1,000. Two customers deposit money to this account using the deposit operation described above. The first customer adds \$10,000 and the second customer adds \$100. In a sequential system, the account balance will be \$11,100 after both deposit operations have finished. \$11,100 is the *correct* account balance after both deposits. However, a different, i.e. *wrong*, outcome is possible in a concurrent system without concurrency control. The two deposit operations could, then, interleave as shown in Table 2.1.

execution order	deposit operation #1	deposit operation #2
1.	read balance: \$1,000	
2.		read balance: \$1,000
3.	write: \$1,000 + \$10,000	
4.		write: \$1,000 + \$100

Table 2.1: Example for two interleaving deposit operations.

The account balance after both deposit operations have finished is \$1,100, the value written by the second deposit operation. The wrong outcome is due to the uncontrolled interleaving of the two deposit operations. To guarantee the correct outcome in this case, concurrency control must ensure that *both* read and write operations of deposit operation #1 must be performed either before or after *both* read and write operations of deposit operation #2.

Node and Network Failures During Operation Execution: Distribution adds further complication. At any point in time in a single-node system, the entire system is either running or it is crashed. In a distributed system, some nodes can be running and some nodes can be crashed. Also, some communications links may be available and some links may be unavailable. Operations that visit different nodes can leave the system in an *inconsistent state*[1] if some of the nodes crash or are unavailable due to network failures during the execution of those operations.

Consider another example from the banking domain: the transfer of funds from a source account to a destination account. A transfer operation can be implemented by performing a withdraw operation on the source account and by performing a deposit operation on the destination account. Consider the scenario where the withdraw operation is performed successfully but the deposit operation cannot be performed because the destination node is crashed or unavailable due to network failure. The transfer operation then has a wrong outcome, in that money was deducted from the source account but has not been added to the destination account.

Node Crashes After Operation Execution: Even after a distributed operation has finished successfully, *subsequent* node crashes can destroy its effects and can leave the system in an inconsistent state. Consider the example that the transfer operation of the

[1] A system is in an inconsistent state if particular domain-specific constraints about system data are not satisfied. An example for such a constraint is that all account balances in a bank database must be positive. The bank database is in an inconsistent state if some account balances are negative.

previous paragraph was performed successfully but a subsequent node crash occurs at the destination account. Then, the system is left in the same inconsistent state as if the node failure had occurred during the operation.

To ease the programming of concurrent and distributed systems, convenient abstractions are used that mask problems like the ones mentioned above from the application programmer. Note that there is no mechanism that masks *all* possible failures. Consider a processor failing by exhibiting arbitrary behaviour, e.g. acknowledging to have performed an operation when, in actual fact, it has not[2]. There is no way of detecting such failures in general. For this reason, *failure models* [Sch93] have been introduced. Failure models classify common types of failures. Mechanisms that mask failures are specified with reference to failure models. These state which kinds of failures can and cannot be masked by a particular mechanism. A convenient abstractions for reliable computing is the transaction concept, which is introduced in the following section.

2.2 Transactions

The transaction concept [BHG87, GR93] was originally developed in the database area in the early seventies [BD72, Bjo73, Dav73]. It ensures reliability under the following failure model.

- A node consists of *volatile* and *permanent* memory and can *crash* at any time. A node crash destroys volatile memory but leaves all of the permanent memory intact.
- Nodes do not crash forever.
- Messages between nodes can get lost or they may arrive in arbitrary order. However, messages are always delivered to the correct receiver and if they arrive, they arrive intact.

As mentioned above, failures outside this failure model can occur in reality. An example of such a failure is the corruption of permanent memory. However, a system can be designed so that the likelihood of failures outside this failure model can be made arbitrarily small. The likelihood of failure then depends on how much one is willing to pay for reliability in terms of resources. Some failures outside this failure model are discussed in Section 2.2.2.

A transaction forms a group of operations that may access (read or write) system data and that may return a result. A transaction has three properties[3]:

Atomicity: A transaction either happens in its entirety ("*commits*") or not at all ("*aborts*").

Serializability: Operations of concurrent transactions appear to the outside world as if they do not interleave.

Permanence: If and when a transaction commits then its effects are made permanent i.e. they are not affected by subsequent node crashes.

[2]This kind of failure is referred to as *Byzantine failure* [LSP82].

[3]The transaction properties are only described intuitively, here. A more complete discussion is performed in the following sections. It shall also be noted that other classifications of the transaction concept can be found in the literature, e.g. in terms of the four properties of atomicity, consistency, isolation and durability [GR93].

Transactions deal with all the problems mentioned in the previous section.

- Transaction executions do not interleave in a way that leads to wrong outcomes.
- If some nodes that are visited by a transaction crash during the execution of a transaction or some nodes cannot be accessed because communication links are unavailable then the transaction aborts. In this case, all changes to data performed by the operations of the transaction are undone. Thus, data inconsistencies cannot occur due to node and network failures during the execution of a transaction.
- If all nodes and communications links needed for the execution of a transaction are available and the transaction finishes successfully, then it commits. In this case, all changes to data performed by the operations of the transaction are made permanent. Thus, subsequent node node crashes cannot destroy data written by a committed transaction and hence cannot lead to inconsistencies.

The transactional properties are ensured by mechanisms commonly termed *concurrency control* (for serializability) and *recovery* (for atomicity and permanence). Both mechanisms are discussed in the following sections.

2.2.1 Concurrency Control

2.2.1.1 Serializability

Serializability is the definition of correctness of concurrency control in transactional systems [BHG87]. It is therefore the goal of concurrency control to provide serializability in order to avoid errors caused by interleaving transactions.

Reconsider the deposit example of Section 2.1 where the deposit operation of the first customer is described as transaction T_1 and the deposit operation of the second customer is described as transaction T_2. The problem of incorrect outcome due to execution interleaving is avoided trivially if T_1 and T_2 never interleave, i.e. the two transactions are scheduled serially. A *serial schedule of two transactions* T_1 and T_2 is defined to be that either all operations of T_1 execute before all operations of T_2 or all operations of T_2 execute before all operations of T_1. The definition does not state in which order T_1 and T_2 execute as long as they execute in some particular serial order. A *serial schedule of a set of transactions* is defined to be that all pairs of transactions in this set are scheduled serially.

The serial scheduling of all transactions in a system trivially solves the problem of incorrect outcomes due to interleaving execution, since it does not allow transactions to interleave at all. However, it has serious drawbacks since it also allows no concurrency at all. In a distributed system that incorporates many processors, serial execution of transactions makes poor use of the system's processing resources. Poor performance is a consequence.

Therefore, the concept of serial schedules is extended to the concept of serializable schedules, which keeps the advantages of serial schedules while removing their disadvantages. The schedule of two transactions T_1 and T_2 is defined to be *serializable* if T_1 and T_2 have the same effect on system data and return the same result as if they had been scheduled serially. Consequently, a *serializable schedule for a set of transactions* requires serializable schedules for all pairs of transactions in this set. Every serial schedule is also a serializable schedule but the opposite is not true. Serializable schedules allow interleaving executions of transactions as long as this does not affect data accesses and return values.

Note that as with serial schedules, no particular execution order is specified for serializable schedules. However, sometimes the semantics of an application requires particular execution orders for transactions. In this case, it is the application program's responsibility that the preferred order actually occurs. For example, if a transaction T_1 must be performed before a transaction T_2, then T_2 should only be started after T_1 has committed.

2.2.1.2 Optimistic versus Pessimistic Concurrency Control

The system components performing concurrency control are called "*concurrency controllers*". Concurrency controllers guard accesses to individual data items to ensure serializability. A concurrency controller controlling access to a data item has three options when a transaction's operation requests access to this data item. It can:

1. schedule the request immediately,
2. delay the request and schedule it at some later time or
3. reject the request, hence causing the transaction to abort.

Different concurrency control strategies favour different options:

Optimistic concurrency control favours Options 1 and 3. Requested operations are not delayed but are scheduled immediately (Option 1). Serializability is tested *a posteriori* at transaction commit. However, the system can get into situations in which there is no possibility of finishing all transactions in a serializable way. The system then has to reject operations which causes the respective transactions to abort (Option 3).

Pessimistic concurrency control favours Option 2. Operation requests are delayed until serializability can be ensured *a priori* (Option 2). However, the system may get into deadlock situations in which case some transactions have to be aborted (Option 3).

Optimistic concurrency control potentially allows higher concurrency but it may lead to a phenomenon called "*cascading aborts*". Recall that when a transaction aborts then all effects of the aborting transaction must be undone. They include effects on data as well as effects on other transactions. Consider the following example from [BHG87]. Suppose that the initial values of two data items x and y are 1 and transactions T_1 and T_2 issue operations that are executed in the order shown in Table 2.2.

execution order	T_1	T_2
1.	$write(x, 2)$	
2.		$read(x)$
3.		$write(y, 3)$

Table 2.2: Example for cascading aborts.

Suppose that T_1 aborts. Then, the system undoes T_1's $write(x, 2)$ operation, restoring x to the value 1. Since T_2 reads the value of x that has been written by T_1, T_2 must be aborted, too—a cascading abort. Thus, the system must also undo T_2's $write(y, 3)$ operation, restoring y to 1.

Cascading aborts are undesirable because they require significant bookkeeping and entail the possibility of forcing many transactions to abort just because some other transaction happened to abort.

Pessimistic concurrency control avoids cascading aborts but may lead to *deadlocks*. Deadlocks are described in Section 2.2.1.4. Neither of the two concurrency control strategies always outperforms the other one. It is merely the characteristics of a particular application domain which determine which one of the two is more appropriate. In domains where transactions rarely conflict, an optimistic approach is more suitable. In domains where conflicts are common, a pessimistic scheme is preferable [BHG87]. In addition, other factors like the workload of a system (the number of concurrently executing transactions) affects the performance characteristics of the two strategies. Almost all concurrency control mechanisms (see Section 2.2.1.4) have optimistic and pessimistic versions. In practice, pessimistic concurrency control is more commonly used than optimistic concurrency control since it has better performance characteristics over a wider range of parameters [BHG87].

2.2.1.3 Single-Version versus Multiple-Version Concurrency Control

In *single-version concurrency control*, all transactions access (i.e. read and write) data items *directly*. In contrast, in *multiple-version concurrency control*, each write operation to a data item causes the creation of a new copy of the data, called a "*version*". Working on versions of the data instead of on the data itself, may help the concurrency controller avoid rejecting operations that arrive late. Without going into details, it shall be noted that most concurrency control mechanisms (see next section) have been defined for single and multiple versions.

2.2.1.4 Two-Phase Locking

Three main concurrency control mechanisms can be distinguished: *two-phase locking* ("*2PL*"), *timestamp ordering* and *serialization graph testing* [BHG87]. 2PL and especially a particular version called "*strict 2PL*" is the most popular mechanism in commercial systems [BHG87] and is introduced below.

In 2PL, data items are associated with *locks*. The most commonly used lock modes are *read/write locks*. Other lock modes are *mutual exclusion ("mutex") locks* and *type-specific locks*. Transactions must acquire "appropriate" locks before they access data; e.g. they must acquire a read lock before reading a data item and a write lock before writing to a data item. Transactions *hold* a lock until they *release* it. A transaction cannot acquire a lock as long as it is held by another transaction in a *conflicting* mode. Whether a particular lock mode conflicts with another lock mode is typically defined in a *lock compatibility matrix*. The lock compatibility matrix for read/write locking is shown in Table 2.3, allowing multiple readers but only a single writer. The rows represent the lock mode of the lock that is requested. The columns represent the lock mode of the lock that is held. The table entries show the compatibility.

	read	write
read	yes	no
write	no	no

Table 2.3: Lock compatibility matrix for read/write locking.

In strict 2PL, transactions may not release any locks before they commit or abort. [EGLT76] show that strict 2PL ensures serializability. However, strict 2PL may lead to *deadlocks* as shown in the example of Table 2.4. T_1 and T_2 denote two transactions, D_1 and D_2 two data items.

execution order	T_1	T_2
1.	acquires read lock on D_1	
2.		acquires read lock on D_2
3.	tries to acquire write lock on D_2	
4.		tries to acquire write lock on D_1

Table 2.4: Example for deadlock.

T_1 cannot acquire a write lock on D_2 unless T_2 releases its read lock, i.e. commits or aborts. Conversely, T_2 cannot acquire a write lock on D_1 unless T_1 releases its read lock, i.e. commits or aborts. No progress is possible unless at least one of the two transactions is aborted.

In this example, T_1 waits for T_2 and T_2 waits for T_1. In deadlock literature, the *waits-for* relationship between transactions is typically represented as a graph. Deadlock occurs when there occurs a cycle in the *waits-for graph.*

There are three main approaches to handling deadlocks, namely *prevention*, *avoidance* and *detection*.

Prevention: Accesses to data items are globally ordered so that deadlocks cannot occur. This can, for example, be achieved by having transactions pre-declare the data items they are going to access. The system can then schedule the transactions accordingly. Another way of achieving this is to specify a system-wide canonical order over the data items and have transactions acquire locks according to this order.

Avoidance: There are various mechanisms that abort transactions during execution when there is the *potential* of deadlocks being formed. The simplest form is called "*no-waiting*": a transaction is always aborted and restarted when it fails to acquire a lock. More sophisticated mechanisms include *cautious waiting* and timestamp-based approaches like *wound-wait* and *wait-die* [RSL87].

Detection: While transactions are executing, accesses to data items are recorded, e.g. by maintaining a waits-for graph. Whenever a cycle is detected in the waits-for graph, the cycle is broken by aborting one or more transactions. Another simple form of detection of potential deadlocks is by aborting a transaction when its execution time exceeds a specified timeout limit.

2.2.2 Recovery

This section deals with mechanisms for recovering from failures. Three kinds of failures can be distinguished, namely *transaction abort*, *node crash*, and *catastrophe*.

Transaction Abort: Transactions can abort due to node crashes, deadlocks, messages that cannot be delivered, or explicit software aborts. The atomicity property of transactions requires that all effects of an aborting transaction must be undone.

Node Crash: The volatile memory and all active processes of a crashing node are lost but permanent memory stays intact. The permanence property of transactions requires that committed transactions are not affected by subsequent node crashes.

Catastrophe: The permanent storage of a crashing node gets corrupted. This case is outside the failure model for transactions and it is therefore not handled by transaction mechanisms. Other mechanisms must be employed to recover from catastrophic failures.

Mechanisms for recovery from these kinds of failure are discussed in the following three sections.

2.2.2.1 Abort Recovery

Abort recovery ensures the atomicity property of transactions. If a transaction aborts then all effects of the aborting transaction must be undone. This ensures that the transaction either happens in its entirety or appears not to have happened at all. Two main mechanisms for abort recovery are distinguished, namely *undo logging* and *redo logging*.

Undo Logging: Write operations to data items are applied to the data items directly. However, before a data item is written, its value is saved in an *undo log*. When a transaction commits, undo log elements created by it are simply discarded. However, when a transaction aborts, the the undo log elements are used to restore all data items the transaction has written, to the values they had before the transaction started.

Redo Logging: Write operations to data items are saved in a *redo log* but are not applied to the data items while a transaction is executing. At transaction abort, the redo log entries of the aborting transaction are simply discarded. However, at transaction commit, the redo log entries are replayed on the actual data items.

Undo logging outperforms redo logging in applications where read operations are common and transaction aborts are rare. However, redo logging can exhibit better performance in domains where write and abort operations are common. Most transactional systems use some form of undo logging.

2.2.2.2 Crash Recovery

Crash recovery deals with the permanence property of transactions. Committing transactions must save their changes to permanent storage so that subsequent node crashes cannot undo their effects. The most commonly used approach to crash recovery is to keep a *log* on permanent storage along with the actual system data. Data updates, commits and aborts of transactions are recorded in this log. The log is used to repair the system data on permanent storage after a node crash. Two kinds of log records are distinguished. *Update records* contain undo and redo information and *status records* contain commit and abort information.

The most commonly used protocol to ensure that the commit of a distributed transaction[4] is performed atomically is the *two-phase commit protocol (2PC)*. One node that has been involved in the committing transaction[5] is chosen as the *coordinator*. All nodes involved in the transaction (including the coordinator) are called *participants*. The two phases of the commit protocol are called "*prepare phase*" and "*commit phase*".

Prepare Phase: The coordinator asks all participants to write prepare records to permanent storage. If they have not crashed since the start of the transaction then they perform the write operation and reply positively, otherwise they reply negatively. Once a participant has prepared it cannot commit or abort the transaction on its own.

[4] A distributed transaction is a transactions whose operations visit different nodes.

[5] A node has been involved in a transaction if an operation of this transaction has visited this node.

Commit Phase: If all participants have replied positively, the coordinator can decide to commit. Otherwise it decides to abort. The decision must be written to the log on stable storage before all participants are informed about it. When the participants receive the decision then they must write it to their log on permanent storage. They then reply back to the coordinator. The commit phase finishes only after the coordinator has received positive replies from all participants.

The 2PC protocol is prone to coordinator node crashes and in this case, participant locks are held for a potentially long time. More sophisticated, but also more expensive, mechanisms like the *three-phase commit protocol* [Ske82] address this issue.

2.2.2.3 Catastrophe

Since catastrophes are outside the failure model for transactions, transaction mechanisms do not deal with them. However, logging mechanisms similar to the ones presented above are commonly used to keep the likelihood of data inconsistency and loss as small as desired. A common approach is to use *mirrored disks* as backups of the system's permanent storage. Mirrored disks replicate the system data on different nodes. Increasing the number of mirrored disks decreases the likelihood of unrecoverable failure. It is a matter of how much one is willing to pay for reliability in terms of resources and performance.

In this section, mechanisms for concurrency control and recovery have been presented separately. However, it is worth noting that, strictly speaking, the mechanisms interact in subtle ways. One cannot discuss the correctness of a concurrency control mechanism in isolation from recovery mechanisms and vice versa. Consider the example of a relational database system where several relations are stored on a single page on disk. If concurrency control (e.g. 2PL) is performed on individual relations but abort recovery (e.g. undo logging) is performed on the page level then the transactional properties cannot be ensured. This is because a committing transaction could make pages permanent which contain relations that have been written by uncommitted transactions. Conversely, aborting transactions could undo changes of executing transactions without forcing them to abort.

2.3 Nested Transactions

The transaction concept as introduced in the last section is a convenient abstraction for reliable programming. Although transactions were originally developed for databases, they address problems that also occur in distributed systems. Therefore, transactions have been adopted for distributed systems programming. However, there are drawbacks of the simple, single-level transaction concept when used in general distributed systems programming. The simple transaction concept is only suitable for short and simple transactions. This is because it has the following restrictions.

- It does not allow the composition of several simple transactions into more complex transactions.
- It does not allow concurrency within transactions.
- A single failure like a deadlock causes the whole transaction to abort and possibly a large amount of work to be undone.

Since database queries and updates tend to be short, the single-level transaction concept is normally sufficient for database programming. However, for transactions to be a convenient abstraction in general distributed programming, the restrictions mentioned above need to be addressed. *Nested transactions* [Ree78, Mos81] do exactly this.

In the nested transaction model, transactions can create other transactions called *subtransactions*. Subtransactions can execute synchronously or asynchronously. Transaction nesting structure can be represented by a *transaction tree* where nodes of the tree represent transactions and arcs of the tree represent *is-subtransaction-of* relationships. Transaction trees can be arbitrarily deep. The root node of a transaction tree is called *top-level transaction*, all inner nodes are called *subtransactions*. Usual tree notations like *parent, child, ancestor* and *descendant* are used. Note that the ancestor and descendant relationships are reflexive, i.e. every transaction is its own ancestor and descendant. The non-reflexive counterparts are *proper ancestor* and *proper descendant*. Various top-level transactions executing concurrently in a system form a forest of transaction trees. The three transactional properties of serializability, atomicity and permanence are ensured for the execution of each entire transaction tree.

Atomicity: The execution of an entire transaction tree runs to completion (*top-level transaction commit*) or the effects of the entire transaction tree are undone (*top-level transaction abort*). All effects of a transaction tree whose top-level transaction has committed are visible to other transaction trees. Top-level transaction abort ensures that all descendant transactions have aborted and therefore the effects of the entire transaction tree are undone.

Serializability: The execution of each transaction tree is serialized with the execution of every other transaction tree.

Permanence: All changes to system data performed by any transaction in the tree are made permanent at top-level transaction commit.

Since top-level transaction commit and abort represent the commit and abort of the entire transaction tree, the term top-level transaction is henceforth used to denote the entire transaction tree. Subtransactions have different serializability, atomicity and permanence properties to top-level transactions.

Atomicity: The execution of a subtransaction subtree[6] runs to completion (*subtransaction commit*) or the effects of the entire subtree are undone (*subtransaction abort*). Recall that whenever a top-level transaction has committed, it can never be consequently aborted. This is not true for subtransactions. Subtransaction abort ensures that all descendant transactions have aborted. This means that a committed subtransaction can be aborted by an aborting *proper* ancestor transaction. No effects of a committed subtransaction are visible to other top-level transactions. Also, an aborting subtransaction does not necessarily cause its parent transaction to abort. It is the parent transaction's decision to retry or ignore the subtransaction, take some compensating action or abort itself.

Serializability: The serializability property is maintained between asynchronous subtransactions.

Permanence: Effects of committing subtransactions are conceptually not made permanent. There are, however, *early writing* and *checkpointing* strategies that write

[6] A subtransaction subtree consists of the subtransaction and all its descendant transactions.

subtransaction commit log entries to permanent storage before top-level transaction commit. However, these strategies only reduce the amount of work to be done at top-level transaction commit or they reduce the likelihood of top-level transaction aborts due to node crashes. They are not necessary to ensure the semantics of nested transactions.

Nested transactions address the deficiencies of single-level transactions described above.

- Arbitrary transactions can be composed into larger transactions.
- Concurrency is allowed within subtransactions. Serializability between subtransactions ensures that there are no incorrect outcomes due to execution interleaving.
- Failures do not necessarily cause a top-level transaction to abort. Aborting subtransactions can be retried or compensating action can be taken which potentially avoids large amounts of work to be undone.

Two main mechanisms for implementing nested transactions have been proposed: Reed's mechanism is based on timestamp ordering [Ree78] and Moss' mechanism uses locking [Mos81]. Moss' design is most commonly used. A brief overview is given below.

Concurrency Control: Moss' concurrency control mechanism is an extension of 2PL. Data items are associated with locks. Two lock modes are supported: read locks and write locks. Transactions can acquire locks if for all transactions currently holding this lock the following is true: either the lock modes are compatible according to the read/write locking rules (Table 2.3) or the transaction holding the lock is an ancestor of the transaction requesting the lock. On subtransaction commit, locks held by the transaction are handed to the parent which then holds the lock[7]. This process is called *upward lock inheritance*[8]. On subtransaction and top-level transaction abort and on top-level transaction commit, locks held by the transaction are released. See Chapter 5 for details.

Recovery: Moss uses a form of undo logging for abort recovery. Every transaction performing a write operation creates an undo log entry. On subtransaction and top-level transaction abort, all data items written by the aborting transaction and all of its descendents are restored to their values before the transaction started. All log entries created by the aborting transaction and all its descendents can then be discarded. On top-level transaction commit, all log entries of the entire transaction tree can be discarded.

Moss' design uses a 2PC protocol for top-level transaction commit. No early writing is performed at subtransaction commit. For crash recovery, a simple logging mechanism is used.

2.4 Object-Orientation in Distributed Systems

2.4.1 The Main Concepts of Object-Orientation

Another technology that has originally been developed in a different area of computer science, namely simulation, has been applied to distributed systems programming: *object-*

[7] Slightly different mechanisms are described in [Mos81] and [Mos85]. Chapter 5 goes into details of the differences.

[8] Different terminology has been used for this concept, including "*lock inheritance*" and "*lock anti-inheritance*". In this thesis, the term "upward lock inheritance" is used to easily distinguish this concept from another concept called "downward lock inheritance". The differences are discussed in Chapter 5.

orientation [Mey88, Boo90, WBWW90, RBP+91]. The main concept of object-orientation is the *object*. An object is an entity that encapsulates:

- private state information, in form of *variables*;
- operations, called "*methods*", that can access (read and modify) the object's variables.

An object's variables are completely protected and hidden from other objects. The only way an object can be examined or modified is by invoking its methods. This property is called "*encapsulation*" and it supports *information hiding*. Objects communicate by invoking other objects' methods. This is called "*sending messages to objects*" and therefore communication between objects is called "*message passing*". Sometimes, *public* and *private* methods are distinguished. Only public methods can be invoked by other objects whereas private methods can only be invoked by the object itself. The implementation of an object's methods is hidden from other objects. Only the interfaces of public methods are visible to other objects. The interfaces of all public methods specify a well-defined interface for the functionality, an object provides.

The *class* concept is a direct extension of the *abstract data type* concept. A class acts as a template from which objects may be created, specifying the objects' variables and methods. Every object is an *instance* of some class.

Different classes can be specified to be in a *subclass-superclass relationship*. A mechanism called "*inheritance*" allows commonalities between subclasses to be factored out and specified once in a superclass. Instances of a subclass encapsulate not only all variables and methods defined in the subclass' definition but also all variables and methods defined in its superclasses' definitions. If the superclasses have superclasses themselves then their variables and methods are included as well and so on. The terms *descendent class* and *ancestor class* are used for repeated subclass-superclass relationships. Variables and methods that have not been defined in a class itself but in one of its ancestor classes are said to be "*inherited*" by the class.

A subclass is free to add variables and methods not specified by any of its ancestor classes. It is also free to modify the implementation of methods which are specified by ancestor classes. This process is called "*overriding*" inherited methods. There are different versions of the inheritance concept.

Single Inheritance versus Multiple Inheritance: Classes and their subclass-superclass relationships form a directed graph where classes form the nodes and the relationships form the arcs. Single inheritance requires this graph to be a tree whereas multiple inheritance only requires that the arcs do not form cycles. Multiple inheritance is more general than single inheritance but it can lead to name clashes of variable or method names defined in different ancestor classes.

Strict versus Non-Strict Inheritance: Both forms of inheritance allow descendant classes to add inherited variables and methods and change the implementation of methods. Non-strict inheritance additionally allows subclasses to remove methods or change their interfaces. Strict inheritance disallows this. Non-strict inheritance is more general than strict inheritance but it makes the subclass-superclass relationship incompatible with the useful subtype-supertype relationship.

Single-Rooted versus Multi-Rooted Inheritance: Single-rooted inheritance allows, system-wide, only one class which has no superclass. This class is typically called

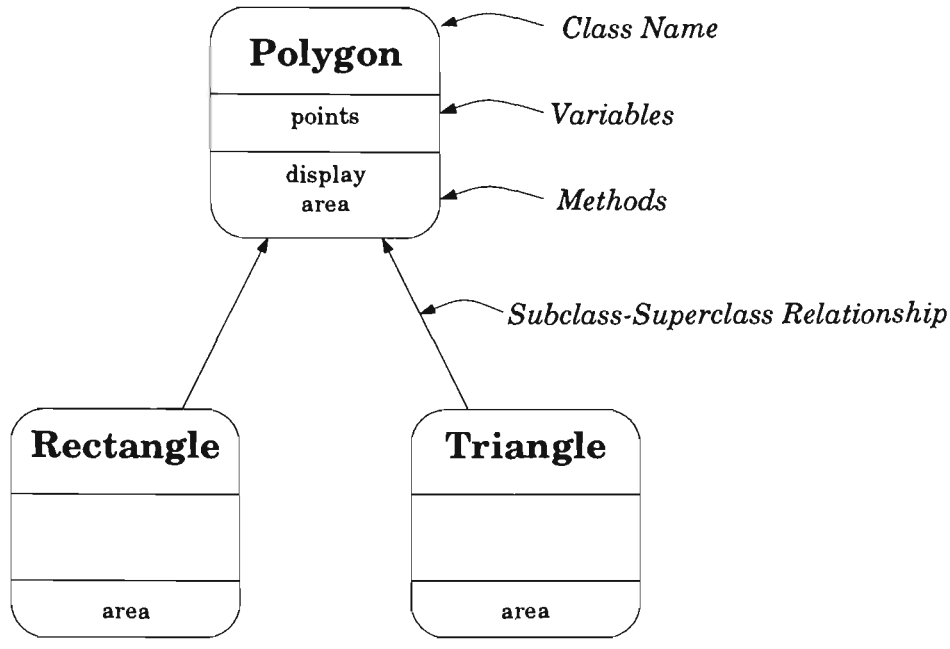

Figure 2.1: A single inheritance hierarchy for polygon classes

`Object`. All other classes defined in a system are descended from `Object`. Multi-rooted inheritance allows several classes to have no superclass. Multi-rooted inheritance is more general than single-rooted inheritance. However, single-rooted inheritance conveniently allows common behaviour like copying and printing to be shared by all system classes by defining it in terms of methods of `Object`.

Consider the example of a single inheritance hierarchy in Figure 2.1. Instances of class `Polygon` represent polygons which are described by a collection of points. The points describing a polygon are specified in a variable called `points`. Class `Polygon` defines two methods `display` to display a polygon on a screen and `area` to return the area of a polygon. The classes `Rectangle` and `Triangle` are defined as subclasses of `Polygon`. Due to inheritance, all instances of `Rectangle` and `Triangle` have a variable `points` and methods `display` and `area`. The `display` method defined in `Polygon` is sufficient for `Rectangle` and `Triangle` and therefore does not need to be overridden. However, different formulas for computing the area of rectangles and triangles make overriding the `area` method necessary.

Classes that cannot be instantiated themselves, but that have descendant classes that can be instantiated, are called *abstract classes*. `Polygon` could be an example.

The fact that all instances of `Polygon` and all its descendant classes have an `area` method (either via inheritance or via overriding) allows one to write programs in which general polygon type objects can be sent the message `area`. General polygon type objects, here, mean instances of class `Polygon` itself or any of its descendant classes. The important point is that such programs can be used for instances of different polygon type objects. It can be decided at run-time of such a program which type of polygon is actually used and therefore which implementation of `area` is to be applied. This ability to write programs that "take several forms"[9] is called "*Polymorphism*". Polymorphism allows flexibility in programming and factoring of commonalities and is an important feature of object-orientation.

In the polygon example, the subclass-superclass relationship is used to express an *is-a relationship* that exists between the real-world entities these classes represent: every

[9]This is the meaning of the term "Polymorphism".

rectangle is a polygon and every triangle is a polygon. Because of this relationship, the different entities share common behaviour, e.g. they all can be displayed or have some area. The usage of inheritance in this context is therefore called "*behaviour sharing*".

There is another common usage of inheritance which is called "*code sharing*". Consider the example of a class `LinkedList` which implements the common linked list data type with methods `first`, `addFirst`, `removeFirst` and `isEmpty`. Now consider a class `Stack` which implements the common stack data type with methods `push`, `pop`, `top` and `isEmpty`. Class `Stack` is defined as a subclass of `LinkedList`. Method `push` invokes the inherited method `addFirst`, `pop` invokes `removeFirst`, `top` invokes `first` and `isEmpty` is inherited but not overridden. In contrast to the polygon example, there is no is-a relationship between `Stack` and `LinkedList`. The relationship can be rather described as "*is-implemented-by*". However, as with behaviour sharing, this usage of inheritance allows code to be factored out, defined only once and used in various different contexts.

Although there are many different approaches to object-orientation, there are three fundamental concepts that are common to all of them: the concepts of object, class and inheritance. When inheritance is left out then the term "*object-based*" is used. Object-orientation has first been applied to programming languages. Smalltalk-80 [GR89], C++ [Str86] and Eiffel [Mey88] are prominent examples. Recently, object orientation has also been applied to the analysis and design phases of software development [Boo90, WBWW90, RBP+91, HS91, CY91a, CY91b].

2.4.2 Advantages of Object-Orientation

Reusability: Classes describe behaviour of abstract data types. Classes which are useful in various different contexts can be defined in reusable class libraries. A lot of effort can be put into the optimization and validation of classes that are used very often. Behaviour sharing and code sharing allow common behaviour and code to be factored out and used in various different contexts. Encapsulation allows classes to be used in different contexts without the danger of internal implementations interacting in unexpected ways. Polymorphism facilitates such usage.

Extensibility: Designing object systems involves specifying object interfaces and their message communications. After object interfaces have been specified, their functionality can often be implemented rapidly in a prototypical fashion. This allows early validation of the functional specification of a system. Refining the implementations of objects, e.g. for minimizing time and space requirements and improving reliability, does not affect the general system behaviour provided interface specifications are adhered to. Also, adding new classes and objects does not affect interactions of existing objects. This is because interactions between objects are reduced to the messages they pass. This allows an incremental development methodology where a system is prototyped first and then gradually extended and refined to the final system.

Maintainability: The fact that the interaction of objects is limited to the messages they pass makes the maintenance of large object systems easier. Also, using class libraries in which single components have been validated and optimized increases the reliability of a system. Exchanging the implementation of objects during system operation is unproblematic as long as the old object interfaces are still supported.

These general advantages of object-orientation, reusability, extendibility and maintainability, also apply to distributed systems.

Reusability: Distributed systems tend to be very large and therefore expensive to develop. Software reuse can reduce such costs.

Extensibility: Extensibility features are particularly useful for distributed systems that tend to evolve during their usage.

Maintainability: Maintaining distributed systems is more complex than maintaining sequential, single-node systems. The fact that the implementation of objects can be replaced relatively easily at run-time is a useful feature for maintaining distributed systems.

Apart from these general advantages of object-orientation, the paradigm is particularly useful in the context of distributed systems for a number of reasons. An object is a unit of tightly coupled data and processing, whereas different objects are loosely coupled. This property is advantageous when objects are distributed with every object residing on only one node. Then, intra-object computation is always performed locally and only intra-object communication may require network access. The loose coupling of objects brings about that expensive network communications are rare.

Objects can be extended naturally to distributed objects where the object is a unit for many important concepts of transactional and distributed systems.

Remote Access: Message passing between local objects extends naturally to message passing between remote objects as a means of accessing remote nodes. Remote method invocation can, for example, be implemented via the *remote procedure call* [BN84]. This also integrates naturally with the common client/server paradigm where the sender object acts as a client and the receiver object acts as a server.

Concurrency Control: Encapsulation is beneficial for concurrency control. Concurrency controllers can be specified on a per-object basis that schedule the invocation of public methods. Uncontrolled access to an object's state can not occur. This is because an object's state can be inspected or modified only by invoking its public methods.

Another useful unit for concurrency control is an object's individual variables. Variables can only be read or written. Therefore, read/write locking can be applied. Choosing the level of concurrency control (whole object versus individual variables) is a trade-off between concurrency control cost and the amount of concurrency gained [Faz94].

Abort Recovery: An object's state is specified by the values of its variables. Like concurrency control, the object paradigm offers two useful units for abort recovery: whole objects and individual variables. When performed on the object level, the effects on all variables must be undone when at least one variable has been written. When performed on the individual variable level, the effects on all variables that have been written must be undone.

Recall from Section 2.2 that concurrency control on individual variables and abort recovery on whole objects is problematic.

Crash Recovery: The object is a natural unit for persistence, allowing recovery from node crashes. The state of such a *persistent object*, i.e. the values of its variables, is mirrored on permanent storage.

Replication: A single logical object can physically be replicated on different nodes. Replication can be useful for performance or reliability reasons. The fact that an object's state can only be accessed via its public methods allows the system to maintain consistency between copies. The system can ensure consistency on method invocations, e.g. using a quorum mechanism [Gif79].

Migration: Objects as a whole can be migrated between nodes. If message passing is *location transparent*[10] then migrating objects do not affect system behaviour.

This enumeration shows that the object concept is very beneficial in the context of distributed systems. Therefore, object-orientation is widely used for distributed systems. It shall be noted that inheritance, however, although central to providing many of the advantages of object-orientation, is often left out in the context of distributed systems. This is due to efficiency concerns, since inheritance may require replication of large amounts of code on different nodes or the performing of remote code lookups at run-time.

2.5 Distributed Systems Supporting Nested Transactions and Objects

Nested transactions and object-orientation have been applied to distributed systems. The first major implementation to integrate both technologies was *Argus* [Lis82, LS83, LCJS87, Lis88]. Argus supports objects called "*guardians*". A guardian resides on one node in a heterogeneous network and encapsulates data elements called "*objects*". These objects are data structures rather than objects in the sense of object-orientation. Two kinds of objects are distinguished: *atomic objects* and *non-atomic objects*. Atomic objects support transactional properties and are the unit for concurrency control, abort and crash recovery. Non-atomic objects are volatile and do not provide concurrency control and recovery.

Guardians are the unit of remote access. Guardians define a set of methods that are called "*handlers*". The only way of inspecting or modifying a guardian's object is by invoking its handlers. Handler invocation is location-transparent. Argus takes care of all the details for constructing and sending messages. Every handler call implicitly creates a transaction. Handlers that invoke other handlers create nested transactions. Transactions can also be created explicitly. Concurrency between parent and child transactions is not supported. However, a concurrent loop construct specifies concurrency between sibling transactions. Transactional properties are ensured as long as transactions access atomic objects only. Accessing non-atomic objects reduces the cost of transactions since non-atomic objects do not perform concurrency control and recovery. Therefore, transactional properties cannot be ensured in this case. Non-atomic objects allow the application programmer to explicitly defy serializability or have non-committed transactions communicate in special situations where this is desired.

The Argus project was successful in that it is much easier to develop reliable distributed systems in Argus than in comparable systems which were in use at that time. Argus has had and still has a great impact on distributed systems research. Many research systems followed the Argus example and integrated nested transactions and object technology with distributed systems[11]. Examples are *Camelot/Avalon* [EME91], LOCUS

[10] Location transparency means that there is no syntactic difference between a local method invocation and a remote method invocation. The system distinguishes the two cases and reacts accordingly.

[11] It shall be noted that some of the research systems were developed at the same time as Argus and there has, of course, been mutual influence.

[MMP83], TABS [SBD+84], *Eden* [PN85], *Clouds* [DLAR91], *Arjuna* [SDP91], *Apertos*[12] [YTM+91], *Venari/ML* [HKM+94], *Karos* [GCLR92] and *Hermes/ST* [FHR94].

Nevertheless, Argus had serious drawbacks. Due to limited personnel, many well known and obvious optimizations were not implemented and therefore, the overall system performance was poor. One goal of the Camelot/Avalon project was to provide the same ease-of-programming advantages as Argus but with acceptable performance. Camelot/Avalon was carefully designed for this purpose and all known optimizations to standard protocols were implemented. This lead Gray and Reuter to state that "Camelot can be taken to be the first proven implementation of nested transactions as a general facility" [GR93].

Research prototypes like Argus and Camelot/Avalon have matured nested transactions and object technology in distributed systems so that, today, this technology is applied to large-scale commercial systems and the number of these systems is growing rapidly. Examples are ANSA [Arc91], ObjectStore [Obj], Versant [Ver], Encina[13] [Tra91], KALA [Pen], PCTE [WN93] and FORTE [For].

[12] Apertos has formerly been called "*Muse*".

[13] Encina is based on the C programming language but is currently extended to provide object support [Dix94].

Chapter 3

The Hermes/ST Distributed Programming Environment

This chapter presents some linguistic constructs of the Hermes/ST distributed programming environment. It is in part based on [FHR94]. As in Argus, one goal of Hermes/ST is to make distributed programming *easier*. Another important goal is to facilitate the development of *efficient* programs. Both goals are approached through strict *separation of concerns* via *parameterization*. For example, the volatility or persistence of objects is not a class property but an instance property, specified via parameters of the instance creation. Transaction semantics are not specified explicitly in method code but rather as parameters of the method invocation. This parameterization supports the general advantages of object-orientation as described in Section 2.4.2: reusability, extensibility and maintainability. These advantages are discussed individually for the different linguistic constructs introduced in this chapter.

This chapter is structured as follows. Section 3.1 presents an example application, a distributed bank, which is used throughout the chapter for demonstration of the linguistic constructs. Section 3.2 presents the Hermes/ST object model. The generalized message scheme is introduced in Section 3.3. Section 3.4 deals with concurrency control. Then, a complete implementation of a distributed bank application is described in Section 3.5. Finally, in Section 3.6, the linguistic features of Hermes/ST are evaluated and compared against the linguistic features of other object-oriented distributed programming environments that support nested transactions.

3.1 The Distributed Bank Example

The distributed bank has often been used as a test application for distributed programming environments [Lis88, EME91, Hew91]. The example described in this section is derived from the banking system in [Lis88].

An electronic international bank is composed of *branches* and *tellers*, which are geographically distributed. Each branch and teller can communicate with any branch. Each branch stores a collection of *accounts*. Accounts are identified by their *branch code* and an *account name*, and are either *cheque* or interest bearing *savings* accounts. Tellers are used to open and close accounts, deposit, withdraw and (internationally) transfer money. A special teller, the *main office*, has knowledge about all branches in the bank, and provides special managerial functions such as conducting audits. Other teller types are *automatic teller machines* and *bank clerks* which represent the computer interfaces of human tellers.

3.2 The Hermes/ST Object Model

The Hermes/ST object model is inspired by the Smalltalk object model and, in fact, has been implemented in Smalltalk. However, it does not depend on any particular feature of Smalltalk and could as well be implemented in any other object-oriented language.

Hermes/ST classes are defined in a single inheritance hierarchy with a single root called "`HermesObject`". A set of special classes are called "*constants*". They include numbers, characters, strings and dates. Instances of constants are *immutable*, i.e. none of their methods change their internal states.

Class descriptions specify the variables and methods of instances, *Hermes/ST objects*. *Uniform reference semantics* [Mey88] is used for accessing objects, i.e. objects are always referred to via pointers (so-called "`HermesPointers`") but are never contained by other objects. Thus, the Hermes/ST object model is *fine-grained*[1] [CC91]. The state of an object is determined by the objects its variables refer to. Two kinds of variables are distinguished: *named variables* and *indexed variable*, which are, for example, used for arrays. Named variables must be accessed through specific read and write *access methods*, e.g. `accountName` (read access) and `accountName:` (write access) for a variable named `accountName`[2]. The methods `at:` (read access) and `at:put:` (write access) are used to access indexed variables.

Different objects may reside on different nodes in the network but every particular object resides on only one node. Object replication and migration is (currently) not supported. Objects communicate via message passing. Sending messages to other objects is location-transparent. The Hermes/ST message scheme is described in the next section.

Two kinds of objects are distinguished: *volatile objects* and *transactional persistent objects*, simply called "*persistent objects*". Every class can be instantiated as both volatile and persistent objects. Volatility and persistence henceforth refer to the *kind* of an object. Similar to Smalltalk, instance creation is performed via a *class method* in Hermes/ST. The kind of an object is specified as an *instance creation parameter*. `instantiate:#volatile` returns the reference to a new volatile object. `instantiate:#persistent` returns the reference to a new persistent object. The location (if different from the local node), a symbolic alias (which is registered with a name server), and other features can be specified via additional instance creation parameters.

Persistent objects support transactional semantics. They perform concurrency control and recovery. Persistent objects have a mirror image of their *persistent state* on permanent storage. The persistent state of an object includes variables referring to constant objects and other persistent objects but not variables referring to volatile objects. A typical example of a volatile object referred to by a persistent object is a window, e.g. a graphical display of a persistent object representing a bank account. On permanent storage, variables referring to constant objects are stored by value, variables referring to persistent objects are stored by reference, and variables referring to volatile objects are replaced by nil pointers. Special code can be specified to initialize volatile objects referred to by a persistent object when the persistent object is activated in memory (e.g. after a node crash).

Volatile objects support no transactional properties. They are not concurrency con-

[1] The Hermes/ST object model has recently been extended to allow objects to contain other objects via *nested encapsulation* [Faz94]. This allows objects of various granularities: fine-grained, medium-grained, and large-grained. A description and analysis of this scheme is beyond the scope of this thesis.

[2] Smalltalk programmers may wish to note that specific access methods are provided for all Hermes/ST classes via a set-up routine. Redefining the semantics of the assignment and instance variable read access would have provided cleaner syntax. However, this would have required modifying the Smalltalk compiler which is beyond the scope of the Hermes/ST work.

trolled, perform no recovery and have no mirror image on permanent storage. Volatile objects are typically used for volatile aspects of persistent objects (e.g. a window), temporary variables, message parameters and return values. Since volatile objects exhibit better performance characteristics than persistent objects they are typically used in all cases where the integrity of data is not essential. Like non-atomic objects in Argus, volatile objects can be used to explicitly defy serializability and have transactions communicate non-committed data if this is required in special circumstances.

Appendix A.1 presents the Hermes/ST code for the classes `Tree` and `TreeNode` which implement the abstract data type of a *binary search tree* [Knu73]. A binary search tree is a binary tree where the contents of every node (i.e. the elements of the tree) can be compared and are in the following relationship: for every node, all elements in the left subtree are less than the node contents itself and all elements in the right subtree are greater than the node contents. There are no two nodes with the same contents in the tree. Traversing the tree in pre-order results in a sorted list of all elements with the smallest element first and the largest elment last. The binary search tree is used by branches of the distributed bank to efficiently store accounts, sorted according to their account number. The classes `Tree` and `TreeNode` specify the definition of both volatile and persistent binary search trees.

A binary search tree represents a sorted collection of items. Class `Tree` is therefore descended from classes `HermesCollection`, `HermesSequenceableCollection` and `HermesSortedCollection` (in root-to-leaf order). Its ancestor classes define a complete interface for general collections, sequenceable collections and sorted collections. The interface includes methods for enumerating all collection elements, such as, `do:`, `collect:` and `select:`, methods for finding elements, such as `detect:`, and methods for printing a collection, such as `printString`. `Tree`'s responsibility is simply to add some basic methods which support the complete interface. These methods include `do:` for enumerating all tree elements, `add:ifExisting:` for adding new elements and `remove:ifAbsent:` for removing elements.

The class definitions and documentation and the implementation of the methods can be found in Appendix A.1. They represent a short and elegant text-book style implementation of binary search trees. The fact that these classes can be used to instantiate transactional persistent objects does not add to the complexity of the implementation. `hermesSelf` refers to the hermes object receiving a particular message. It is analogous to `self` in Smalltalk. Flexibility of the instantiation is achieved in method `add:ifExisting:` by using `hermesSelf kind`. Method `kind` returns the kind of an object, either `#volatile` or `#persistent`. Whenever a new element is added to the search tree, objects of the same kind as their tree parents are created. This ensures that if a volatile empty tree is created via `Tree instantiate:#volatile` then every added tree element will be volatile. Conversely, if a persistent empty tree is created via `Tree instantiate:#persistent` then every added tree element will be persistent. Such a persistent search tree provides all transactional properties. Particularly, it is implicitly concurrency controlled and supports a high degree of concurrency as described in Section 3.4. `remove:ifAbsent:` explicitly deletes the tree nodes it removes. This is because automatic garbage collection, although performed for volatile objects by the Smalltalk system, is not supported for persistent objects in the current version of Hermes/ST.

3.2.1 Development Advantages

3.2.1.1 Reusability

The binary search tree is a good example for the kind of software reuse facilitated by the Hermes/ST object model. Classes or sets of classes that are useful in different contexts—sequential, non-transactional programming and distributed transactional programming—can be defined once and used in these various contexts. This is facilitated because volatility and persistence are not class features but features of individual instances. This is achieved via parameterization of the instance creation.

3.2.1.2 Extensibility

The Hermes/ST object model also supports extendibility which makes the system particularly well suited to incremental development. This approach was used successfully in the implementation of the distributed bank, described in Appendix A.5 and various other, much larger projects[CCM+93, RHR+93].

The development strategy is as follows. After completing the design of the distributed application, a *single-machine sequential prototype* of the application is first implemented using volatile objects. Since this implementation presents a centralized prototype of a distributed design, distributed aspects can be implemented as well. This prototype is debugged, and the design is at least partially validated. Detection and removal of design and implementation errors, many of which are not directly related to the distributed, concurrent or fault-tolerant nature of the application, are performed. The debugging/design validation process at this stage is greatly eased because it is performed on a single machine without concurrency, distribution and its potential problems (see Section 2.1).

This validated prototype is then extended. Implicit concurrency control, recovery and permanence are added by changing instantiation parameters from `#volatile` to `#persistent`. Structural changes to the code, and the errors that these tend to introduce, are avoided through Hermes/ST's parameterised instantiation approach. After testing of this new prototype, distribution can be added likewise, or explicit concurrency and fault tolerance properties can be added to the application (see Sections 3.3 and 3.4).

The implementation of the distributed bank example and its validation was completed in a few days. In particular, the implementation of the binary search tree classes and their validation was performed within a few hours.

3.2.1.3 Maintainability

Separation of concerns increases the maintainability of objects. Changing the kind of an object from `#volatile` to `#persistent` or vice versa does not affect the functional behaviour of the object. All that changes is the performance and reliability characteristics of the object. This means that a change in the kind of an object is localized to the object itself and does not affect other objects, which is advantageous in terms of maintainability.

3.3 The Generalized Message Scheme

3.3.1 Message Kind and Transaction Parameters

Hermes/ST objects communicate via passing messages. Message arguments and return values are generally passed by reference. Only constant objects are passed by value since they are immutable. Methods can access (i.e. read and write) the receiver's variables and can, in turn, invoke other methods. Three kinds of messages are supported: *synchronous*,

asynchronous and *wait-by-necessity* messages. They are henceforth referred to as the *kind* of a message.

Synchronous: In a synchronous message, the sender is always suspended until the receiver has finished execution and has returned the message result[3].

Asynchronous: An asynchronous message creates a new thread of control that executes concurrently with the sender's thread. The sender is not suspended and is not returned a message result.

Wait-By-Necessity: A wait-by-necessity message[4] is a mixture between a synchronous and an asynchronous message. The message creates a new thread of control that executes concurrently with the sender's thread. As in the asynchronous case, the sender's thread is not suspended. However the wait-by-necessity message does return a result immediately after invocation—or rather a placeholder for the actual message result called a "*voucher*"[5]. The actual result is eventually returned into the voucher and can then be used by the sender. If the sender attempts to use the result before it has been returned, the sender is suspended until the result is returned.

Every kind of message can create a transaction. Hermes/ST then ensures the transactional properties for the execution of the message itself and all messages that it sends, directly or indirectly via other messages. When a message that creates a transaction sends other messages that create transactions then Hermes/ST ensures nested transaction properties.

The three kinds of messages and the fact that each message can create a transaction provides six types of messages: synchronous messages that do or do not create transactions, asynchronous messages that do or do not create transactions and wait-by-necessity messages that do or do not create transactions. All these six types can be arbitrarily mixed and nested. For example, a synchronous transaction creating message may send an asynchronous non-transaction creating message which, in turn, may send a wait-by-necessity message that creates a (nested) transaction. All asynchronous and wait-by-necessity messages can execute concurrently with their sending threads[6], regardless of whether they create transactions or not. This allows, for example, sibling transactions and ancestor and descendent transactions to execute concurrently. Chapter 4 defines the semantics of such messages.

3.3.2 Specification of Message Parameters

In all object-oriented languages, messages are specified by the *receiver object*, the *method name* and the *method arguments*. For example, in the Smalltalk message `branch deposit: amount to:account`, the receiver is `branch`, the method name is `deposit:to:` and the arguments are `amount` and `account`. In order to allow the specification of the three kinds of methods, transaction creation and other message properties, Hermes/ST extends the

[3] As in Smalltalk, the case of a synchronous *procedure call*, i.e. a synchronous message where the sender is not interested in the result, is not handled explicitly. The application programmer can in this case return a dummy result, e.g. nil.

[4] There are various terms used for this concept in concurrent and distributed programming. The term "wait-by-necessity" was introduced by Caromel [Car90]. Other examples are "implicit futures" [Hal85] or "FUTURE" [Lie87], "HURRY" [YT87] and "future type message passing" [YSTH87].

[5] Again, other terms have been used, including "awaited object" [Car90], "implicit future" [Hal85], "future variable" [GCLR92] and "CBox" [YT87].

[6] This is provided there are no conflicting data accesses.

standard message specification scheme to allow optional *message parameters*. Message parameters are conceptually different from the arguments of a message. They describe properties of a message, i.e. a method *invocation* rather than the properties of a method itself. Syntactically, the message parameters are specified between the receiver and the method name, separated by semicolons[7]. If no message parameters are specified then defaults are assumed. For example, `branch deposit:amount to:account` describes a synchronous message that does not create a transaction. In contrast, `branch asynchronously; transactionCreating; deposit:amount to:account` describes an asynchronous message that creates a transaction.

3.3.3 The Weighted Voting Example

A good example for the usefulness of the various types of messages is the implementation of Gifford's weighted voting for replicated objects [Gif79]. In the bank example, replication is used for daily interest and exchange rates which are replicated at every branch for high availability. Gifford's mechanism is used to ensure consistent updates. The Hermes/ST code can be found in Appendix A.2.

The implementation uses methods for concurrently enumerating collections, namely `doInParallel:`, `doInParallelAndWait:`, and `collectInParallel:` (see Appendix A.2.1). `doInParallel:` allows the sending of a number of asynchronous messages where the invoking thread is not suspended. `doInParallelAndWait:` is equivalent to the concurrent loop which Argus and Camelot/Avalon provide. A number of asynchronous messages are sent but the invoking thread is suspended until all messages have returned. `collectInParallel:` is a most useful generalization of the wait-by-necessity concept where a number of messages is sent in parallel. The sender of the messages then continues and can, at a later time, collect the results in order of their arrival. All three mechanisms can be used transactionally and non-transactionally and are implemented easily with the Hermes/ST message constructs.

For the implementation of Gifford's weighted voting, `collectInParallel:` is used to concurrently collect the required number of votes for reading or writing variables in a replicated object (Appendix A.2.2). The access methods (`read:` and `write:to:`) start collecting the incoming votes and test whether a quorum is reached. As soon as a quorum is reached, they continue execution, performing the actual read or write operations. `write:to:` uses `doInParallelAndWait:` for this task to make sure that all write operations have actually been performed. Votes arriving after the respective quorum has been reached can be handled in different ways. `read:` simply discards them. `write:to:` uses them to update out-of-date replicas. Since this update is not critical for the correctness of the write operation, it is performed using `doInParallel:`. See Appendix A.2 for class and method code and comprehensive comments.

Both access methods, `read:` and `write:to:` can be invoked with or without message parameter `transactionCreating`. If an access method or a method invoking an access method is specified to create a transaction, then Hermes/ST ensures transactional properties.

3.3.4 Additional Message Parameters

Messages that create a transaction can specify a range of additional parameters. They include `mode:`, `retries:` and `timeout:`.

[7] Smalltalk programmers may wish to note that this is an unusual application of the cascading construct (;). The reason for this choice is a compromise between the wish to specify message parameters in a concise way and the wish to avoid changing Smalltalk's syntax and hence Smalltalk's compiler.

- Two main transaction *modes* are distinguished: `abortIfFail` and `performIfFail`. `abortIfFail` specifies that an aborting subtransaction causes its parent transaction to abort. `performIfFail` specifies that an aborting transaction does not cause its parent transaction to abort—instead, a specified exception is executed.

- `retries:` allows the specification of how many times to retry a failed transactional message before it is aborted.

- In Hermes/ST, network, node and software failures are not distinguished. Furthermore, Hermes/ST does not prevent deadlocks. A *timeout* mechanism is used to detect deadlocks, software and hardware failures. The specification of timeout values can be critical for the overall performance of a system. Because of the dynamic nature of transaction nesting, it can be hard for a programmer to statically specify a timeout value for a message that creates a transaction. Therefore, Hermes/ST provides *accumulative timeouts*. Every transaction is assigned a timeout value that can be specified via the message parameter `timeout:`. Whenever a subtransaction starts, the parent transaction's timeout value is increased by the child's timeout value. Thus, timeouts accumulate over nested transactions. When a transaction's timeout value is exceeded, it fails, which may lead to a transaction abort, depending on the specified `mode:` and `retries:`parameters.

Another important Hermes/ST message parameter is the `lock:` parameter. `lock:` allows methods to be invoked using type-specific, user-defined concurrency control. Section 3.4.2 gives a description of such concurrency control specifications. Other message parameters are provided which are not discussed here. See [FHR93c] for details.

3.3.5 Specifying Invocation Parameters in Method Interfaces

Note that not all message parameters concern the receiver of a message. For example, the sender of a message is responsible for thread creation for asynchronous and wait-by-necessity messages and for retrying failed transactions. Transaction objects are concerned with messages that create transactions. The receiver object is concerned with lock parameters. Often, particular methods are always invoked with the same message parameters. For example, a distributed bank transfer is always invoked transactionally.

Hermes/ST allows message parameters to be specified as part of the public interface in the *definition* of a method[8]. The syntax is as follows. Between method header (consisting of method name and arguments) and method body (consisting of the statements), the message parameters are specified enclosed by double quotes, following the class name `MessageParameters` and separated by semicolons[9]. Example:

```
transfer: amount from: account1 to: account2
  "MessageParameters transactionCreating; timeout: 2"
...method body...
```

In this example, every invocation of method `transfer:from:to:` creates a transaction with timeout value of 2 seconds unless specified otherwise.

Recall that Hermes/ST classes are defined in an inheritance hierarchy. Message parameters can be specified for all methods of Hermes/ST classes. When methods are

[8] Note that a client object invoking a method on a server object knows the method's public interface.

[9] Smalltalk programmers may wish to note that a special message parameter compiler has been implemented that runs over method comments. This way of specifying message parameters does not require a change of the Smalltalk method declaration syntax and therefore a modification of the Smalltalk compiler.

overridden in subclasses, all message parameters specified by ancestor classes are *inherited individually* and can be *overridden individually*. Message parameters for a particular method that are not explicitly specified in the method definition and are not explicitly specified in the definition of the method in any ancestor class are determined by a *default value*. The default for the message kind is `synchronous`, the default for `lock` is `NoLock` (a lock type which does not conflict with any other lock type), the default for transaction creation is `nonTransactionCreating`, the default for transaction mode is `#abortIfFail`, the default for `retries` is `0` and for `timeout` is `1` (second).

The public interface of Hermes/ST methods conceptually includes the values for all message parameters, determined either by explicit specification, inheritance or default values. Clients that invoke a Hermes/ST method may override message parameters specified in its interface. So, the precedence for message parameters is as follows. Parameters specified at method invocation override parameters specified at method definition. Parameters specified in method definitions of descendant classes override parameters specified in definitions of ancestor classes. Parameters specified in the definition of classes override defaults.

See the example of a transfer method and auxiliary withdraw and deposit methods in Appendix A.3. The methods `deposit:to:` and `withdraw:from:` of class `Branch` are specified to create a new transaction when invoked. By default, invocations of `deposit:to:` and `withdraw:from:` are synchronous. This is because the semantics of the deposit and withdraw operations require that they be performed synchronously and create a transaction when invoked from a teller.

Method `transfer:from:name:to:name:` of class `Teller` invokes the deposit and withdraw methods but it overides two of its message parameters at invocation. The transfer method itself creates a transaction, as specified by the transaction parameter in its definition. For performance reasons outlined in the next section, the transfer method invokes the deposit and withdraw methods asynchronously and non-transaction creating. Thus, the message parameters specified in the public interfaces of the deposit and withdraw methods are overridden in two ways. The message kind parameter is changed from its default value `synchronously` to `asynchronously` and the transaction parameter is changed from the parameter specified at definition, `transactionCreating`, to `nonTransactionCreating`.

The transfer method in class `AutomaticTellerMachine` inherits all message parameters specified in class `Teller`. It can override individual parameters. In this case, the `timeout` parameter is changed to 2 seconds (see Appendix A.5.3).

3.3.6 Development Advantages

3.3.6.1 Reuse

By separating message parameters from method code, the Hermes/ST generalized message scheme supports convenient reuse of methods in various contexts. Examples are the withdraw and deposit methods, which create a transaction when invoked directly from a teller, and do not create a transaction when invoked from within a transfer operation.

3.3.6.2 Extensibility

The Hermes/ST generalized message scheme supports an incremental development strategy for reliable distributed systems particularly well. A system developer can design methods with transactions in mind but implement them non-transactionally first. These non-transactional methods are easier to debug since no underlying transactional system

masks software failures. After functional validation of these non-transactional methods, transactions can arbitrarily be put in place where data integrity is important. This process only requires changing message parameters—no structural changes need to be made. The transactional system can be tested, its performance can be monitored and bottlenecks can be detected. Since transactions are expensive, fine tuning may need to be performed to resolve bottlenecks.

One way of decreasing transactional expense is to cut down transactional nesting depth where possible. Consider the transfer example above. Note that `transfer:from:name:to:name:` is always invoked transactionally and the whole transaction should abort if either the withdraw or deposit operation fails. Further note that the transfer transaction is relatively short so that the level of recovery introduced by nested transactions is not necessary. Therefore, for performance reasons, the withdraw and deposit operations are not performed as subtransactions. See the performance figures presented in Section 5.7.

Another way of decreasing transactional expense is to increase concurrency. The transfer method, again, serves as an example of this. One can combine both approaches, cutting down transactional depth and increasing concurrency, due to the separation of transaction and thread semantics in Hermes/ST. The way message parameters can be specified at method definition and overridden at declaration makes this fine-tuning step relatively easy.

For longer transactions, the probability of success can be increased by using nested transactions. `retries:` and `performIfFail` allow parent transactions to continue when subtransactions fail. Transient failures and deadlocks can be managed through retries[10]. Longer failures can be managed by specifying appropriate compensating actions using `performIfFail`.

3.3.6.3 Maintainability

Maintainability is increased by the strict separation of concerns that Hermes/ST provides. Changing individual message parameters does not affect other parameters. Take, again, the transfer implementation as an example. Individual changes of the message kind and transaction parameters of the deposit and withdraw messages do not affect the functional behaviour of the transfer method. This allows localized changes to methods which is advantageous for maintainability.

3.4 Concurrency Control

3.4.1 Implicit Concurrency Control

The easiest way for an application developer to prescribe concurrency control in a Hermes/ST application is to use system-defined *implicit locking*. Hermes/ST methods do not have to be specified as "readers" or "writers". Furthermore there is no need for dedicated lock acquisition code to be included in the specification of a method[11].

Implicit locking has been implemented in Hermes/ST via a mechanism called "*minimal locking*" [FHR93b]. Minimal locking acquires read/write locks before accesses to individual persistent object variables. Lock acquisition is performed automatically by the Hermes/ST system. Lock release is also performed by the Hermes/ST system, either immediately after the access (for non-transactional messages) or at transaction commit and abort (for transactional messages). In combination with Hermes/ST's small-grained

[10] A more effective way of combating deadlocks is described in Section 3.4.2.

[11] When and if such code is needed, it can, however, be specified. See Section 3.4.2.

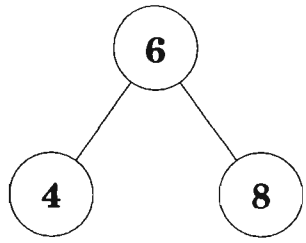

Figure 3.1: An example binary search tree.

object model (see Section 3.2), minimal locking always ensures correct locking. All data items read are read locked and all data items written are write locked. Minimal locking always locks the minimal amount of data accessed—hence its name. Minimal locking achieves what is termed "maximal concurrency" in [FHR93b]. Providing "maximal concurrency" can be expensive in terms of time (for the acquisition and release of locks) and space (for lock objects). Therefore, minimal locking has recently been refined to a variable locking mechanism that allows implicit concurrency control on a coarser grain [Faz94]. This coarser-grain locking decreases concurrency but it also decreases scheduling expense and the probability of deadlocks. A discussion of this scheme is beyond the scope of this thesis.

The code for the binary search tree, introduced in Section 3.2, demonstrates implicit locking (recall Appendix A.1). When class `Tree` is instantiated as a persistent object, then all instances of `Tree` and `TreeNode` are persistent and concurrency controlled. Implicit locking allows concurrent "add" and "remove" operations to different parts of the tree. Consider the example tree in Figure 3.1 containing values 4, 6 and 8. Insertions of the values 2 and 5 can be performed concurrently, even if they belong to different transactions, since they affect different parts of the tree. More concretely, the insertion of value 2 read locks the `left` variables of nodes 6 and 4 and only write locks the `root` variable of the left subtree of node 4. Insertion of value 5 read locks the `left` variable of node 6 and the `right` variable of node 4 and write locks the `root` variable of the right subtree of node 4. Thus, lock conflict does not occur (see Table 2.3).

The same is true for concurrent removal of the value 4 and an insertion of the value 7. However, the removal of the value 4 and the insertion of the value 1 cannot be performed concurrently since both operations modify the same part of the tree. More concretely, removal of value 4 write locks the `root` variable of the left subtree of node 6. Insertion of value 1 attempts to write lock the same variable. A lock conflict occurs (see Table 2.3). Implicit locking delays one of the requested operations until after the other operation has finished (for non-transactional messages) or its transaction has committed or aborted.

3.4.2 Explicit Concurrency Control

Hermes/ST explicit concurrency control is achieved through the *programmable lock approach* [FHR93b]. In the programmable lock approach, type-specific concurrency control is defined in the class specifications of *programmable locks*. Programmable locks form a hierarchy with the abstract class `ProgrammableLock` as the root. Hermes/ST provides a set of system-defined programmable lock classes. They include classes for mutual exclusion, traditional read/write locking, fair read/write locking and bounded buffer synchronization.

The class `ProgrammableLock` defines two methods, `isSchedulable:` and `isCompatibleWith:`, which return boolean values, in this case `true` (see Appendix A.4.1). These methods can be overridden by subclasses. The method `isSchedulable:` allows a pro-

grammable lock to make scheduling decisions on the basis of persistent object state. The method `isCompatibleWith:` defines a programmable lock's "compatibility" with other programmable locks.

Programmable locks are associated with Hermes/ST methods via the `lock:` message parameter (see Section 3.3) and are instantiated when a persistent object receives a message. Arguments can be passed to `lock:` which are stored as internal variables of the `ProgrammableLock` object and thus can be used by `isSchedulable:` and `isCompatibleWith:`. Arbitrary objects can be passed. However, two particular types shall be mentioned here. They are the *message arguments* and *guard methods.*

3.4.2.1 Passing the Message Arguments to a Programmable Lock

Consider the example of programmable lock class `AccountWriteLock` which is subclassed from `WriteLock` (see Appendix A.4.2 and A.4.3). `AccountWriteLock` is the lock parameter specified at the definition of method `deposit:to:` in class `Branch`. The lock association in `deposit:to:` specifies that the argument `accountName` is passed to `AccountWriteLock`. This means that whenever a `Branch` object receives a `deposit:to:` message, an instance of `AccountWriteLock` is created and the actual argument `accountName` is stored as one of its internal variables. `accountName` is used by `AccountWriteLock`'s `isCompatibleWith:` method to test whether `otherLock` refers to the same account as the lock itself. `AccountWriteLock` weakens the compatibility predicate of its superclass `WriteLock` by invoking its `isCompatibleWith:` method (`super isCompatibleWith: otherLock`) and using a disjunction (`or:[self account ~= otherLock account]`). Logically, `AccountWriteLock` implements a write lock for an individual account rather than the whole branch. Lock compatibility is not tested by an individual `Account`'s concurrency controller but rather by the `Branch`'s concurrency controller. The usefulness of `AccountWriteLock` to avoid deadlocks is described in Section 3.4.2.3.

3.4.2.2 Passing Guard Methods to Programmable Locks

Guard Methods [Atk91] are read-only methods that allow programmable locks to inspect object state. Consider the example of the programmable lock class `SavingsAccountsWriteLock` which is subclassed from `WriteLock` (see Appendix A.4.4 and A.4.5). `SavingsAccountsWriteLock` is the lock parameter specified at the definition of method `addInterest` of class `Branch`. `addInterest` accesses all savings accounts of a branch to add any outstanding interest. `SavingsAccountsWriteLock` conceptually locks all savings accounts of a branch in write mode to allow `addInterest` to be performed without interference from other operations that modify savings accounts. `SavingsAccountsWriteLock` `isCompatibleWith:` checks the type of `otherLock`'s account (`#cheque` or `#savings`) using the guard method `typeCheckMethod`. This guard method is passed as a parameter to `SavingsAccountsWriteLock` in the `lock` message parameter specification of method `addInterest`.

Strictly speaking, passing `typeCheckMethod` to `SavingsAccountsWriteLock` is not necessary. It could have been hard-coded in its `isCompatibleWith:` method. However, parameterizing the type-check method increases the reusability of `SavingsAccountsWriteLock`, making it applicable for classes with different type-check methods.

3.4.2.3 Using Programmable Locks for Deadlock Avoidance

Hermes/ST implicit locking may cause deadlock if, for example, a branch-internal transfer operation from one savings account to another savings account interferes with an `add-`

`Interest` invocation. Associating `addInterest` with a `SavingsAccountsWriteLock` and associating `withdraw:from:` and `deposit:to:` (the two methods invoked in the transfer method) with an `AccountWriteLock` avoids such a deadlock. This is because `SavingsAccountsWriteLock` conceptually locks all savings accounts of a particular branch in write mode. A `SavingsAccountsWriteLock` is incompatible with every `AccountWriteLock` that controls the access to a savings account. Thus, in case of a conflict, the execution of one of the operations (`transfer` or `addInterest`) is delayed until after the other operation's transaction has committed or aborted.

3.4.3 Development Advantages

3.4.3.1 Reusability

The fact that concurrency control is not specified within method code allows implicitly concurrency controlled methods to be conveniently used in a non-concurrent and concurrent context. The binary search tree implementation of Section 3.2 serves as an example.

The Hermes/ST explicit concurrency control mechanism does not only support the reuse of methods that are explicitly concurrency controlled. It also facilitates reuse of concurrency control specifications themselves.

- The association of programmable locks and Hermes/ST methods is separated from the method definition. This allows one to conveniently use a method in both a sequential and concurrent context.
- The concurrency control specification for a Hermes/ST class is *composable*: subclasses that add and/or override methods can individually add/change programmable lock associations. Composability is achieved by a combination of separating the programmable lock association from method definition and associating programmable locks with methods individually.
- Programmable locks are specified separately from the Hermes/ST classes in which they are applied. This allows a common concurrency control behaviour (e.g. mutual exclusion) to be applied in different classes where appropriate.
- Since programmable locks are defined in an inheritance hierarchy, concurrency control behaviour can be reused through *programming by difference*. Examples are the implementations of `SavingsAccountsWriteLock` and `AccountWriteLock`, which utilize the locking behaviour of their superclass `WriteLock` and weaken the compatibility predicate using a logical "or" operator.

3.4.3.2 Extensibility

Hermes/ST implicit locking allows the transition from (non-concurrent) volatile objects to fully concurrency controlled persistent objects without changing method definitions or adding concurrency control specifications. However, if it is necessary to add explicit concurrency control to an implicitly concurrency controlled Hermes/ST application, the incremental strategy still applies. First, simple system-defined programmable locks like mutual exclusion locks or read/write locks can be employed. Performance analysis of the simple concurrency controlled system may detect bottlenecks. These bottlenecks can then be alleviated by the introduction of more sophisticated application-specific programmable locks such as `SavingsAccountsWriteLock` and `AccountWriteLock`.

3.4.3.3 Maintainability

Separating the concurrency control specification from the functional specification of a method has advantages in terms of maintainability. Both aspects can be modified individually without affecting the other. Also, validation can be performed for each aspect individually.

3.5 Hermes/ST Implementation of the Distributed Bank

This section describes important classes and methods of the Hermes/ST implementation of the distributed bank. The code can be found in Appendix A.5.

Class Teller (Appendix A.5.1) is an abstract class with three subclasses `HeadOffice`, `AutomaticTellerMachine` and `BankClerk`. The class definition for `Teller` specifies three variables `name`, `currencyTable` and `interface`. `name` uniquely specifies a teller, `currencyTable` is used for international transfers as described below and `interface` refers to a window, a graphical user interface. For `AutomaticTellerMachine`, this is the interface that a bank customer uses at an automatic teller machine. For `BankClerk`, it is the interface that a bank clerk uses when serving customers. For `HeadOffice` it is the interface that administrators use in the head office of the bank. `interface` refers to a volatile object. When a node crashes, then user interfaces are lost. Windows are re-opened when the node comes up again. This is specified in special initialization code which is not included in Appendix A.5.1.

The method `transfer:from:name:to:name:` performs a traditional fund transfer as described in Section 3.3. The method `internationalTransferFrom:name:to:name:` implements a more complex international transfer operation that involves a currency exchange. This method is interesting since it uses all three message kinds, synchronous, asynchronous and wait-by-necessity. Every branch keeps a currency table in variable `currencyTable` for all traded currencies. This can be slightly out of date. A currency table which always keeps the exact current exchange rate can be remotely accessed at the head office. Assume that for small transfers, i.e. transfers that do not exceed a particular limit, the locally stored exchange rate can be used, whereas for large transfers, the exact rate must be used. In order to optimize the performance of the transfer method, the exchange rate request to the head office is performed concurrently with the amount request to the source branch—using a wait-by-necessity and a synchronous invocation. If the amount to transfer does not exceed the limit, then the actual transfer can go ahead without waiting for the exact exchange rate to be returned. The exact rate is only used when necessary. For performance reasons outlined in Section 3.3.6, the actual transfer is performed concurrently using asynchronous invocations without creating subtransactions.

The `HeadOffice` class (Appendix A.5.2) additionally provides methods for creating and deleting branches and tellers and to perform audits.

The `Branch` class (Appendix A.5.5) defines a variable `accounts` which is initialized to an empty persistent binary search tree in the `Branch` instance creation method (see the class protocol `instance creation`). All accounts contained in a particular branch are stored in `accounts`, ordered according to their `accountName`.

Methods like `deposit:to:` and `withdraw:from:` use an auxiliary method `lookUp:`. `lookUp:` descends the accounts tree to return a Hermes/ST object reference to the specified account. If the account cannot be found, `abortCurrentTransaction:` is invoked. In the case of a transactional invocation, this causes the current transaction to abort and the specified symbol `#noSuchAccount` to be passed to the client of the aborting transaction. In the case of a non-transactional invocation, an exception is raised. Methods

`deposit:to:` and `addInterest` are explicitly concurrency controlled using programmable lock classes `AccountWriteLock` and `SavingsAccountsWriteLock`, as described in Section 3.4.2. Methods `openAccount:` and `closeAccount:` allow new accounts to be opened or accounts to be closed.

The class `Account` (Appendix A.5.6) defines three variables `name`, `type` and `balance`. `name` uniquely identifies a particular account, e.g. via an account number. `type` distinguishes chequing from savings accounts. `balance` stores the current account balance. Methods for depositing and withdrawing money are provided.

3.6 Evaluation and Comparison to Other Approaches

3.6.1 Evaluation

The linguistic constructs introduced in this chapter integrate transactional and distributed features into an object-oriented language without compromising important features of object-orientation: reusability, extendibility and maintainability. The following comparison sections show that this is not the case for many existing object-oriented distributed systems supporting nested transactions.

Hermes/ST is a prototype implementation of concepts described in this chapter and elsewhere [Faz94, Ran94]. Its purpose is to test the validity of these concepts. Hermes/ST is implemented in ObjectWorks\Smalltalk-80 V4.1 [Par92] and is currently running on Sun SparcStations, connected via a local area network. It includes the Hermes/ST language extension to Smalltalk-80 and development tools like class browsers and a distributed debugger. It also includes the execution environment with a name server, concurrency controllers, transaction, communications, persistence and recovery handlers. For details see [FHR93c].

To test the validity of the concepts introduced in this chapter, a number of projects have been developed in Hermes/ST. A smaller project was the implementation of the distributed bank as an example application for [FHR94] and this thesis. The distributed bank was implemented by the author within a few days. Two larger projects developed in Hermes/ST are "*Universal Personal Telecommunications*" [CCM+93] that implements an advanced telecommunications service [CCI91] and a *reliable distributed name server* [RHR+93]. The projects were developed by six and five final-year computer science students respectively over one year. Both systems make extensive use of Hermes/ST's distribution and transaction facilities and provide comprehensive graphical user interfaces. The algorithms that have been used are, in part, based on [HF92a, HF92b].

In all projects developed in Hermes/ST, an incremental development strategy was used. The complete systems, including all user interfaces, were first implemented with volatile objects. They were tested and debugged and then presented to the respective clients. Clients were then able to suggest modifications that were taken into account at this stage. Persistence, distribution, concurrency control and transactions were then added successively. This step was performed by modifying instance creation and message parameters only. No structural changes to classes or methods were needed. All graphical user interfaces remained unchanged. Also, clients did not require any modifications to the systems at this stage.

Both stages of the projects, the development of the sequential, single-node prototype and the extension to the final system took about half of the total development time. The incremental development strategy was appreciated by the developers as a controlled way of building complex systems and was employed successfully for these experiments.

This chapter is only concerned with Hermes/ST's linguistic features, not their imple-

mentation or performance. Chapters 4 and 5 deal in part with these issues. Consequently, the following comparison of Hermes/ST with other object-oriented distributed systems supporting nested transactions addresses linguistic aspects only. Three systems are compared: the well-known systems Argus and Avalon/C++ whose linguistic constructs cover a large class of other systems, and Venari/ML, a relatively new system with a number of novel linguistic constructs.

3.6.2 Argus

Argus [Lis82] is a distributed object-based programming system that supports nested transactions. Argus is built on top of the CLU programming language [Lis81]. *Guardians* contain *atomic* or *non-atomic* objects. Atomic objects in Argus are analogous to persistent objects in Hermes/ST; non-atomic objects relate to volatile objects. Since object kind is not an instance property, a data type like a binary search tree must be implemented twice when it is to be used in a transactional and non-transactional context. Since Argus is object-based, code sharing via inheritance is not supported.

There are two ways in which transactions are created in Argus. Firstly, every handler call (i.e. invocation of a guardian's method) implicitly creates a transaction. Secondly, synchronous nested transactions can be created explicitly via the `enter action...end` construct. Only a limited form of thread creation is supported. Threads can only be created via a loop construct (`coenter...end`) for concurrent nested transactions. This construct suspends the parent transaction until all child transactions have committed or aborted. Thus, no ancestor/descendant concurrency between transactions is supported.

In contrast, Hermes/ST permits threads to be created independently of transactions. This allows non-transactional threads, transactional threads that do or do not create subtransactions, sibling and ancestor/descendant concurrency in transactions. Argus' limited transaction/thread model makes it difficult to implement concepts like voting, where a thread creates a number of new threads to collect votes concurrently but continues immediately after the required number of votes has been obtained. The implementation in Hermes/ST is straight forward and allows the required amount of concurrency (see Section 3.3). In Argus, the same amount of concurrency can only be achieved by artificially making the vote counting thread a sibling of the voting threads. This has several disadvantages. Firstly, the code must be obscured in order to alleviate the deficiencies of the language. Secondly, sibling threads have to communicate, e.g. via shared variables. Thirdly, turning parts of the parent thread into a child thread changes the serializability semantics of the parent thread, as outlined in Section 5.2.

Apart from creating subtransactions from within transactions, Argus allows the creation of new top-level transactions from within transactions via the `enter topaction...end` construct. This is a convenient mechanism. However, it should be used with care since it allows non-committed transactions to exchange data and therefore may defy the transactional properties. Section 4.9 presents an extension to the linguistic constructs described in this chapter that not only allows top-level transactions, but also top-level threads and synchronous messages to be created from within a transaction. Like the `topaction` construct in Argus, it should be used with care for the same reasons.

Argus does not provide implicit concurrency control. Locks are acquired explicitly in method code via the `read_lock` and `write_lock` primitives. The system performs the release of locks at transaction commit and abort. In contrast, implicit locking in Hermes/ST is convenient since the programmer does not have to reason over concurrency control and the lack of concurrency control statements in the code increases reusability of methods in concurrent and non-concurrent contexts. It is also safe, since data is always

locked before being accessed. Furthermore, due to the maximal concurrency property, it often exhibits good performance. However, implicit locking in Hermes/ST may lead to deadlocks. It can exhibit poor performance due to a large number of lock acquisitions and does not always produce the optimal level of concurrency control for particular data types. Argus addresses the deadlock and performance issues via explicit read/write locking and the concurrency control issue via type-specific locking. In Hermes/ST, all three issues are addressed via the programmable lock approach. Type-specific locking in Argus [WL85] is more sophisticated than Hermes/ST's programmable lock approach in that it allows higher concurrency than strict 2PL. However, programming type-specific locks in Argus is complex [WL85]. Hermes/ST, on the other hand, does not attempt to leave the boundaries of strict 2PL. Rather, issues of convenience, composability, reusability, extensibility and maintenance are emphasized.

3.6.3 Avalon/C++

Avalon/C++ [EME91] is the distributed programming language built on top of the Camelot distributed operating system. Therefore, Hermes/ST's linguistic constructs are compared with Avalon/C++'s rather than Camelot's linguistic constructs. Avalon/C++ is an extension to the C++ programming language [Str86] and supports single inheritance, like Hermes/ST.

Analogous to guardians in Argus, Avalon/C++ defines *servers* that encapsulate objects. The kind of an object is a class property, determined via inheritance from one of three *base classes*: `recoverable`, `atomic` and `subatomic`. The instances of the three base classes are comparable to Hermes/ST's persistent objects. Instances of `recoverable` have a mirror image on permanent storage but do not perform concurrency control and abort recovery. `atomic` and `subatomic` are subclassed from `recoverable` and hence inherit its properties. In addition, they add concurrency control and abort recovery so that they ensure transactional properties. `atomic` allows a quick and convenient way to define new transactional objects, while `subatomic` provides primitives to give programmers more detailed control over the objects' synchronization and recovery mechanisms.

Hermes/ST currently only provides an equivalent to instances of the `atomic` base class: persistent objects. An extension to allow more kinds of objects, e.g. persistent only objects, is currently being developed [Ran94]. Since object kind is a class property in Avalon/C++, abstract data types like the binary search tree, must be implemented several times when used in several contexts: non-recoverable, recoverable, and atomic. The introduction of multiple inheritance in Avalon/C++ could alleviate this problem. Subclasses of tree classes could then multiply inherit from the respective base classes to add the required behaviour. A drawback of this approach is that for every kind supported, a new subclass must be created.

Avalon/C++ provides a richer transaction/thread model than Argus does. It allows the creation of synchronous nested transactions (via `start transaction{...}`), concurrent transactions (via `costart{transaction...}`) and the creation of top-level threads and top-level transactions (via `toplevel`) like in Argus. Additionally, it allows concurrent threads within transactions (via `costart{...}`). In the `costart` construct, the invoking thread is suspended until all invoked threads have finished, i.e. ancestor/descendant concurrency is not provided. In contrast, Hermes/ST allows both siblings and ancestor/descendant concurrency. Non-transaction creating threads in Avalon/C++ are not serialized. In contrast, Hermes/ST allows both serialized and non-serialized non-transaction creating threads (see Chapter 4).

Like Argus, Avalon/C++ only supports explicit lock acquisition in method code via

the `read_lock()` and `write_lock()` methods of class `atomic`. The `subatomic` class is a starting point for classes with type-specific concurrency control. The mechanisms for type-specific concurrency control in Avalon/C++ are more sophisticated than those in Hermes/ST in that they allow the implementation of objects with higher concurrency. Via inheritance, concurrency control specifications can be reused in different subclasses. However, since concurrency control is specified within a class, it cannot be applied to classes that belong to different inheritance hierarchies. The separation of concurrency control specifications from classes and methods in which they are used in Hermes/ST allows higher reusability, extensibility and maintainability of concurrency control specifications than does Avalon/C++.

3.6.4 Venari/ML

Venari/ML [NW91, WFMN92, HKM+94] is a concurrent, functional programming system, supporting nested transactions, that has been developed at Carnegie Mellon University. Venari/ML is neither distributed nor object-oriented. The novel linguistic constructs for specifying transactions and threads however do support reusability, extensibility and maintainability, and are worth comparing to Hermes/ST.

Venari/ML is implemented on top of the SML functional programming language [MTH90]. Like Hermes/ST, Venari/ML allows transactions to be specified independently from threads. Transactions and threads are specified over function calls via the higher order functions `transact` and `fork`. This scheme is similar to Hermes/ST's generalized message scheme and hence provides the same flexibility. Transactions can create synchronous and asynchronous subtransactions and can create non-transaction creating threads. Such threads can be either serialized or non-serialized. Sibling concurrency as well as ancestor/descendant concurrency is supported.

In addition, Venari/ML supports the separation of the transactional properties serializability, atomicity and permanence. Thus, threads can be specified to exhibit only some of the three properties. The specification of all three properties provides full nested transaction semantics. As with full transactions, such weaker transactions are specified via higher order functions that are applied to function calls.

This separation of transactional properties allows more sophisticated fine-tuning of applications. As in Hermes/ST, changing transactional specifications of function calls changes only their performance and reliability characteristics but not their functional behaviour in the absence of failures.

Venari/ML is the only system, of which the author is aware, that provides similar support for reusability, extendibility and maintainability in terms of transaction and thread specification, as Hermes/ST does. However, there are major differences between Venari/ML and Hermes/ST in terms of the semantics and implementation of transaction/thread scheduling. Section 5.5 presents details.

Chapter 4

Scheduling in a Generalized Transaction/Thread Model

Chapter 4 represents the core of this thesis. This is reflected in its size relative to other chapters and the fact that it is titled like the thesis itself. In this chapter, novel scheduling semantics are defined for the generalized message scheme. An implementation-independent schedulability predicate is presented that satisfies the scheduling semantics. Furthermore, an efficient implementation of the schedulability predicate is described. A simpler version of this work has been published in [Hum93]. The correctness of both the schedulability predicate with respect to the scheduling semantics and the algorithms with respect to the schedulability predicate are discussed. However, no formal proofs are given. Instead, the concepts and their justifications are explained in an intuitive way. A large number of figures and examples supports this approach. This is also true for definitions. Definitions are only formal where necessary. They are informal if the intuitive meaning is clear. Definitions for transactional properties are not repeated in this chapter. Rather, references to their introduction in Chapter 2 are given.

Although the correctness analyses are not formal, they are rigorous and very comprehensive. More than twenty pages of this chapter are devoted to correctness discussions. Readers that are solely interested in the mechanisms can safely skip these sections without missing information that is necessary for the understanding of the following sections.

Chapter 4 is structured in the following way. Section 4.1 presents all definitions necessary for the understanding of the mechanisms, described in this chapter. Section 4.2 defines the scheduling properties for the generalized message scheme. The schedulability predicate is defined in Section 4.3 and its correctness is analyzed in Section 4.4. A general design for the implementation of the scheduling mechanism is described in Section 4.5. Efficient algorithms for the schedulability predicate and their correctness are discussed in Section 4.6. The last three sections describe useful extensions to the generalized message scheme. The introduction of wait-by-necessity messages is performed in Section 4.7. Scheduling for non-serialized transactional threads is described in Section 4.8. Finally, Section 4.9 describes an extension that allows sending top-level messages from within nested messages.

4.1 Definitions

4.1.1 Messages and Message Trees

In this section and following sections, a subset of the generalized message scheme, introduced in Section 3.3, is defined more formally. Since this chapter and this thesis are

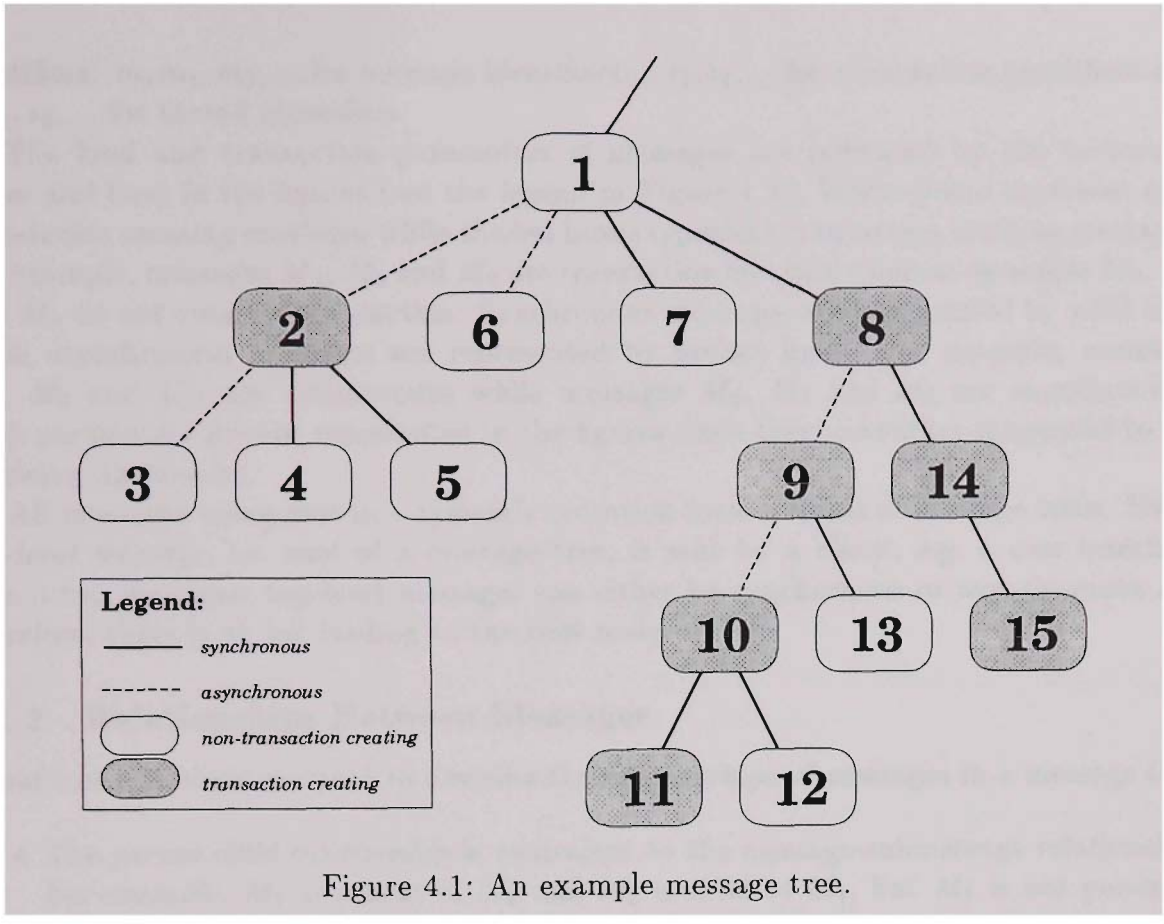

Figure 4.1: An example message tree.

mainly concerned with scheduling issues, only three message parameters are included: the parameters describing the message kind, transaction characteristics and lock specification. To simplify the concepts presented in this chapter, only two message kinds are taken into account first: synchronous and asynchronous. Section 4.7 presents an extension to include wait-by-necessity messages.

A *message*[1] is specified by a *receiver* object, *message parameters*, a *method name* and *arguments*. Message parameters describe the *kind* of a message (either *synchronous* or *asynchronous*), its *transaction characteristics* (*transaction creating* or *non-transaction creating*) and its *lock type*. Every message can *access* (read and write) the receiver object's variables and send other messages, either to the receiver object or other objects. Messages can be described as nodes in a *message tree* where the arcs represent *message-submessage relationships*, i.e. the relationships between messages and the messages they send.

See Figure 4.1 for an example message tree which is referred to throughout this chapter[2]. In this figure and all other figures of message trees, the following notations are used.

Messages, the nodes of the tree, are represented by boxes. Message-submessage relationships, the arcs of the tree, are represented by lines. For example, the messages labeled with 2, 6, 7 and 8 are submessages of the message labeled with 1 (the root of the tree). Boxes are numbered. Such a number can be used in various contexts for different concepts. Prefixed by an upper case letter M, it represents a *message identifier*, prefixed by an upper case letter T it represents a *transaction identifier* and prefixed by an upper case letter S it represents a *thread identifier*. For example, the root node represents message M_1.

Lower case letters are used to denote placeholders for message, transaction and thread

[1] All definitions are emphasized by italics.

[2] In order to avoid going back to Figure 4.1 for numerous examples, a loose page with this figure is inserted for the reader's convenience at the end of this thesis.

identifiers: $m, m_1, m_2, ...$ for message identifiers, $t, t_1, t_2, ...$ for transaction identifiers and $s, s_1, s_2, ...$ for thread identifiers.

The kind and transaction parameters of messages are indicated by the texture of boxes and lines in the figures (see the legend in Figure 4.1). White boxes represent non-transaction creating messages while shaded boxes represent transaction creating messages. For example, messages M_2, M_8 and M_9 are transaction creating whereas messages M_3, M_4 and M_5 do not create a transaction. Synchronous messages are represented by solid lines while asynchronous messages are represented by broken lines. For example, messages M_5, M_8 and M_{15} are synchronous while messages M_2, M_3 and M_6 are asynchronous. Lock parameters are not represented in the figures since they are rather tangential to the following discussions.

All messages being sent in a system's execution form a forest of message trees. Every *top-level message*, i.e. root of a message tree, is sent by a *client*, e.g. a user interface. Like other messages, top-level messages can either be synchronous or asynchronous and therefore, there is an arc leading to the root node.

4.1.2 Relationships Between Messages

Usual tree notations are used to describe the relationships of messages in a message tree.

- The *parent-child relationship* is equivalent to the message-submessage relationship. For example, M_1 is parent of M_2 and M_2 is child of M_1, but M_1 is not parent of M_3.

- The *ancestor* ($\leq$) and *descendant* ($\geq$) relationships are the transitive closures of the parent and child relationships. The ancestor and descendant relationships are reflexive, i.e. each message is its own ancestor and descendant. For example, $M_1 \leq M_3$ and $M_8 \leq M_8$ but $M_{15} \not\leq M_5$. Conversely, $M_{10} \geq M_8$ and $M_{14} \geq M_{14}$ but $M_9 \not\geq M_{11}$.

- *real ancestor* ($<$) and *real descendant* ($>$) are the non-reflexive counterparts of ancestor and descendant. For example, $M_2 < M_3$ and $M_9 < M_{11}$ but $M_{14} \not< M_{14}$. Conversely, $M_4 > M_1$ and $M_{15} > M_{14}$ but $M_2 \not> M_{14}$.

- Two messages are *incomparable* ($<>$) if they are neither in an ancestor nor descendant relationship. Any two messages belonging to different message trees are incomparable. Also, $M_3 <> M_6$ and $M_{10} <> M_{15}$ but $M_8 \not<> M_{13}$.

- Two messages m_1 and m_2 are *conflicting (m_1 conflicts with m_2)* if they have the same receiver object and their lock types are incompatible according to a lock compatibility matrix (see, for example, Figure 2.3 for the read/write lock compatibility matrix).

- A message m is a *common ancestor* of messages m_1 and m_2 iff $m \leq m_1$ and $m \leq m_2$. For example, M_1 is common ancestor of M_{11} and M_{12}, M_9 is common ancestor of M_9 and M_{13} but M_8 is not common ancestor of M_9 and M_1.

- For two messages m_1 and m_2 that have a common ancestor there is exactly one *least common ancestor message* m defined ($m = LCA(m_1, m_2)$). m is a common ancestor of m_1 and m_2 and for all other common ancestors m' of m_1 and m_2: $m' < m$. For example, $M_{10} = LCA(M_{11}, M_{12})$, $M_9 = LCA(M_9, M_{13})$ but $M_1 \neq LCA(M_{11}, M_{12})$.

4.1.3 Message Paths and Message Path Elements

A *message path* is a data structure that describes the parameters and the position of a particular message in a message tree. A message path is a non-empty sequence of *message path elements* that contain a message's identifier and its kind and transaction parameters. This sequence includes all messages from the root of a message tree down to the particular message. For example, $[M_9, synch, trans]$[3] is the message path element for message M_9 and $[M_1, synch, nonTrans][M_8, synch, trans][M_9, asynch, trans]$ is the message path for M_9. Every message path is unique in the entire execution of a system.

In order to simplify presentation, the message identifier (e.g. M_9) is used instead of its message path whenever the path is obvious from the context. This is the case for all examples in this chapter since they refer to figures. The message identifier can, in this case, be seen as an alias for the message path.

4.1.4 Regular Expressions for Message Paths

Special classes of message paths are described via regular expressions. The primitives to describe message paths are types of message path elements.

- *trans* stands for a transaction creating message path element (no matter whether it is synchronous or asynchronous). *nonTrans* stands for an non-transaction creating one.
- *synch* stands for a synchronous message path element (transaction creating or non-transaction creating). *asynch* stands for an asynchronous one.
- Both parameters can be combined with a dash, e.g. *synch-nonTrans* stands for a synchronous, non-transaction creating message path element. The other three combinations are used analogously.
- *any* stands for any message path element, synchronous or asynchronous, transaction creating or non-transaction creating.

Regular expressions are constructed by sequencing these elements. For example, M_9 matches the following regular expression: *synch synch asynch* since M_1 is synchronous, M_8 is synchronous and M_9 is asynchronous. Meta symbols are used to describe occurrence patters. Square brackets (*[]*) denote a group of *optional* elements, i.e. elements that occur either not at all or only once. A star (***) denotes an element to be repeated *arbitrarily*, i.e. any number of times (including 0).

Consider the example where m_1 and m_2 denote two message paths, i.e. m_1 and m_2 act as placeholders for sequences of message path elements. Then, the equation $m_2 = m_1$ *synch-nonTrans* [synch-trans any*]* denotes that m_2 starts with all elements of m_1. An arbitrary number of synchronous non-transaction creating elements may follow. Then, optionally, a single synchronous transaction creating element may follow, followed by an arbitrary number of elements of any type. For example, $m_1 = M_1$ and $m_2 = M_{13}$ match the description.

Since m_2 starts with all elements of m_1, m_1 is a prefix of m_2. This means that m_1 and m_2 are defined in the same message tree. This also means that all messages between the root of the message tree and m_1 are also between the root of the message tree and m_2. Therefore, m_1 is an ancestor of m_2 ($m_1 \leq m_2$)[4]

[3] *synch* stands for synchronous, *asynch* stands for asynchronous, *trans* stands for transaction creating and *nonTrans* stands for non-transaction creating. See also Section 4.1.4.

[4] The prefix relationship between ancestors and descendants motivates the $\leq$ notation, e.g. $[M_1, synch, nonTrans] \leq [M_1, synch, nonTrans][M_8, synch, trans][M_9, asynch, trans]$.

4.1.5 Transactions

Transaction creating messages form tree structures within message trees. The message tree of Figure 4.1 incorporates two such *transaction trees*, one with M_2 as the root and one with M_8 as the root. Transaction identifiers are generated with an upper case letter T and the number of the message that created the transaction, e.g. the transaction created by M_2 is called T_2 and the transaction created by M_8 is called T_8.

- Since each transaction is associated with exactly one message in a message tree, the relationships between messages defined in Section 4.1.2 ($<, \leq, >, \geq, <>$) can be extended to transactions. Two transactions are in one of the relationships if the messages that created them are in the same relationship, e.g. $T_8 < T_9$ since $M_8 < M_9$; $T_2 <> T_8$ since $M_2 <> M_8$.

- To simplify presentation, a transaction creating message and its transaction identifier are used interchangeably if it is clear from the context, which concept is meant. This also allows for mixed relationships between messages and transactions. For example, $T_2 < M_3$ since $M_2 < M_3$.

- A message is called "*transactional*" if there is at least one transaction creating message in its message path; otherwise it is called "*non-transactional*"[5], e.g. M_4, M_8 and M_{11} are transactional but M_1, M_6 and M_7 are non-transactional.

- A transaction t is the *top-level transaction of a message* m if t is the first[6] transaction creating message in m's message path[7]. One says "*m belongs to top-level transaction t*". For example, T_8 is the top-level transaction of M_{12}; M_{12} belongs to top-level transaction T_8.

- A transaction t is the *transaction of a message* m if t is the last transaction creating message in m's message path. One says "*m belongs to transaction t*"[8]. For example, T_{10} is the transaction of M_{12}; M_{12} belongs to transaction T_{10}. But M_{12} does not belong to transaction T_9.

- For two messages m_1 and m_2 which have a transactional least common ancestor message m, there is exactly one *least common ancestor transaction* t defined ($t = LCAT(m_1, m_2)$) where t is the transaction of m, e.g. $T_{10} = LCAT(M_{11}, M_{12})$, $T_9 = LCAT(M_{11}, M_{13})$ but $T_8 \neq LCAT(M_{11}, M_{13})$.

- Let m_1 and m_2 be two messages for which $t' = LCAT(m_1, m_2)$ is defined. Let t_1 and t_2 be the transactions of m_1 and m_2 and $t_1 \not\leq t_2$. Then, there is exactly one transaction t *one level below least common ancestor of* m_1 and m_2 defined ($t = 1LBLCAT(m_1, m_2)$) where t is the subtransaction of t' with $t \leq m_1$.

  Pictorially, t is the first transaction found when descending from t' towards m_1. Note that the definition of $1LBLCAT$ is not symmetric, i.e. $1LBLCAT(m_1, m_2) \neq 1LBLCAT(m_2, m_1)$, e.g. $1LBLCAT(M_{11}, M_{15}) = T_9 \neq T_{14} = 1LBLCAT(M_{15}, M_{11})$. $1LBLCAT(M_3, M_5)$ is not defined.

[5] "Transactional" and "non-transactional" is not to be confused with "transaction creating" and "non-transaction creating", e.g. M_{13} is transactional but not transaction creating.

[6] From left to right, i.e. from root to leaf.

[7] This is an example where transactions and messages are used interchangeably where it is clear from the context, that a transaction is meant. The "correct" description is: "t is the transaction created by the first transaction creating message of m's message path".

[8] Note that, according to this definition, a message belongs to at most one transaction, even if this is a subtransaction.

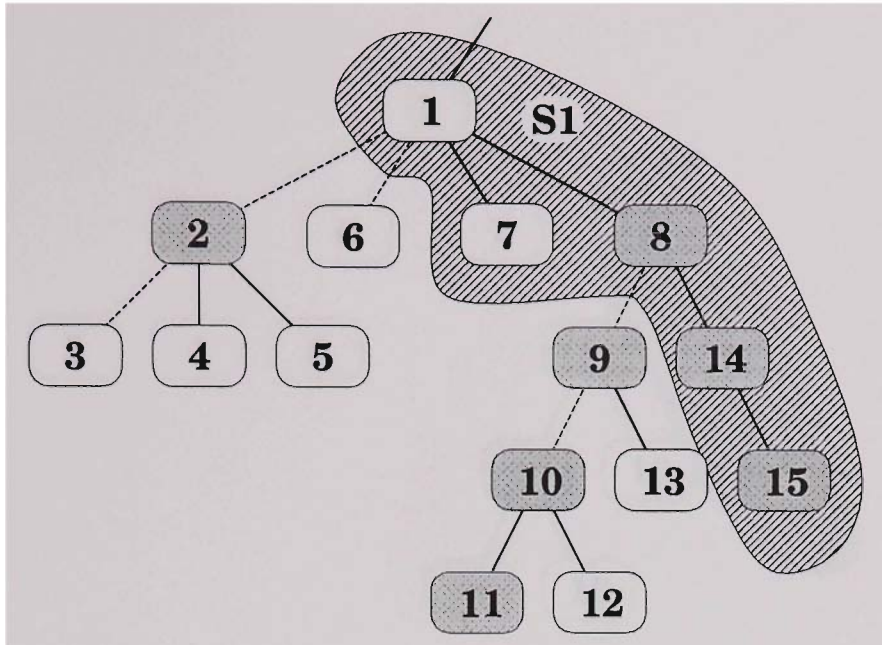

Figure 4.2: Messages belonging to thread S_1.

4.1.6 Threads

Every asynchronous message in a message tree creates a new *thread*. By default, the top-level message of a message tree also creates a thread, no matter whether it is synchronous or asynchronous. Thread identifiers are generated with an upper case letter S and the number of the message that created the thread, e.g. the thread created by M_1 is called S_1 and the thread created by M_{10} is called S_{10}. Definitions for threads are similar to definitions for transactions.

- Since each thread is associated with exactly one message in a message tree, the relationships between messages defined in Section 4.1.2 ($<, \leq, >, \geq, <>$) can be extended to threads.

- To simplify presentation, a thread creating message and its thread identifier are used interchangeably if it is clear from the context, which concept is meant. This also allows for mixed relationships between messages and threads. For example, $S_1 < M_{14}$ since $M_1 < M_{14}$. Furthermore, mixed relationships between threads and transactions are used in the same way, e.g. $S_1 < T_8$ since $M_1 < M_8$.

- A thread s is the *thread of a message* m if s is the last asynchronous message in m's message path. If there is no asynchronous message in m's message path then s is the top-level message. One says "*message* m *belongs to thread* s".

  For example, S_1 is the thread of messages M_1, M_7, M_8, M_{14} and M_{15}; conversely messages M_1, M_7, M_8, M_{14} and M_{15} belong to thread S_1 (see Figure 4.2). M_{12} belongs to S_{10} and M_3 belongs to S_3 but M_{10} does not belong to S_8.

  Since the top-level message of a message tree always creates a thread, every message in a message tree belongs to exactly one thread. A thread can be seen as the set of messages that includes the thread creating message, its synchronous children, their synchronous children and so on.

- Two messages in a message tree are *synchronous with respect to each other* if they belong to the same thread. Conversely, two messages are *asynchronous with respect*

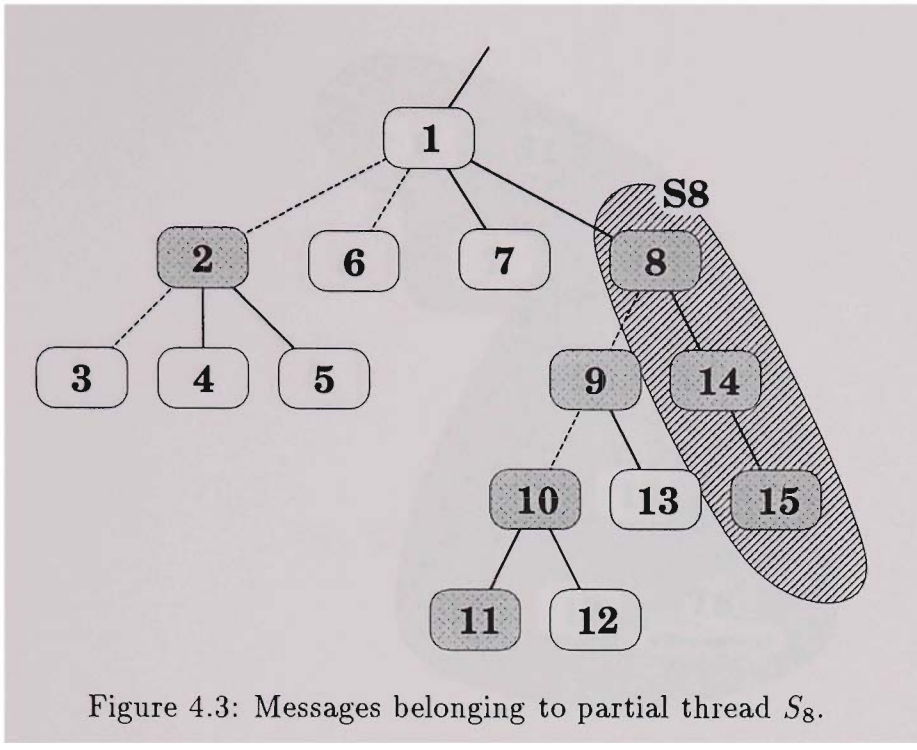

Figure 4.3: Messages belonging to partial thread S_8.

to each other if they belong to different threads[9].

4.1.7 Partial Threads Under Transactions

As pointed out in Section 3.3, transaction creation and thread creation are independent of each other. This means that there can be threads created within transactions (e.g. S_3 within T_2) and transactions within threads (e.g. T_8 within S_1). The fact that transactions and threads are specified independently of each other does not mean that there are no interactions between the two concepts. In order to deal with such interactions, the thread concept is extended to a concept of called "partial thread".

- The definition of a *partial thread* is equivalent to the definition of a thread with one exception. Every message in a message tree creates a partial thread, not only asynchronous ones.

  Every thread is also a partial thread but the opposite is not true. Like with threads, identifiers for partial threads are created with an upper case S and the number of the message that creates it. For example, partial thread S_8 is created by message M_8 and messages M_8, M_{14} and M_{15} belong to it (see Figure 4.3). All messages belonging to a partial thread always also belong to one particular thread, e.g. all messages belonging to partial thread S_8 belong to thread S_1[10]. The definition of a partial thread is used for the following important definition.

- *Thread s under transaction t (s/t)* is defined if there are messages in a message tree that belong to both thread s and transaction t or any of its descendant transactions. s/t is a partial thread with

$$s/t = \begin{cases} t^{11} & \text{if } s < t \\ s & \text{otherwise} \end{cases}$$

[9] Note that for two messages to be asynchronous with respect to each other, neither of the two messages needs to be asynchronous itself.

[10] So, a partial thread is conceptually a part of a thread—hence its name.

[11] In this context, t refers to the partial thread which is created by the message that created t.

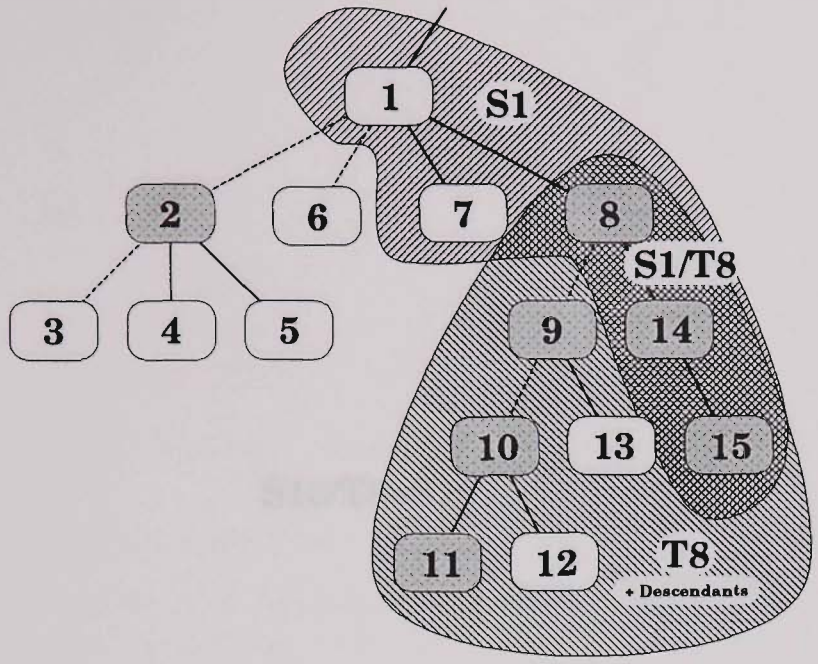

Figure 4.4: S_1/T_8: S_1 enters T_8. The two areas show messages that belong to S_1 and messages that belong to T_8 and its descendant transactions. The intersection contains the messages that belong to $S_8 = S_1/T_8$.

Messages belonging to s/t belong to both thread s and transaction t and all its descendant transactions. Pictorially, if thread s "enters" transaction t then the part of it which belongs to transaction t and its descendant transactions is used, e.g. $S_1/T_8 = S_8$ (see Figure 4.4). Otherwise, if s is created within t or any of its descendant transactions, then the whole of thread s is used, e.g. $S_{10}/T_8 = S_{10}$ (see Figure 4.5).

4.1.8 Schedules

4.1.8.1 Serial Schedules

Serial Schedules are defined for messages, (partial) threads and (nested) transactions. Central to the definitions is the concepts of the start and finish of the execution of a message.

- A *message starts execution* at the moment when the first line of its method's code starts execution. It *finishes execution* after the last line of its method's code has finished execution or a return statement is reached. Sending a message to the receiver on a possibly remote node, waiting for schedulability conditions to be satisfied and sending a result back to the sender are not included in the time span between start and finish of the execution of a message. A synchronous message can only *return a result* after it has finished execution. A synchronous, transaction creating message can only return a result after the transaction that it creates has committed or aborted.

- Two *messages m_1 and m_2 are scheduled serially (are in a serial schedule)* iff either m_1 finishes execution before m_2 starts execution or m_2 finishes execution before m_1 starts execution.

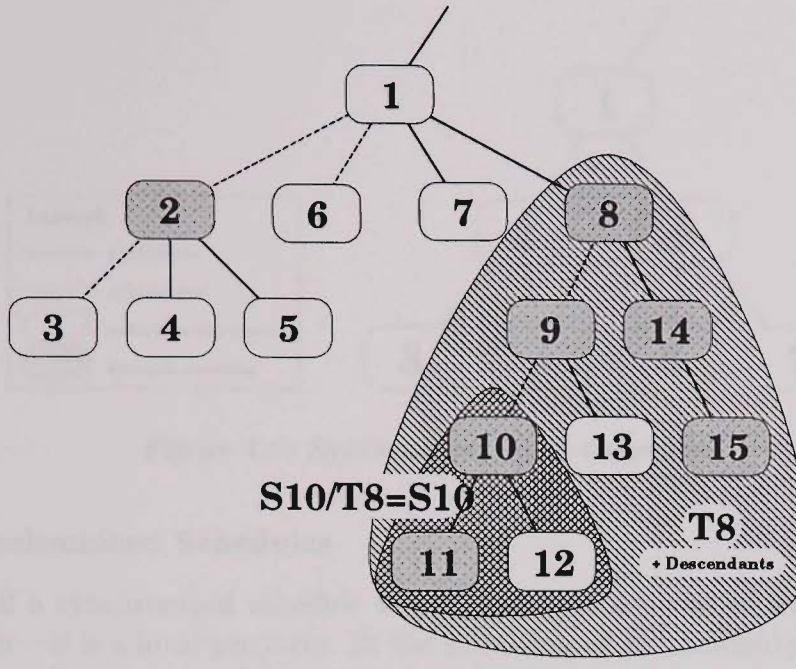

Figure 4.5: S_{10}/T_8: S_{10} is created within T_8. The two areas show messages that belong to S_{10} and messages that belong to T_8 and its descendant transactions. The intersection (S_{10}/T_8) contains all messages that belong to S_{10}.

- *Two (partial) threads s_1 and s_2 are scheduled serially (are in a serial schedule)* iff their thread creating messages are scheduled serially.

  Note that when a message that creates a (partial) thread s finishes execution then it is ensured that all messages belonging to s have finished execution. This is because messages belonging to s are all synchronous children of s, their synchronous children and so on. Therefore all these children return a result after they have finished execution and their respective parents cannot continue before the return.

  Further note that this definition does not specify the scheduling of threads that are created by any children of messages belonging to s_1 or s_2. For example, if threads S_1 and S_2 in Figure 4.1 are scheduled serially then the definition of serial execution does not pose any restrictions on the scheduling of S_9 or S_{10}.

- *Two transactions t_1 and t_2* (no matter whether they are top-level transactions or subtransactions) are *scheduled serially (are in a serial schedule)* iff iff either t_1 commits or aborts before t_2 starts execution or t_2 commits or aborts before t_1 starts execution. For conditions on the commit and abort of top-level transactions and subtransactions refer back to Sections 2.2 and 2.3.

4.1.8.2 Serializable Schedules

- Two *(partial) threads s_1 and s_2 are serialized (are in a serializable schedule)* iff their variable accesses (i.e. variables read and written by all messages belonging to s_1 and s_2) are the same as if s_1 and s_2 were in some serial schedule.

- Two *transactions t_1 and t_2 are serialized (are in a serializable schedule)* iff their variable accesses (i.e. variables read and written by all messages belonging to t_1 and t_2 and their descendant transactions) are the same as if t_1 and t_2 were in some serial schedule.

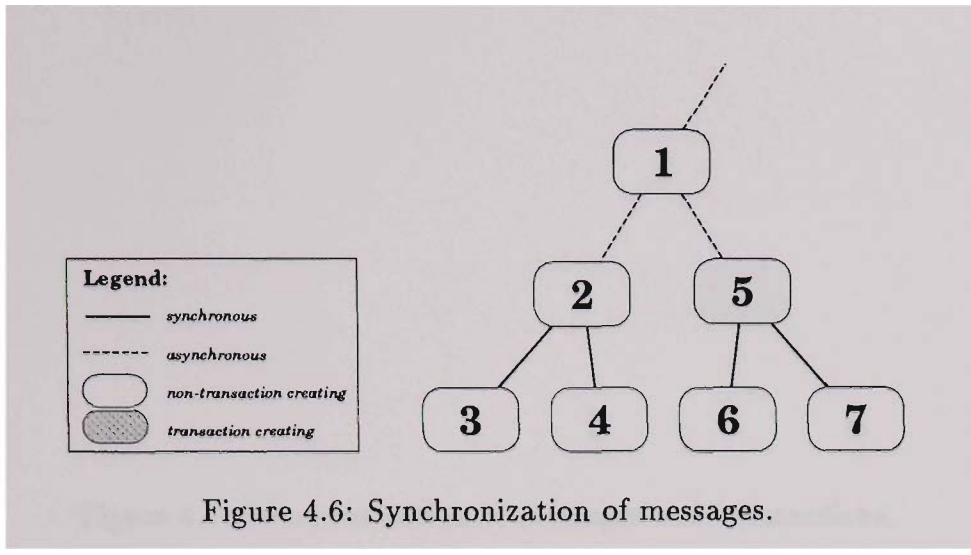

Figure 4.6: Synchronization of messages.

4.1.8.3 Synchronized Schedules

The concept of a synchronized schedule is similar to the concept of a serialized schedule but it is weaker—it is a local property. In the definition of serializability, a set of messages is involved that visit a set of objects (for threads, the set of messages that belong to the threads; for transactions the set of messages that belong to the transactions and their descendant transactions). In the definition of synchronized schedules, there are only two messages and one object involved.

- Two messages m_1 and m_2 with the same receiver object o are *synchronized (are in a synchronized schedule)* iff their variable accesses (i.e. variables of o read and written by m_1 and m_2) are the same as if m_1 and m_2 were in some serial schedule.

  Consider the example message tree of Figure 4.6. Assume that both messages M_3 and M_7 have an object O_1 as receiver and both messages M_4 and M_6 have another object O_2 as receiver. M_3 conflicts with M_7 and M_4 conflicts with M_6 and they execute in the order M_3, M_6, M_4, M_7, i.e. M_3 finishes before M_6 starts, which finishes before M_4 starts which finishes before M_7 starts. Then, both pairs of messages are synchronized: M_3 and M_7 on O_1 and M_4 and M_6 on O_2. However, the two threads S_2 and S_5 are not serialized.

- Let m be a message and t a transaction where there are messages m' that belong to t and have the same receiver object o as m. *Message m and transaction t are synchronized (are in a synchronized schedule)* iff their variable accesses to object o (i.e. variables of o read and written by messages m and m') are the same as if m and t were in some serial schedule[12].

  Consider the example of Figure 4.7. Assume that messages M_2, M_3, M_5 and M_6 all have an object O as their receiver object and they are all mutually conflicting. If the messages execute in the order M_2, M_5, M_6[13] then M_2 and T_4 are synchronized. This is not the case if the messages execute in order M_5, M_2, M_6. The serialization of M_2 and T_4 is independent of the scheduling of M_3. If the messages execute in order M_2, M_5, M_3, M_6 then the serialization condition between M_2 and T_4 is not violated.

[12] Either message m finishes execution before transaction t starts execution or t commits or aborts before m starts execution.

[13] M_2 finishes execution before M_5 starts execution which, in turn, finishes execution before M_6 starts execution.

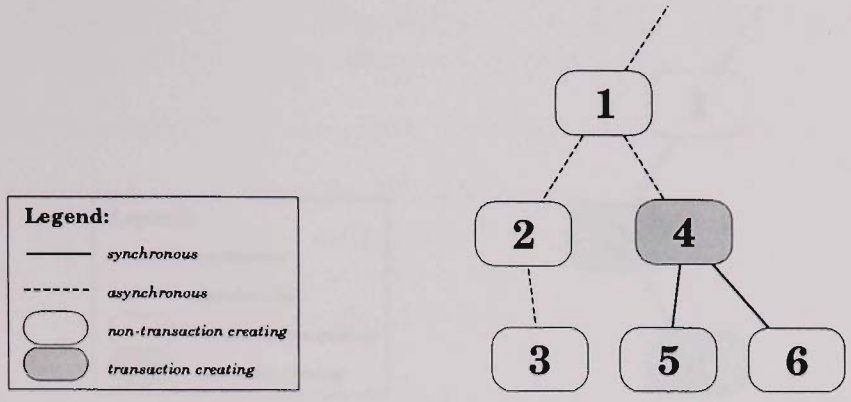

Figure 4.7: Synchronization of messages and transactions.

Serializability and synchronization properties can be implemented via 2PL. In all cases, appropriate locks (e.g. read/write locks) are acquired before data accesses and are released at some appropriate time after data accesses. For serializability, locks are acquired for all messages belonging to a thread or transaction and its descendant transactions and are released at the finish of execution or transaction commit/abort. For synchronized schedules, appropriate locks are only acquired for the receiver object. They are released at the finish of message execution or at (top-level) transaction commit/abort.

4.1.9 Cascading Aborts

If a transaction t_1 reads variables that a non-committed transaction t_2 has written, and t_2 aborts subsequently, then t_1 must be aborted as well (see Section 2.2.1.2). Such an abort is called a "*cascading abort*". There is one exception for nested transactions. If $t_2 < t_1$ and t_1 reads variables, t_2 has written and subsequently, t_2 aborts then t_1 must be aborted as well. However, this is not called a "cascading abort" since the abort is not due to the interleaved variable accesses but due to the semantics of nested transactions. If a transaction aborts then all its descendant transactions must be aborted as well (see Section 2.2.1.2).

4.1.10 Return Dependencies

Let m_1 and m_2 be two messages where m_1 cannot finish execution successfully[14] unless m_2 has finished execution. m_1 is *return dependent on message* m_2 *(m_1 and m_2 are in a return dependency)* if this dependency is caused by the semantics of the return of messages[15] and the semantics of (nested) transactions[16,17].

The simplest example of a return dependency is a synchronous message. For example, if a message m_1 sends a message m_2 synchronously, then m_1 cannot finish execution before m_2 has finished execution. This is due to the semantics of the return of messages and is independent of whether m_1 and m_2 are conflicting or not. Hence, m_1 and m_2 are in a return dependency.

A more subtle example is shown in Figure 4.8. In this case, M_1 is return dependent

[14] Successfully means here "without the involvement of aborts".

[15] A message sending a synchronous message waits until the child has finished execution and has returned a value. A message sending an asynchronous message does not wait. See Section 3.3.

[16] A transaction cannot commit before it has finished execution and all descendant transactions have committed or aborted. A synchronous transaction does not return a result before it has committed or aborted. See Sections 2.2 and 2.3.

[17] This means in particular that the dependency is not caused by conflicting messages that lead to deadlock.

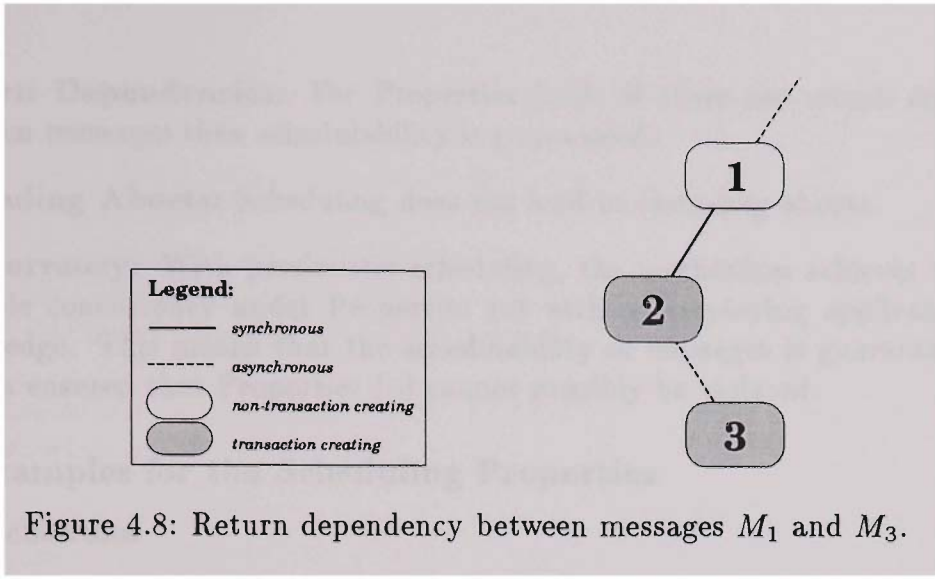

Figure 4.8: Return dependency between messages M_1 and M_3.

on M_2 although M_1 and M_2 are asynchronous with respect to each other. M_1 cannot finish before M_2 has returned, due to the semantics of the return of synchronous messages. M_2 cannot return before T_2 has committed[18] due to the semantics of (top-level) transactions. T_2 cannot commit before all subtransactions (here T_3) have committed, due to the semantics of nested transactions. T_3 cannot commit unless M_3 has executed, due to the semantics of transactions. This is why M_1 and M_3 are in a return dependency.

4.2 The Scheduling Properties

The scheduling mechanism for the generalized message scheme, presented in this thesis, has the following four properties.

1. **Schedules**

   (a) **Serializability of Top-Level Transactions:** For all pairs t_1 and t_2 of top-level transactions in the execution of a system: if t_1 and t_2 are not in a return dependency then t_1 and t_2 are serialized.

   (b) **Serializability of Transactional Partial Threads:** Let m_1 and m_2 be two messages that are asynchronous with respect to each other and that belong to the same top-level transaction. Let s_1 and s_2 be the threads of m_1 and m_2, respectively. Then, for all such m_1, m_2, s_1 and s_2 in the execution of a system: if $s_1/LCAT(m_1, m_2)$ and $s_2/LCAT(m_1, m_2)$ are not in a return dependency then they are serialized.

   (c) **Synchronization of Non-Transactional Messages and Top-Level Transactions:** For all non-transactional messages m and top-level transactions t in the execution of a system: If there is a message belonging to top-level transaction t that has the same receiver object as m and there is not return dependency between m and t or vice versa, then m and t are synchronized.

   (d) **Synchronization of Non-Transactional Messages:** For all two non-transactional messages m_1 and m_2 in the execution of a system: if m_1 and m_2 have the same receiver object and there is no return dependency between m_1 and m_2 then they are synchronized.

[18] The abort case is not considered here since the definition of return dependencies deals with the successful finish of methods only. Of course, T_2 can also abort. But the concept of return dependencies has been introduced to consider the important question whether a transaction can *possibly* commit.

2. **Return Dependencies:** For Properties 1a-d: if there are return dependencies between messages then schedulability is guaranteed.

3. **Cascading Aborts:** Scheduling does not lead to cascading aborts.

4. **Concurrency:** With pessimistic scheduling, the mechanism achieves the highest possible concurrency under Properties 1-3 without employing application-specific knowledge. This means that the schedulability of messages is guaranteed as soon as it is ensured that Properties 1-3 cannot possibly be violated.

4.2.1 Examples for the Scheduling Properties

4.2.1.1 Schedules

Serializability of Top-Level Transactions: For all examples presented in this section, refer to Figure 4.1. Consider the example that messages M_{12} and M_4 are conflicting and M_{12} starts execution before M_4 is sent. Note that this scenario is possible since M_{12} and M_4 are asynchronous with respect to each other.

The scheduling mechanism determines the schedulability of M_4 according to the scheduling properties. M_{12} and M_4 belong to different top-level transactions (T_8 and T_2). Since T_8 and T_2 are not in a return dependency[19], Property 1a requires top-level transactions T_8 and T_2 to be serialized. Therefore, M_4 cannot be scheduled unless either of the following two events has happened. The transaction of M_{12} (T_{10}) has aborted[20] or the top-level transaction of M_{12} (T_8) has committed.

In the abort case, serializability is not defied since if T_{10} aborts, all of its effects are undone. The schedule in the commit case is equivalent to the serial schedule "T_8 before T_2". This is because M_4 is not scheduled before top-level transaction T_8 has committed.

If M_4 was scheduled before either of the two events (e.g. immediately after T_{10} has committed) then other conflicting messages that belong to top-level transaction T_8 could execute subsequently. In this case, either serializability is defied or cascading aborts are necessary.

Serializability of Transactional Partial Threads: Consider the example that messages M_{14} and M_{12} are conflicting and M_{14} starts execution before M_{12} is sent. The two messages belong to different threads (S_1 and S_{10}, respectively), are not in a return dependency and belong to the same top-level transaction, T_8. Property 1b requires that $S_8 = S_1/T_8$ and $S_{10} = S_{10}/T_8$ be serialized. Therefore, M_{12} cannot be scheduled unless S_8 has finished execution. This is the serial schedule "S_8 before S_{10}".

This example demonstrates why Property 1b requires $s_1/LCAT(m_1, m_2)$ and $s_2/LCAT(m_1, m_2)$ to be serialized rather than s_1 and s_2 to be serialized. In the case where one of the two threads, say, s_1 enters $LCAT(m_1, m_2)$ (i.e. the case where $s_1/LCAT(m_1, m_2) \neq s_1$) there is a return dependency between s_1 and s_2. Therefore, there cannot be a serial schedule between s_1 and s_2 and thus, s_1 and s_2 cannot possibly be serialized. In this example, $s_1 = S_1$, $s_2 = S_{10}$ and $LCAT(m_1, m_2) = T_8$. S_1 starts execution before S_{10} because $S_1 < S_{10}$. However, S_1 cannot finish execution before S_{10} has executed. This is because M_1 waits for T_8 to commit before it returns a value. T_8 cannot commit before all of its descendant threads have finished execution, including S_{10}. Thus, there is a return dependency between S_1 and S_{10}. However, there cannot be a return dependency between S_1/T_8 and S_{10}/T_8.

[19] Section 4.4.1 examines in general when two messages are in a return dependency.

[20] Note that the abort of any ancestor transaction of T_{10} (T_8, T_9) causes T_{10} to be aborted, too.

Synchronization of Non-Transactional Messages and Top-Level Transactions: Consider the example that messages M_6 and M_{12} are conflicting and M_6 starts execution before M_{12} is sent. M_{12} is transactional and M_6 is non-transactional. Property 1c requires M_6 and the top-level transaction of M_{12}, T_8, to be synchronized. This is ensured if M_{12} is not scheduled unless M_6 has finished execution.

Now consider the reverse case. M_{12} starts execution before M_6 is sent[21]. To ensure Property 1c, M_6 is not scheduled unless either of the following two events has happened.

1. The transaction of M_{12}, T_{10}, aborts. Then, all effects of T_{10} are undone, as if T_{10} had not happened at all.

2. The top-level transaction of M_{12}, T_8, commits. This schedule is equivalent to the serial schedule "T_8 before M_{12}".

Synchronization of Non-Transactional Messages: Consider the example that messages M_6 and M_7 are conflicting. M_6 has started execution before M_7. Property 1d requires M_6 and M_7 to be synchronized. This is ensured if M_7 is not scheduled unless M_6 has finished execution. This is the serial schedule "M_6 before M_7".

4.2.1.2 Return Dependencies

Consider the example that messages M_1 and M_{13} are conflicting. Since M_1 is an ancestor of M_{13}, it starts execution before M_{13} is sent. This example has similarities with the example for Property 1c. M_1 is non-transactional and M_{13} is transactional. However, there is an important difference. M_1 is return dependent on the top-level transaction of M_{13}, T_8. This is because T_8 is a synchronous child of M_1, the simplest form of return dependency. For this reason, there cannot possibly be a serial schedule between M_1 and T_8. M_1 cannot execute before T_8 because T_1 waits for T_8 to return a result; T_8 cannot execute before M_1 since it is a descendant. This is why Property 1c does not require M_1 and T_8 to be serialized in this case.

Instead, Property 2 allows M_{13} to progress so that T_8 has a chance of finishing successfully. Note that, although M_1 and T_8 are technically not synchronized, there is no danger that the two methods concurrently access variables in a conflicting way. This is because M_1 is suspended until T_8 commits or aborts.

4.2.1.3 Cascading Aborts

Consider the example that messages M_{11} and M_{15} are conflicting and M_{11} starts execution before M_{15} is sent. M_{11} and M_{15} belong to different transactions (T_{11} and T_{15}, respectively) that are descended from the same top-level transaction, T_8. Property 3 requires that cascading aborts be prevented. This is ensured if T_{15} is not scheduled unless either of the two events has happened: T_{11} aborts or T_9 commits.

If T_{11} aborts then all its effects are undone. Therefore, T_{15} cannot see uncommitted state and aborts cannot cascade. However, if T_{11} commits then cascading aborts are not necessarily avoided. Consider the case that T_{11} commits, M_{15} is scheduled subsequently and reads state that has been written by T_{11}. Then, T_{10} aborts, in turn causing its descendant T_{11} to be aborted. Then, T_{15} has to be aborted as well since it has seen state written by T_{11}—a cascading abort.

The same argument holds when M_{15} is scheduled after T_{10} has committed. However, the argument does not hold if M_{15} is scheduled after T_9 has committed. This is because

[21] Since M_6 and M_{12} are asynchronous with respect to each other, both cases can occur.

the only way, T_9 can be aborted after it has committed is when any of its ancestor transactions aborts. But all of its ancestor transactions (here: only T_8) are also ancestor transactions of T_{15}. Therefore, an abort of T_8 would cause an abort of T_{15} anyway—aborts do not cascade.

Note that $T_8 = 1LBLCAT(M_{11}, M_{15})$. Requiring the transaction one level below the least common ancestor to have committed prevents cascading aborts.

4.2.1.4 Concurrency

In all examples for Properties 1-3, messages are not scheduled unless the possibility of one of the Properties 1-3 to be violated can be ruled out completely without employing application-specific knowledge. This policy decreases concurrency but ensures the scheduling properties. However, schedulability is guaranteed as soon as such a violation can be ruled out. This means that concurrency is only restricted as much as *necessary* to ensure scheduling properties but no further. Hence, using pessimistic scheduling, highest *possible* concurrency is achieved under the restrictions, Properties 1-3 pose.

4.2.2 Discussion of the Scheduling Properties

4.2.2.1 Schedules

Serializability of Top-Level Transactions: This property is equivalent to the serializability condition in the traditional nested transaction model [Mos81].

Serializability of Transactional Partial Threads: This property reflects the extension to Moss' model. Recall that in Moss' model, threads can only be created via asynchronous subtransactions. Asynchronous subtransactions of the same parent transaction are serialized with respect to each other. The generalized message scheme allows transactional threads that do create a subtransaction and others that don't. Property 1b ensures that all threads with the same parent are serialized with respect to each other, no matter whether they create a subtransaction or not. In addition, it ensures that all other threads belonging to a top-level transaction are serialized with respect to each other. This also ensures, for example, that ancestor and descendant transactions be serialized.

Property 1b ensures serializability for all transactional threads that can be scheduled serially. For threads that cannot possibly be serialized in full, only their partial threads that actually can be serialized are considered. This is why Property 1b only considers threads under common transactions. No thread entering a transaction can be serialized with a thread that is created within the transaction or any of its descendant transactions (refer back to the example for Property 1b). However, the threads under the common transaction can be serialized. Like the example for Property 1b suggests, there is no danger of interleaving conflicting accesses between the "outside parts" of threads, entering a transaction and threads created within the transaction. This is, because these "outside parts" are suspended until the transaction commits or aborts.

The semantics provided are intuitive and easy to understand by application programmers. The system ensures that there is no interleaving of any kind of transactional threads, no matter whether they create subtransactions or not. Furthermore, it ensures that every thread has the chance of finishing successfully.

Transactional threads that do not create a subtransaction allow higher concurrency than transactional threads that do create a subtransaction. This is although both types of threads are serialized. Reconsider the example for Property 3. M_{11} and M_{15} are conflicting and M_{11} starts execution before M_{15} is sent. In order to avoid cascading

aborts, M_{15} cannot be scheduled unless T_9 has committed[22]. Serializability between the threads $S_{10} = S_{10}/T_8$ and $S_8 = S_1/T_8$ is also ensured since S_{10} has finished execution by the time T_9 commits. Now imagine that T_8 was not nested, i.e. M_9, M_{10}, M_{11}, M_{14} and M_{15} were all non-transaction creating messages. In this case, M_{15} can be scheduled after S_{10} has finished execution. This still ensures serializability. Cascading aborts cannot be caused by the scheduling of M_{15} since both messages M_{11} and M_{15} belong to the same transaction T_8. Higher concurrency is achieved in the non-nested case since M_{15} can be scheduled at an earlier time.

In the generalized message scheme, the application programmer has the choice between two kinds of transactional threads that both support serializability—subtransaction creating ones and non-subtransaction creating ones. The difference between the two kinds lies in their expense, level of concurrency and level of recovery. In the non-nested transaction case, a failure of any message causes the top-level transaction T_8 to abort. In the nested transaction case, the parent transaction of the aborting transaction has alternatives to aborting. It can retry the subtransaction, try another message or simply ignore the abort of its child. On the other hand, nested transactions provide less concurrency (as shown above) and are more expensive due to recovery related work. With this choice between two kinds of transactional threads that support serializability, the application programmer can explicitly trade-off the expense and level of concurrency with the level of recovery.

Other systems (e.g. Encina or Venari/ML) provide even less expensive kinds of transactional threads than serialized, non-transaction creating threads. These threads do not provide serializability semantics at all. Such threads can be useful if the application programmer knows that due to the semantics of the application, particular threads cannot interleave. Take the example of a bank transfer. No serializability semantics are required to protect the deposit and withdraw operations from interleaving. This is because because the deposit and withdraw operations are performed on *different* accounts and therefore cannot possibly interleave. The advantage of such inexpensive transactional threads is that no performance penalty has to be paid for the unnecessary serialization of threads. To keep the scheduling rules simpler, such threads are not included in the definition of the generalized message scheme. An extension to the scheduling mechanism that allows non-serialized transactional threads is presented in Section 4.8.

Synchronization of Non-Transactional Messages and Top-Level Transactions: Non-transactional messages are unreliable but efficient and can be used for aspects of an application where reliability and data integrity is not important. By using transactional or non-transactional messages, the application programmer can explicitly trade-off reliability versus performance. The idea is to make non-transactional messages as cheap as *possible* but also as expensive as *necessary* in order to maintain the integrity of transactions. If non-transactional messages did not acquire any locks at all they could interfere with transactions in an uncontrolled way, violating the semantics of transactions.

Property 1c ensures that this cannot happen. Non-transactional messages acquire appropriate locks (e.g. read locks if they only read the receiver's variables or write locks if they write to the receiver's variables) before the start of execution and release them straight after the end of execution. Note that a non-transactional message can execute for a long time (e.g. if it sends synchronous, transaction creating messages) or for a very short time (e.g. if it only accesses a single variable). The application programmer can explicitly determine the length of time, locks are held by non-transactional messages by setting appropriate locks at appropriate levels in a message tree.

[22] Only the successful execution of messages is considered here, not the abort case.

4.2.2.2 Return Dependencies

If two messages are in a return dependency, then the threads and transactions they create cannot be serialized. This is because two messages that are in a return dependency cannot be scheduled serially. One approach is to treat all return dependencies as programming errors and cause a deadlock in this case. This approach is considered too restrictive and is therefore not taken in this scheduling mechanism. If two messages are in a return dependency then it is guaranteed that their respective threads and transactions have a chance of finishing successfully. Note that this approach is not problematic in terms of interleaving of conflicting messages. This is because, as the example for Property 3 suggests, messages are always suspended until all messages, they are return dependent on, have finished execution.

4.2.2.3 Cascading Aborts

It has been shown in performance studies that pessimistic concurrency control without cascading aborts exhibits better performance than optimistic concurrency control over a wide range of parameters (see Section 2.2.1.2). This is why this approach has been chosen. It is in line with the concurrency control strategy used in [Mos81].

4.2.2.4 Concurrency

High concurrency is generally desirable in a distributed system since it allows a proper use of the system's resources. As described in Chapter 2, there are certain problems with concurrency if it is uncontrolled. This is why concurrency is restricted by a distributed system so that useful properties, like, e.g. serializability, can be ensured. Properties 1-3 describe such useful properties. They state that concurrency is restricted at least as much as is necessary to ensure them, but possibly more. Property 4 says that concurrency is not restricted unnecessarily—only as much as is necessary to ensure Properties 1-3 without employing additional, application-specific knowledge.

4.3 The Schedulability Predicate

This section defines a schedulability predicate that satisfies the scheduling properties presented in the last section. This is done in terms of two predicates "is schedulable" and "is schedulable with respect to". First, an auxiliary definition "*retDep*" is made.

- For two messages m_1 and m_2: m_1 *retDep* m_2 iff the message paths for m_1 and m_2 are in the following relationship. $m_2 = m_1$ *synch-nonTrans* [synch-trans any*]*. *retDep* is simply used as an alias for the relationship described by the regular expression. In Section 4.4 it is shown that *retDep* is equivalent to the return dependency relationship.

- A message m_2 is *schedulable* iff for all conflicting messages m_1 that have started execution[23] in a system, m_2 is schedulable with respect to m_1[24].

- Let m_1 and m_2 be two messages where m_1 has started execution and m_2 has been sent but has not yet started execution. Let s_1 be the thread of m_1 and, if m_1 is transactional, t_1 the transaction and tl_1 the top-level transaction of m_1. Let s_2 be the thread of m_2 and, if m_2 is transactional, t_2 the transaction and tl_2 the top-level

[23] m_1 might even have finished execution.

[24] The indexes of m_1 and m_2 indicate the order in which the two messages start execution.

transaction of m_2. Message m_2 *is schedulable with respect to message* m_1 iff $s_1 = s_2$ or m_1 *retDep* m_2 or the following three predicates are satisfied.

1. if m_1 is non-transactional then the execution of m_1 must have finished;
2. if m_1 is transactional and m_2 is non-transactional then t_1 must have aborted or tl_1 must have committed;
3. Otherwise (i.e. if both m_1 and m_2 are transactional) then the following two predicates must be satisfied.
   (a) if $tl_1 \neq tl_2$ then t_1 must have aborted or tl_1 must have committed;
   (b) Otherwise (i.e. if $tl_1 = tl_2$) then the following four predicates must be satisfied.
      i. if $t_1 = t_2$ then the execution of s_1/t_1 must have finished;
      ii. if $t_1 < t_2$ then the execution of s_1/t_1 must have finished or s_1/t_1 *retDep* s_2/t_1;
      iii. if $t_1 > t_2$ then $1LBLCAT(m_1, m_2)$ must have committed and the execution of s_1/t_2 must have finished;
      iv. Otherwise (i.e. if $t_1 <> t_2$) then $1LBLCAT(m_1, m_2)$ must have committed and either the execution of $s_1/LCAT(m_1, m_2)$ has finished or $s_1/LCAT(m_1, m_2)$ retDep $s_2/LCAT(m_1, m_2)$.

4.4 Correctness of the Schedulability Predicate

If all messages of different threads and transactions have different receiver objects then they can be scheduled immediately and there is no danger of violating any scheduling predicates of Section 4.2. The same is true if the lock types of messages with the same receiver object never conflict, e.g. if only read accesses are performed to shared data. This is why for the schedulability of a message m_2, only conflicting messages m_1 need to be considered, that have started execution[25].

Before the correctness of the schedulability predicate with respect to the scheduling properties is analyzed, two lemmas are shown. Section 4.4.1 shows that *retDep* is equivalent to the return dependency relationship. Section 4.4.2 shows that cascading aborts are avoided if the transaction one level below the least common ancestor has committed.

4.4.1 Return Dependencies

4.4.1.1 Dependency Rules

The following five rules describe dependency relationships between the execution and return of messages.

1. A message sending a synchronous message waits until the synchronous submessage returns a result. Therefore, the finish of execution of a message depends on the return of synchronous submessages.

2. A message sending an asynchronous message is not suspended. Therefore, the finish of execution of a message does not depend on the finish of execution of asynchronous submessages.

[25] Since the schedulability predicate does not make use of application-specific knowledge, messages that are going to be sent in future cannot be considered.

3. A synchronous non-transaction creating message returns immediately after it has finished execution. Therefore, the return of a synchronous transaction creating message depends only on the finish of execution.

4. A synchronous transaction creating message returns after the transaction it creates has committed or aborted. Therefore, the return of a synchronous transaction creating message depends on the commit or abort of the transaction it creates.

5. Transaction commit entails the finish of execution of the message itself, finish of execution of all threads that belong to it and the commit or abort of all descendant transactions. Therefore, the commit of a transaction depends on the finish of execution of all descendant messages.

These five rules describe all relevant dependency relationships in the generalized message scheme that are due to the semantics of the return of messages and nested transactions. However, this is only the case for the restriction of the generalized message scheme to two message kinds: synchronous and asynchronous. With wait-by-necessity messages, there are more complex dependency relationships (see Section 4.7).

It is worth noting that in these rules, dependencies occur only between ancestor and descendant messages and not between descendant and ancestor messages or between incomparable messages. As pointed out in Section 4.7, this is not necessarily the case for wait-by-necessity messages.

Before the equivalence of *retDep* and the return dependency relationship is shown, a lemma is shown.

4.4.1.2 The Partition Lemma

Let mp_1 and mp_2 be two templates for message paths with mp_1 = *synch-nonTrans* [synch-trans any*]* and mp_2 = *synch-nonTrans* asynch any**. Then, mp_1 and mp_2 partition the set of all message paths, i.e. every message path either matches mp_1 or mp_2 but none matches both.

To show this, the set of all message paths is partitioned into two subsets A and B. A contains all message paths that do not have an asynchronous message path element. B contains all message paths that have at least one asynchronous message path element.

Obviously, none of the elements of A are matched by mp_2. Furthermore, all elements of A are matched by mp_1. This is because there is only two kinds of synchronous message path elements. Ones that create a transaction and ones that don't. mp_1 matches message paths that contain *synch-nonTrans* message path elements only (optional elements do not occur), message paths that contain *synch-trans* message path elements only (if the first "star" denotes zero occurrences and optional elements occur) and arbitrary mixing of the two (via "*any*").

Now consider set B. Set B is is further partitioned into two subsets C and D. C contains message paths that have have only *synch-nonTrans* message path elements (possibly zero) before their first *asynch* message path element. D contains messages that have at least one synch-trans message path elements before their first asynch message path element.

Obviously, all elements of C are matched by mp_2. Furthermore, no element of C can be matched by mp_1 since it requires at least one *synch-trans* message path element before an *asynch* message path element.

Obviously, no element of D is matched by mp_2. Furthermore, all elements of D are matched by mp_1. This is because the only synchronous message path element that does

not match *synch-nonTrans* is *synchTrans*. mp_1 allows a *synch-trans* message path element before the first asynch message path element.

4.4.1.3 Equivalence of retDep and Return Dependency

In the following, it is shown that two messages m_1 and m_2 are in a return dependency iff m_1 *retDep* m_2, i.e. $m_1 \leq m_2$ and $m_2 = m_1$ *synch-nonTrans* [synch-trans any*]*. This is shown in two parts.

1. if m_1 *retDep* m_2 then m_1 is return dependent on m_2;
2. if not m_1 *retDep* m_2 then m_1 is not return dependent on m_2.

m_1 *retDep* m_2: To show: if $m_1 \leq m_2$ and $m_2 = m_1$ *synch-nonTrans* [synch-trans any*]* then m_1 is return dependent on m_2. Two cases are distinguished:

1. $m_2 = m_1$ *synch-nonTrans** (i.e. the optional part does not occur);
2. $m_2 = m_1$ *synch-nonTrans* synchTrans any** (i.e. the optional part occurs).

$m_2 = m_1$ *synch-nonTrans**: This is a trivial case since m_1 is a descendant of m_2 and m_1 is synchronous with respect to m_2. Therefore, m_1 cannot finish execution before m_2 has returned a result (Rule 1).

$m_2 = m_1$ *synch-nonTrans* synchTrans any**: m_1 cannot finish before the first synchronous transaction has returned a result (Rule 1). The synchronous transaction cannot return a result before it has committed (Rule 4)[26]. The transaction cannot commit before all messages that belong to it and any of its descendant transactions have executed, including m_2 (Rule 5). Thus, m_1 is return dependent on m_2.

Now consider the second case.

not m_1 *retDep* m_2: If neither $m_1 \leq m_2$ nor $m_2 = m_1$ *synch-nonTrans* [synch-trans any*]* then m_1 is not in return dependency with m_2. Two subcases are distinguished.

1. $m_1 \not\leq m_2$;
2. $m_1 \leq m_2$ but $m_2 \neq m_1$ *synch-nonTrans* [synch-trans any*]*.

$m_1 \not\leq m_2$: As noted above, two messages can only be in a return dependency if they are in an ancestor/descendant relationship.

$m_1 \leq m_2$ **but** $m_2 \neq m_1$ *synch-nonTrans* [synch-trans any*]*: As follows directly from the partition lemma, this condition is equivalent to the condition $m_2 = m_1$ *synch-nonTrans* asynch any**.

If $m_2 = m_1$ *synch-nonTrans* asynch any** then there is no return dependency between m_1 and m_2. *synch-nonTrans* messages return immediately after they finish execution (Rule 3). The first asynch message returns immediately (Rule 2). Therefore, m_1 can finish execution independently of any descendant of the first *asynch* message, in particular m_2. Thus, there is no return dependency between m_1 and m_2 in this case.

In the following two sections, useful lemmas about return dependencies are shown.

[26]The abort case is not considered here since the return dependency definition deals with successful schedules of messages only.

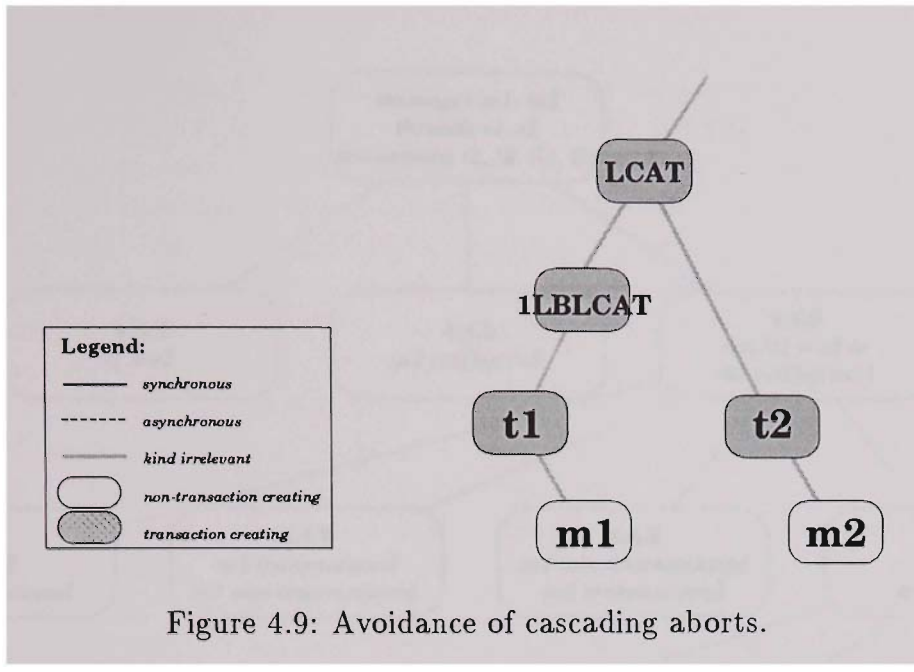

Figure 4.9: Avoidance of cascading aborts.

4.4.1.4 Transitivity of retDep

retDep is transitive, i.e. if for three messages m_1, m_2 and m_3: m_1 *retDep* m_2 and m_2 *retDep* m_3 then m_1 *retDep* m_3.

This is obvious from the definition of a return dependency and the fact that *retDep* is equivalent to the return dependency predicate.

4.4.1.5 Return Dependencies of Intermediate Messages

For three messages m_1, m_2 and m_3: if m_1 *retDep* m_3 and $m_1 \leq m_2 \leq m_3$ then m_1 *retDep* m_2.

Since m_1 *retDep* m_3, $m_3 = m_1$ *synch-nonTrans* [synch-trans any*]*. Since $m_1 \leq m_2 \leq m_3$, m_2 is either of the form m_1 or m_1 *synch-nonTrans** or m_1 *synch-nonTrans* synch-trans* or m_1 *synch-nonTrans* synch-trans any**. In any case, m_2 matches the criterion for m_1 *retDep* m_2.

4.4.2 Cascading Aborts

Consider the case that messages m_1 and m_2 are conflicting, e.g. m_2 reads one of its receiver's variables that m_1 has written. An abort of t_1 may then cause a cascading abort of t_2. In this section it is shown that a cascading abort cannot occur if such a message m_2 is never scheduled unless the following events have happened.

1. in case $tl_1 \neq tl_2$: tl_1 has committed;
2. in case $tl_1 = tl_2$: $1LBLCAT(m_1, m_2)$ has committed.

4.4.2.1 $tl_1 \neq tl_2$

This case is trivial. A top-level transaction or any of its descendant transactions cannot abort after it has committed.

4.4.2.2 $tl_1 = tl_2$

See Figure 4.9. All figures of message trees in this and following sections demonstrate relationships between m_1, m_2, t_1, t_2, tl_1 and tl_2. Boxes are labeled with these placeholders for identifiers. To increase the generality of the figures, the meaning of lines is extended.

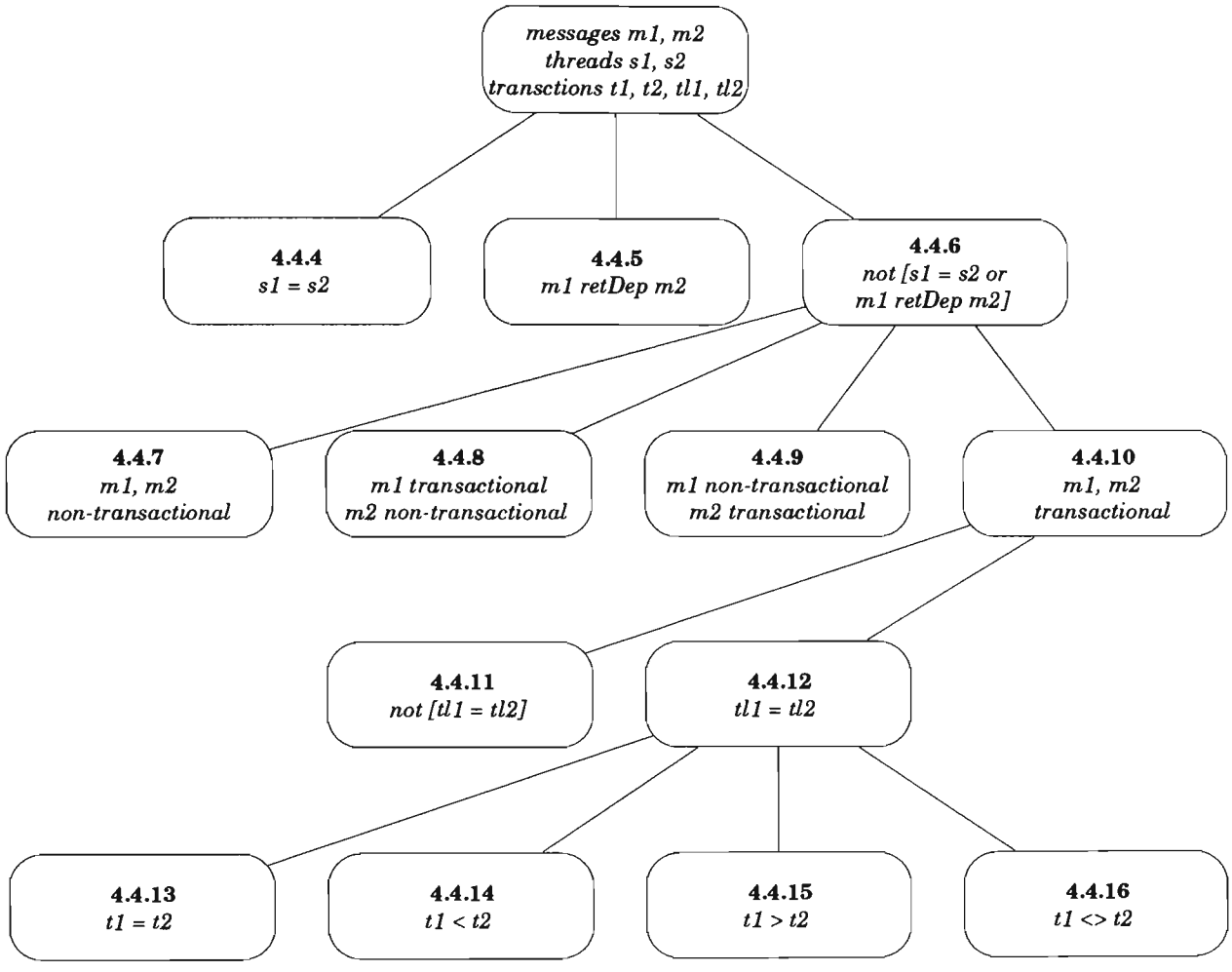

Figure 4.10: Partition of cases.

Solid lines denote that two messages are synchronous with respect to each other, i.e. they are linked via an arbitrary number of synchronous messages, not necessarily only one. Analogously, a broken line denotes that two messages are asynchronous with respect to each other, i.e. that they are linked via at least one asynchronous message, not only necessarily one. Grey lines are used if the kind of messages is irrelevant for a particular case.

The only way, t_1 can be aborted after $1LBLCAT$ has committed is via the abort of $LCAT(m_1, m_2)$ or any of its ancestor transactions. However, $LCAT(m_1, m_2)$ and its ancestor transactions are also ancestors of t_2. Therefore, their abort causes the abort of t_2 due to the semantics of nested transactions. This is not a cascading abort.

4.4.3 The Partition of Cases

In the following sections, the correctness of the schedulability predicate is shown with respect to the scheduling properties. To cope with the complexity of the correctness analysis, the set of all pairs of messages, m_1 and m_2, that can be compared for schedulability is broken down into a large number of subsets. This division into subcases is performed such that it is easy to see that all cases are covered. Also, the scheduling properties can be shown relatively easily for each individual subcase. Figure 4.10 shows the partition of cases examined. Section numbers indicate the sections in which particular cases are analyzed. It is suggested that the section numbers are used as a guidance through the large number of cases.

First examine the separation into the three main cases $s_1 = s_2$, m_1 *retDep* m_2 and $\neg[s_1 = s_2 \vee m_1$ *retDep* $m_2]$. This separation is not a partition, i.e. there are pairs of messages that are covered by both the first and the second case. However, the fact that the third case is the negation of the disjunction of the first two cases makes it obvious that all cases are covered. The third case is further separated into four subcases. Messages m_1 and m_2 can be either transactional or non-transactional. This makes four combinations which are considered individually. The case that both messages are transactional is further separated into two subcases: $tl_1 \neq tl_2$ and $tl_1 = tl_2$. Again, it is obvious that this covers all cases. $tl_1 = tl_2$ is separated into four subcases $t_1 = t_2$, $t_1 < t_2$, $t_1 > t_2$, $t_1 <> t_2$. The definitions of these relationships make it obvious that all cases are covered. Recall that $m_1 <> m_2$ is defined as neither $m_1 \leq m_2$ nor $m_2 \leq m_1$.

Figure 4.10 does not actually show the separation into all subcases. Some cases are even split up further. Whenever a separation into subcases is performed, it is easy to see that all possible cases are covered. For each individual case, it is shown that the schedulability predicate satisfies the five scheduling properties and in particular the following.

Schedules:

Serializability of Top-Level Transactions: if m_1 and m_2 are both transactional and $tl_1 \neq tl_2$ and not tl_1 *retDep* tl_2 then tl_1 and tl_2 are serialized.

Serializability of Partial Transactional Threads: if m_1 and m_2 are both transactional and $tl_1 = tl_2$ and not m_1 *retDep* m_2 then $s_1/LCAT(m_1, m_2)$ and $s_2/LCAT(m_1, m_2)$ are serialized.

Synchronization of Non-Transactional Messages and Top-Level Transactions: if m_1 is non-transactional and m_2 is transactional and not m_1 *retDep* tl_2 then m_1 and tl_2 are synchronized. If m_1 is transactional and m_2 is non-transactional and not tl_1 *retDep* m_2 then tl_1 and m_2 are synchronized.

Synchronization of Non-Transactional Messages: if both m_1 and m_2 are non-transactional and not m_1 *retDep* m_2 then m_1 and m_2 are synchronized.

Return Dependencies: If there is a return dependency in the previous four subcases then m_2 is schedulable.

Cascading Aborts: An abort of t_1 can not cause a cascading abort of t_2.

Concurrency: A weaker schedulability predicate potentially violates any of the Properties 1-3.

Initially, consider three cases.

1. $s_1 = s_2$
2. m_1 *retDep* m_2
3. neither $s_1 = s_2$ nor m_1 *retDep* m_2.

Note that although some pairs of messages m_1 and m_2 may fall into Cases 1 and 2, the fact that Case 3 is the negation of Cases 1 and 2 ensures that all message pairs are covered.

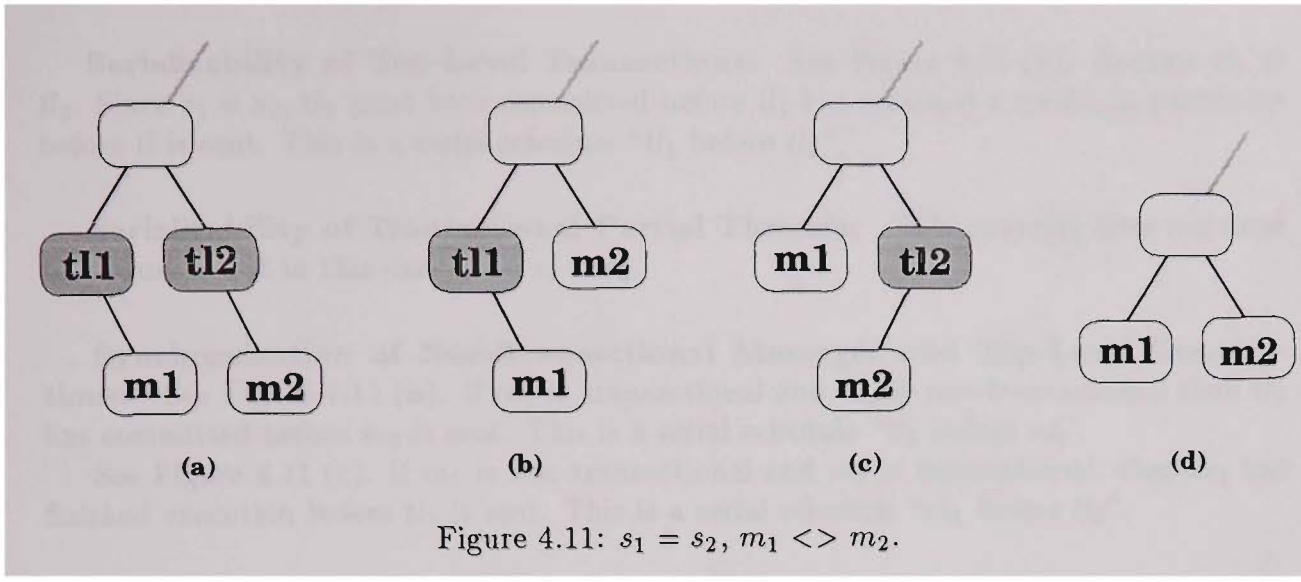

Figure 4.11: $s_1 = s_2$, $m_1 <> m_2$.

4.4.4 $s_1 = s_2$

In this case, the schedulability predicate for m_2 is satisfied unconditionally.

4.4.4.1 Schedules

Consider three subcases.

$$m_1 \left\{ \begin{array}{c} < \\ > \\ <> \end{array} \right\} m_2$$

$m_1 < m_2$: Since $m_1 < m_2$ and m_1 and m_2 are synchronous with respect to each other ($s_1 = s_2$), m_1 *retDep* m_2.

Serializability of Top-Level Transactions: This property does not need to be considered in this case. Since $m_1 < m_2$, both messages cannot belong to different top-level transactions.

Serializability of Transactional Partial Threads: This property does not need to be considered in this case. Since $s_1 = s_2$, both messages cannot belong to different threads.

Synchronization of Non-Transactional Messages and Top-Level Transactions: m_1 transactional and m_2 non-transactional is not possible since $m_1 < m_2$. Therefore, assume that m_1 is non-transactional and m_2 is transactional. Then, $m_1 \leq tl_2 \leq m_2$. Since m_1 *retDep* m_2, also m_1 *retDep* tl_2, according to the lemma of Section 4.4.1.5. Thus, Property 1c does not require synchronization.

Synchronization of Non-Transactional Messages: Since m_1 *retDep* m_2, Property 1d does not require synchronization.

$m_1 > m_2$: This case is not possible. m_1 cannot have started before m_2 if $m_1 > m_2$.

$m_1 <> m_2$: See Figure 4.11.

Serializability of Top-Level Transactions: See Figure 4.11 (a). Assume $tl_1 \neq tl_2$. Since $s_1 = s_2$, tl_1 must have committed before tl_1 has returned a result, in particular before tl is sent. This is a serial schedule "tl_1 before tl_2".

Serializability of Transactional Partial Threads: This property does not need to be considered in this case since $s_1 = s_2$.

Synchronization of Non-Transactional Messages and Top-Level Transactions: See Figure 4.11 (b). If m_1 is transactional and m_2 is non-transactional then tl_1 has committed before m_2 is sent. This is a serial schedule "tl_1 before m_2'".

See Figure 4.11 (c). If m_1 is non-transactional and m_2 is transactional, then m_1 has finished execution before tl_2 is sent. This is a serial schedule "m_1 before tl_2".

Synchronization of Non-Transactional Messages: See Figure 4.11 (d). m_1 has finished execution before m_2 is sent. This is the serial schedule "m_1 before m_2'".

4.4.4.2 Cascading Aborts

Cascading aborts can only occur if both m_1 and m_2 are transactional. Consider the following subcases.

$$tl_1 \left\{ \begin{array}{c} \neq \\ = \end{array} \right\} tl_2$$

$tl_1 \neq tl_2$: tl_1 has committed or aborted before tl_2 (and for that reason m_2) is sent. This is because synchronous transaction creating messages return only after they commit or abort. Therefore, tl_2 cannot see uncommitted state of tl_1.

$tl_1 = tl_2$: Consider four subcases.

$$t_1 \left\{ \begin{array}{c} = \\ < \\ > \\ <> \end{array} \right\} t_2$$

$t_1 = t_2$: An abort of t_1 is equivalent to an abort of t_2.

$t_1 < t_2$: If t_1 aborts then t_2 must be aborted, too. Since it is a descendant transaction, this is not a cascading abort.

$t_1 > t_2$: See Figure 4.12 (a). Since m_1 and m_2 are synchronous with respect to each other, $1LBLCAT(m_1, m_2)$ must have returned. Since transaction creating synchronous message return only after they have committed, $1LBLCAT(m_1, m_2)$ has committed. Therefore, cascading aborts are avoided (refer back to Section 4.4.2).

$t_1 <> t_2$: See Figure 4.12 (b). Since m_1 and m_2 are synchronous with respect to each other, $1LBLCAT(m_1, m_2)$ must have returned and therefore committed. Cascading aborts are avoided.

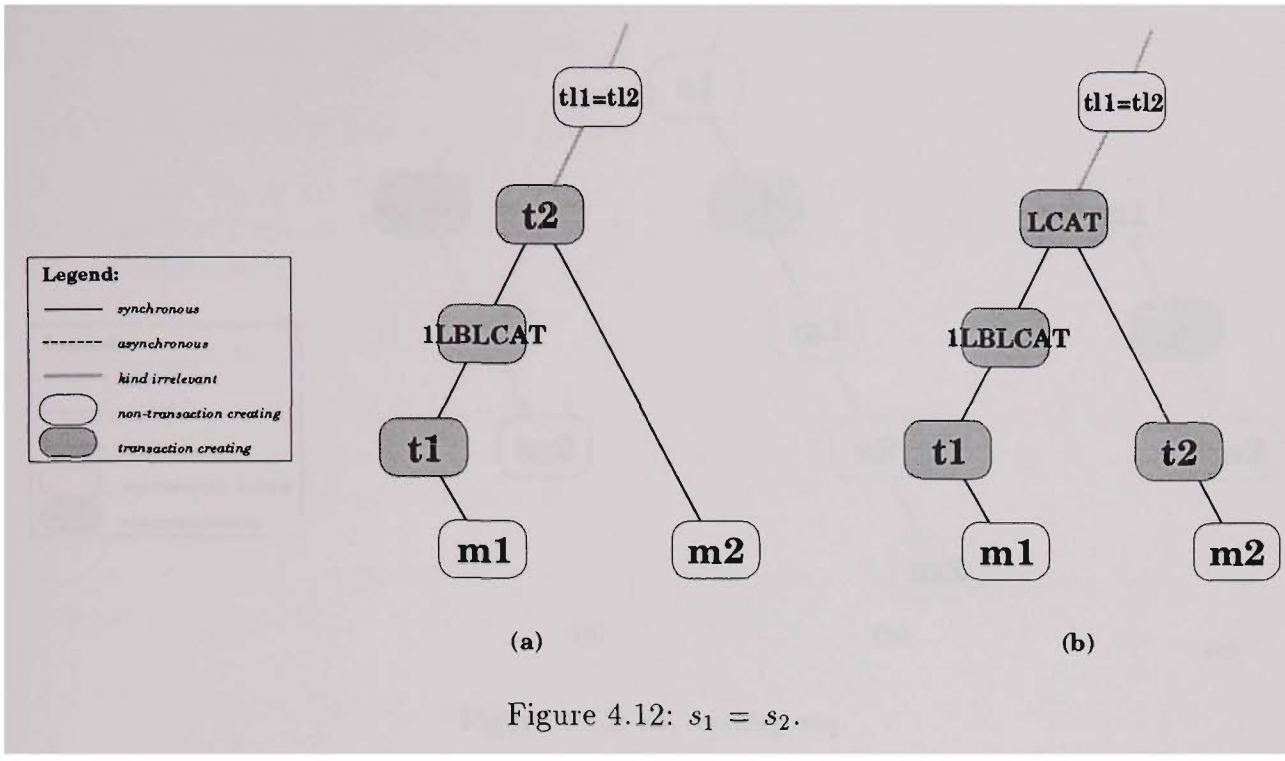

Figure 4.12: $s_1 = s_2$.

4.4.4.3 Return Dependencies

m is schedulable unconditionally. Therefore schedulability is guaranteed in the return dependency case.

4.4.4.4 Concurrency

m is schedulable unconditionally. This is trivially the earliest possible schedule.

4.4.5 m_1 *retDep* m_2

In this case, the schedulability predicate for m_2 is satisfied unconditionally. See Figure 4.13.

4.4.5.1 Schedules

Serializability of Top-Level Transactions: See Figure 4.13 (a). $m_1 < m_2$ since m_1 *retDep* m_2. Therefore, if both m_1 and m_2 are transactional, $tl_1 = tl_2$. Therefore, serializability between tl_1 and tl_2 is not required.

Serializability of Transactional Partial Threads: See Figure 4.13 (b).

1. Since $s_2/LCAT(m_1, m_2)$ is synchronous with respect to m_1, $s_1/LCAT(m_1, m_2)$ *retDep* m_1.
2. Since m_1 *retDep* m_2 and $m_1 \leq s_2/LCAT(m_1, m_2) \leq m_2$, m_1 *retDep* $s_2/LCAT(m_1, m_2)$.
3. Since (1) and (2), $s_1/LCAT(m_1, m_2)$ *retDep* $s_2/LCAT(m_1, m_2)$.
4. Since (3), no serializability is required.

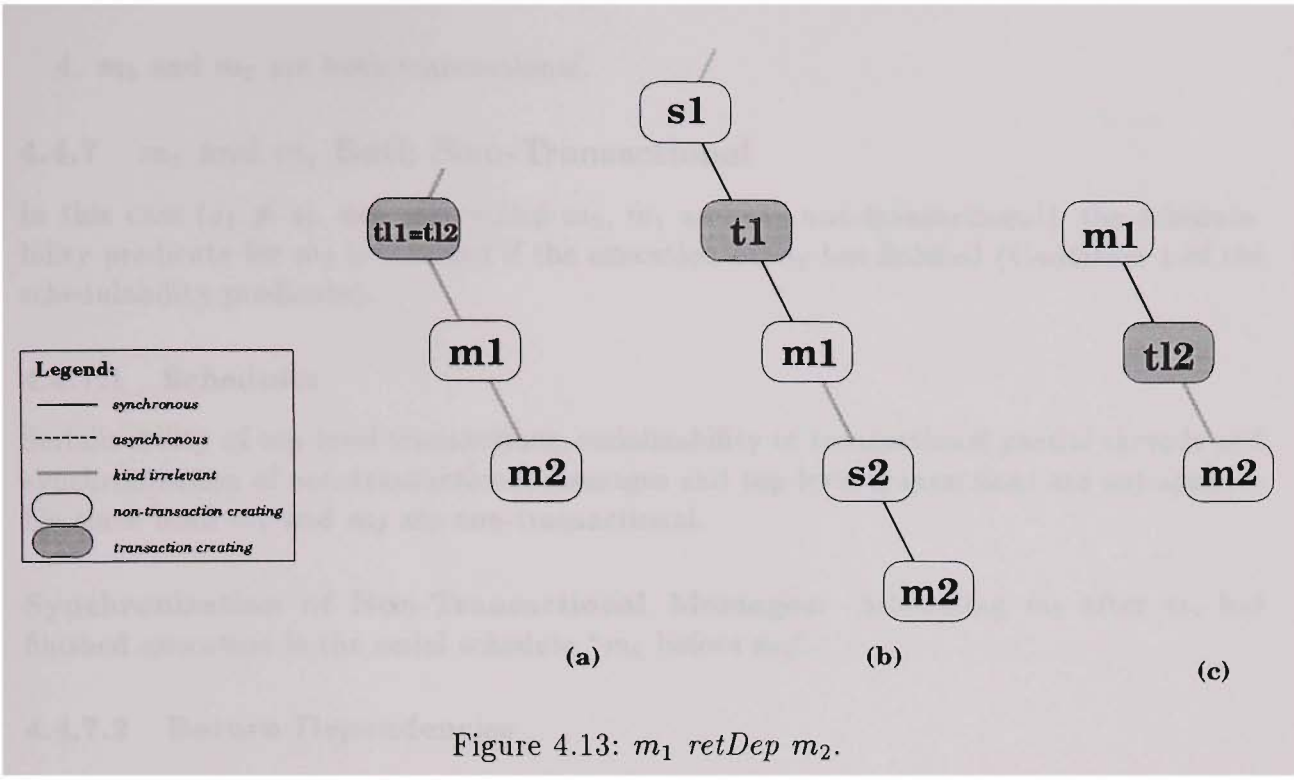

Figure 4.13: m_1 *retDep* m_2.

Synchronization of Non-Transactional Messages and Top-Level Transactions: See Figure 4.13 (c). The case that m_1 is transactional and m_2 is non-transactional is not possible since $m_1 \leq m_2$. Therefore, consider the case that m_1 is non-transactional and m_2 is transactional. Since m_1 *retDep* m_2 and $m_1 \leq tl_2 \leq m_2$, m_1 *retDep* tl_2. Therefore, synchronization is not required.

Synchronization of Non-Transactional Messages: Since m_1 *retDep* m_2, synchronization is not required.

4.4.5.2 Return Dependencies

The schedulability condition is satisfied unconditionally.

4.4.5.3 Cascading Aborts

Assume that both m_1 and m_2 are transactional. Then, $t_1 \leq t_2$ since $m_1 \leq m_2$. If t_1 aborts then t_2 must be aborted due to the semantics of nested transactions—a cascading abort can not occur.

4.4.5.4 Concurrency

Since the schedulability condition is satisfied unconditionally, m_2 can be scheduled immediately which is the earliest possible schedule.

4.4.6 $s_1 \neq s_2$ and not m_1 *retDep* m_2

Consider four subcases.

1. m_1 and m_2 are both non-transactional;
2. m_1 is transactional and m_2 is non-transactional;
3. m_1 is non-transactional and m_2 is transactional;

4. m_1 and m_2 are both transactional.

4.4.7 m_1 and m_2 Both Non-Transactional

In this case ($s_1 \neq s_2$, not m_1 *retDep* m_2, m_1 and m_2 non-transactional), the schedulability predicate for m_2 is satisfied if the execution of m_1 has finished (Condition 1 of the schedulability predicate).

4.4.7.1 Schedules

Serializability of top-level transactions, serializability of transactional partial threads and synchronization of non-transactional messages and top-level transactions are not applicable since both m_1 and m_2 are non-transactional.

Synchronization of Non-Transactional Messages: Scheduling m_2 after m_1 has finished execution is the serial schedule "m_1 before m_2".

4.4.7.2 Return Dependencies

Not m_1 *retDep* m_2.

4.4.7.3 Cascading Aborts

Since m_1 and m_2 are non-transactional, aborts are not an issue.

4.4.7.4 Concurrency

Assume a weaker schedulability predicate, i.e. m_2 is scheduled before m_1 has finished execution. Then, without using application-specific knowledge, it cannot be ruled out that variable accesses of m_1 and m_2 interleave in a way that defies synchronization. For example, m_1 may read a variable that has been written by m_1 but that has been overwritten by m_2.

4.4.8 m_1 Transactional and m_2 Non-Transactional

In this case ($s_1 \neq s_2$, not m_1 *retDep* m_2, m_1 transactional, m_2 non-transactional), the schedulability predicate for m_2 is satisfied if t_1 has aborted or tl_1 has committed (Condition 2 of the schedulability predicate). See Figure 4.14 (a).

4.4.8.1 Schedules

Since m_1 is transactional and m_1 is non-transactional, serializability of top-level transactions, serializability of transactional partial threads and synchronization of non-transactional messages are not applicable.

Synchronization of Non-Transactional Messages and Top-Level Transactions: If t_1 aborts then all of its effects, including the effects of m_1, are undone as if t_1 had not happened at all. If tl_1 commits then this schedule is equivalent to the serial schedule "tl_1 before m_2".

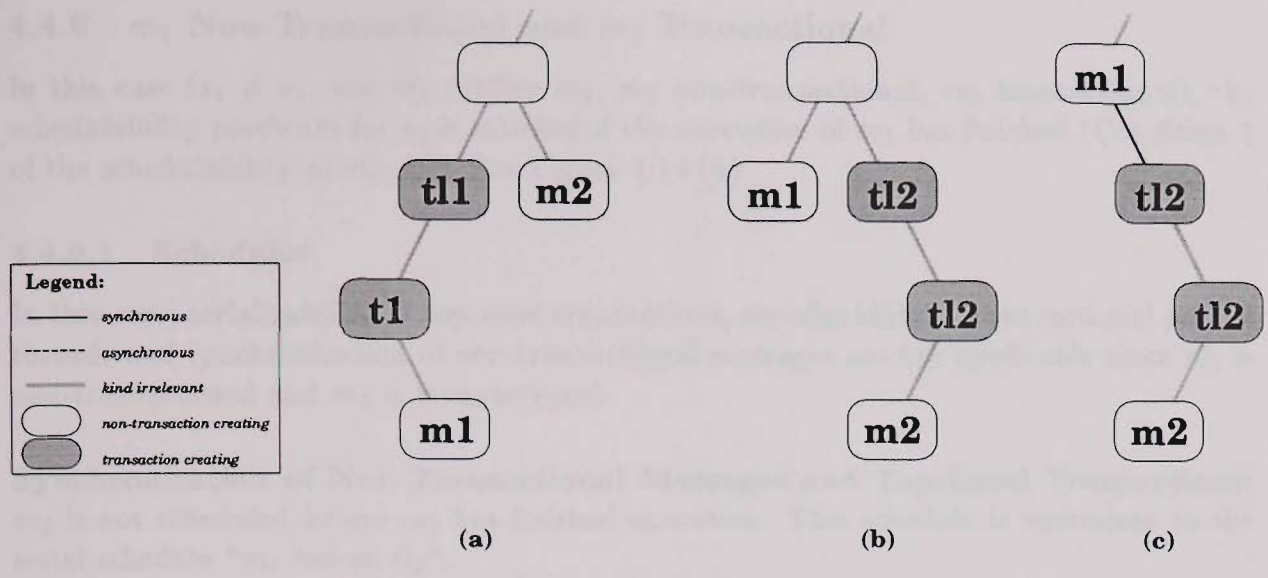

Figure 4.14: One of the two messages is transactional, the other one is non-transactional.

4.4.8.2 Return Dependencies

Consider three subcases.

$$m_1 \left\{ \begin{array}{c} < \\ > \\ <> \end{array} \right\} m_2$$

$m_1 < m_2$: This case is not possible. If $m_1 < m_2$ then m_2 must be transactional, too.

$m_1 > m_2$: This case is not possible. If $m_1 > m_2$ then m_1 could not have started execution before m_2.

$m_1 <> m_2$: See Figure 4.14 (a) again. Since m_2 is non-transactional and $m_1 <> m_2$, also $tl_1 <> m_2$. Return dependencies can only occur between ancestor and descendant messages.

4.4.8.3 Cascading Aborts

Cascading aborts are not an issue in this case since m_2 is non-transactional.

4.4.8.4 Concurrency

Assume a weaker schedulability predicate and consider two cases.

1. t_1 aborts and m_2 is scheduled before the abort of t_1. Without employing application-specific knowledge, it cannot be ensured that all effects of the aborting transaction t_1 are undone. For example, m_2 may read a variable that has been written by m_1.

2. tl_1 commits and m_2 is scheduled before the commit of tl_1. Without employing application-specific knowledge, it cannot be avoided that variable accesses of t_1 and m_2 interleave in a way that violates synchronization. For example, m_2 may overwrite a variable that has been written by m_1 which is, subsequently, read by another message m_1' which belongs to tl_1 or any of its descendant transactions.

4.4.9 m_1 Non-Transactional and m_2 Transactional

In this case ($s_1 \neq s_2$, not m_1 *retDep* m_2, m_1 non-transactional, m_2 transactional), the schedulability predicate for s_2 is satisfied if the execution of m_1 has finished (Condition 1 of the schedulability predicate). See Figure 4.14 (b)

4.4.9.1 Schedules

In this case, serializability of top-level transactions, serializability of transactional partial threads and synchronization of non-transactional messages are not applicable since m_1 is non-transactional and m_2 is transactional.

Synchronization of Non-Transactional Messages and Top-Level Transactions: m_2 is not scheduled before m_1 has finished execution. This schedule is equivalent to the serial schedule "m_1 before tl_2".

4.4.9.2 Return Dependencies

See Figure 4.14 (c). In this case, m_1 cannot be in return dependency with tl_2 since m_1 is not in return dependency with m_2.

Assume the opposite, i.e. that m_1 was in return dependency with tl_2. Then, tl_2 must be in the following relationship with m_1: $tl_2 = m_1$ *synch-nonTrans synchTrans*. Note that tl_2 is top-level and therefore the first transaction creating message in m_2's message path. Since $tl_2 \leq m_2$, $m_2 = tl_2$ *any**. Thus, $m_2 = m_1$ *synch-nonTrans synchTrans any**, a contradiction to not m_1 *retDep* m_2.

4.4.9.3 Cascading Aborts

Since m_1 is non-transactional, cascading aborts are not an issue.

4.4.9.4 Concurrency

Assume a weaker schedulability predicate, i.e. m_2 is scheduled before m_1 has finished execution. Then, without employing application-specific knowledge, it cannot be ruled out that conflicting variable accesses of m_1 and m_2 violate the synchronization property, e.g. m_2 may write a variable that has been written by m_1 and is subsequently read by m_1.

4.4.10 m_1 and m_2 Both Transactional

Consider two subcases.

$$tl_1 \left\{ \begin{array}{c} \neq \\ = \end{array} \right\} tl_2$$

4.4.11 $tl_1 \neq tl_2$

In this case ($s_1 \neq s_2$, not m_1 *retDep* m_2, m_1 and m_2 transactional, $tl_1 \neq tl_2$), the schedulability predicate for m_2 is satisfied if t_1 has aborted or tl_1 has committed (Condition 3a of the schedulability predicate). See Figure 4.15.

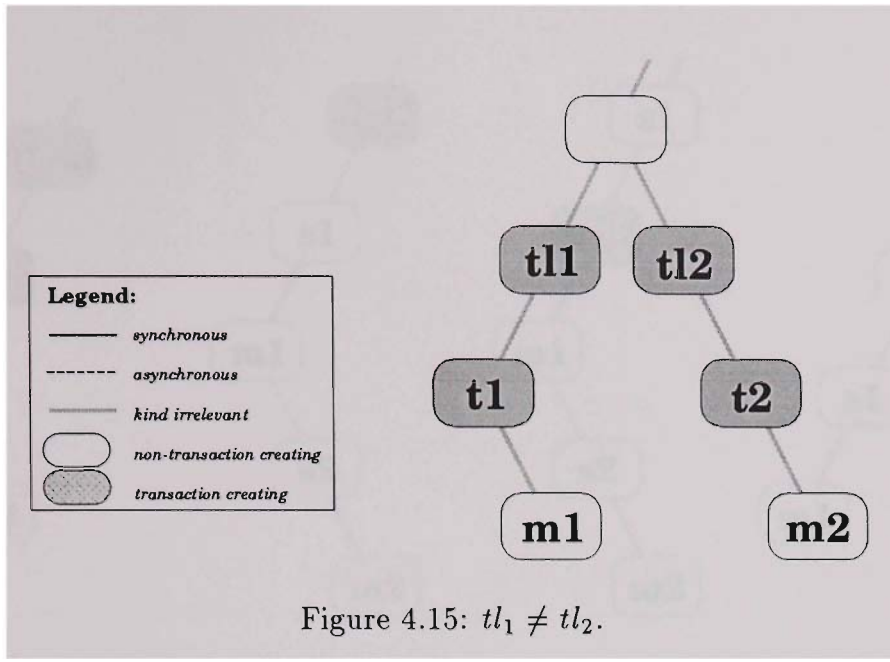

Figure 4.15: $tl_1 \neq tl_2$.

4.4.11.1 Schedules

Serializability of Top-Level Transactions: Consider two cases.

1. If t_1 aborts then all of its effects are undone, including the effects of m_1, as if t_1 had never executed.

2. If tl_1 commits then m_2 is not scheduled before tl_1 has committed. This schedule is equivalent to the serial schedule "tl_1 before tl_2".

In this case, serializability of transactional partial threads, synchronization of non-transactional messages and top-level transactions and synchronization of non-transactional messages are not an issue since both m_1 and m_2 are transactional but belong to different top-level transactions.

4.4.11.2 Return Dependencies

$tl_1 <> tl_2$, since $tl_1 \neq tl_2$. Otherwise, one of the two transactions would be a descendant of the other and therefore not be top-level. Since there can only be a return dependency between an ancestor and a descendant, tl_1 and tl_2 cannot be in a return dependency.

4.4.11.3 Cascading Aborts

In case of an abort of t_1, all effects of t_1 are undone and therefore tl_2 cannot see uncommitted state of tl_1—cascading aborts cannot occur.

After a top-level transaction has committed, it cannot be subsequently aborted—cascading aborts cannot occur.

4.4.11.4 Concurrency

Assume a weaker schedulability condition and consider two cases.

1. Assume that t_1 aborts and m_2 is scheduled before t_1 has aborted. Then, serializability of tl_1 and tl_2 cannot be ensured without employing additional application-specific knowledge. For example, m_2 may read a variable that m_1 has written.

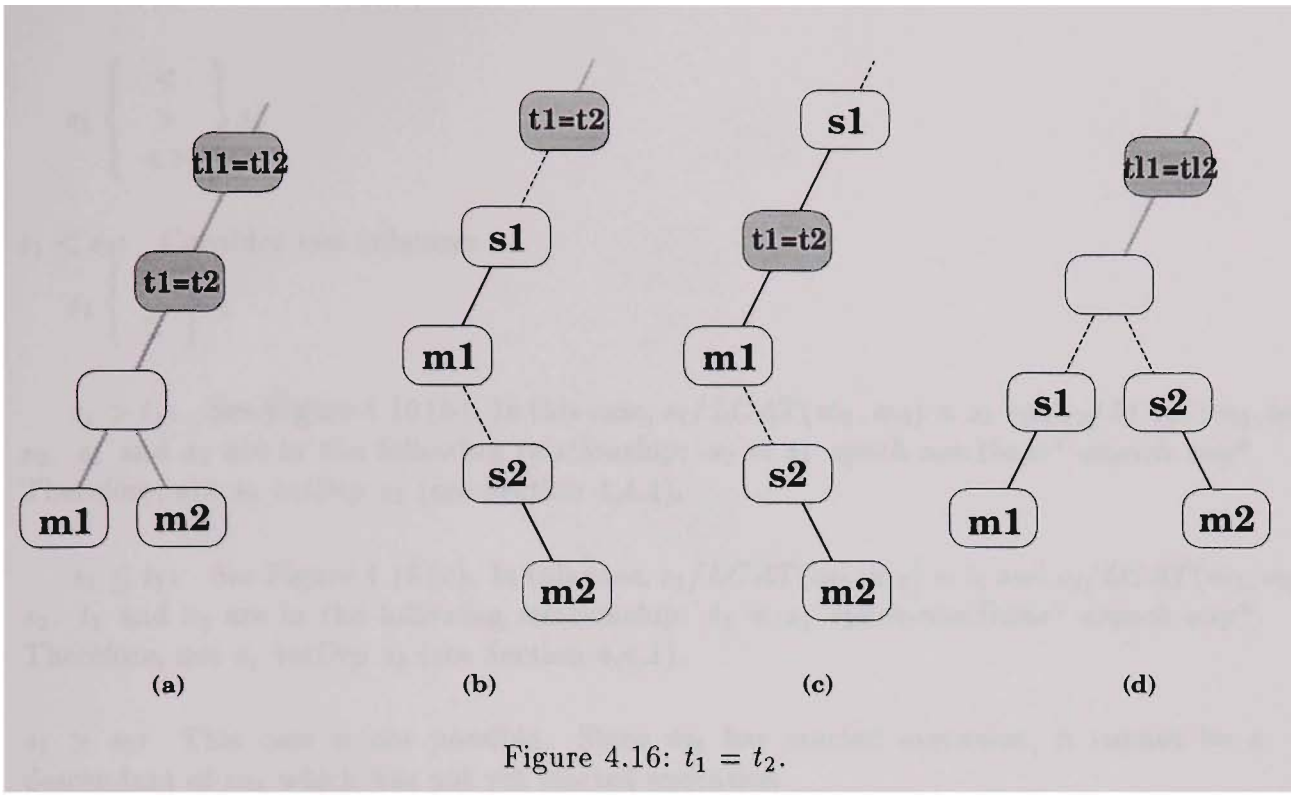

Figure 4.16: $t_1 = t_2$.

2. Assume that tl_1 commits and m_2 is scheduled before tl_1 has committed. Then, serializability of tl_1 and tl_2 cannot be ensured without employing additional application-specific knowledge. For example, m_2 may read a variable that m_1 has written.

4.4.12 $tl_1 = tl_2$

Consider four subcases.

$$t_1 \left\{ \begin{array}{c} = \\ < \\ > \\ <> \end{array} \right\} t_2$$

4.4.13 $t_1 = t_2$

In this case ($s_1 \neq s_2$, not m_1 *retDep* m_2, m_1 and m_2 transactional, $tl_1 = tl_2$, $t_1 = t_2$), the schedulability predicate for m_2 is satisfied if s_1/t_1 has finished execution (Condition 3(b)i of the schedulability predicate). See Figure 4.16.

4.4.13.1 Schedules

In this case, serializability of top-level transactions, synchronization of non-transactional messages and top-level transactions and synchronization of non-transactional messages are not an issue since $tl_1 = tl_2$.

Serializability of Transactional Partial Threads: Since $t_1 = t_2$, $LCAT(m_1, m_2) = t_1$. m_2 is not scheduled before s_1/t_1 ($= s_1/LCAT(m_1, m_2)$) has finished execution. This schedule is equivalent to the serial schedule "$s_1/LCAT(m_1, m_2)$ before $s_2/LCAT(m_1, m_2)$".

4.4.13.2 Return Dependencies

In this case, there cannot be a return dependency between $s_1/LCAT(m_1, m_2)$ and $s_2/LCAT(m_1, m$
Consider three subcases.

$$s_1 \left\{ \begin{array}{c} < \\ > \\ <> \end{array} \right\} s_2$$

$s_1 < s_2$: Consider two subcases.

$$s_1 \left\{ \begin{array}{c} > \\ \leq \end{array} \right\} t_1$$

$s_1 > t_1$: See Figure 4.16 (b). In this case, $s_1/LCAT(m_1, m_2) = s_1$ and $s_2/LCAT(m_1, m_2) = s_2$. s_1 and s_2 are in the following relationship: $s_2 = s_1$ *synch-nonTrans* asynch any**. Therefore, not s_1 *retDep* s_2 (see Section 4.4.1).

$s_1 \leq t_1$: See Figure 4.16 (c). In this case, $s_1/LCAT(m_1, m_2) = t_1$ and $s_2/LCAT(m_1, m_2) = s_2$. t_1 and s_2 are in the following relationship: $t_2 = s_1$ *synch-nonTrans* asynch any**. Therefore, not s_1 *retDep* s_2 (see Section 4.4.1).

$s_1 > s_2$: This case is not possible. Since m_1 has started execution, it cannot be a descendant of m_2 which has not yet started execution.

$s_1 <> s_2$: See Figure 4.16 (d). $s_1/LCAT(m_1, m_2) = s_1$, $s_2/LCAT(m_1, m_2) = s_2$. Since there can only be a return dependency between ancestors and descendants, $s_1/LCAT(m_1, m_2)$ and $s_2/LCAT(m_1, m_2)$ are not in a return dependency.

4.4.13.3 Cascading Aborts

Since $t_1 = t_2$, cascading aborts are not an issue.

4.4.13.4 Concurrency

Assume a weaker schedulability predicate, i.e. m_2 is scheduled before s_1/t_1 has finished execution. Then, serializability between $s_1/LCAT(m_1, m_2)$ and $s_2/LCAT(m_1, m_2)$ cannot be ensured, without employing application-specific knowledge. For example, m_1 may have written a variable, m_2 overrides this variable and another message m_1' that belongs to m_1/t_1 subsequently reads this variable.

4.4.14 $t_1 < t_2$

In this case ($s_1 \neq s_2$, not m_1 *retDep* m_2, m_1 and m_2 transactional, $tl_1 = tl_2$, $t_1 < t_2$), the schedulability predicate for m_2 is satisfied if s_1/t_1 has finished execution or s_1/t_1 *retDep* m_2 (Condition 3(b)ii of the schedulability predicate). See Figure 4.17.

4.4.14.1 Schedules

In this case, serializability of top-level transactions, synchronization of non-transactional messages and top-level transactions and synchronization of non-transactional messages are not an issue since $tl_1 = tl_2$.

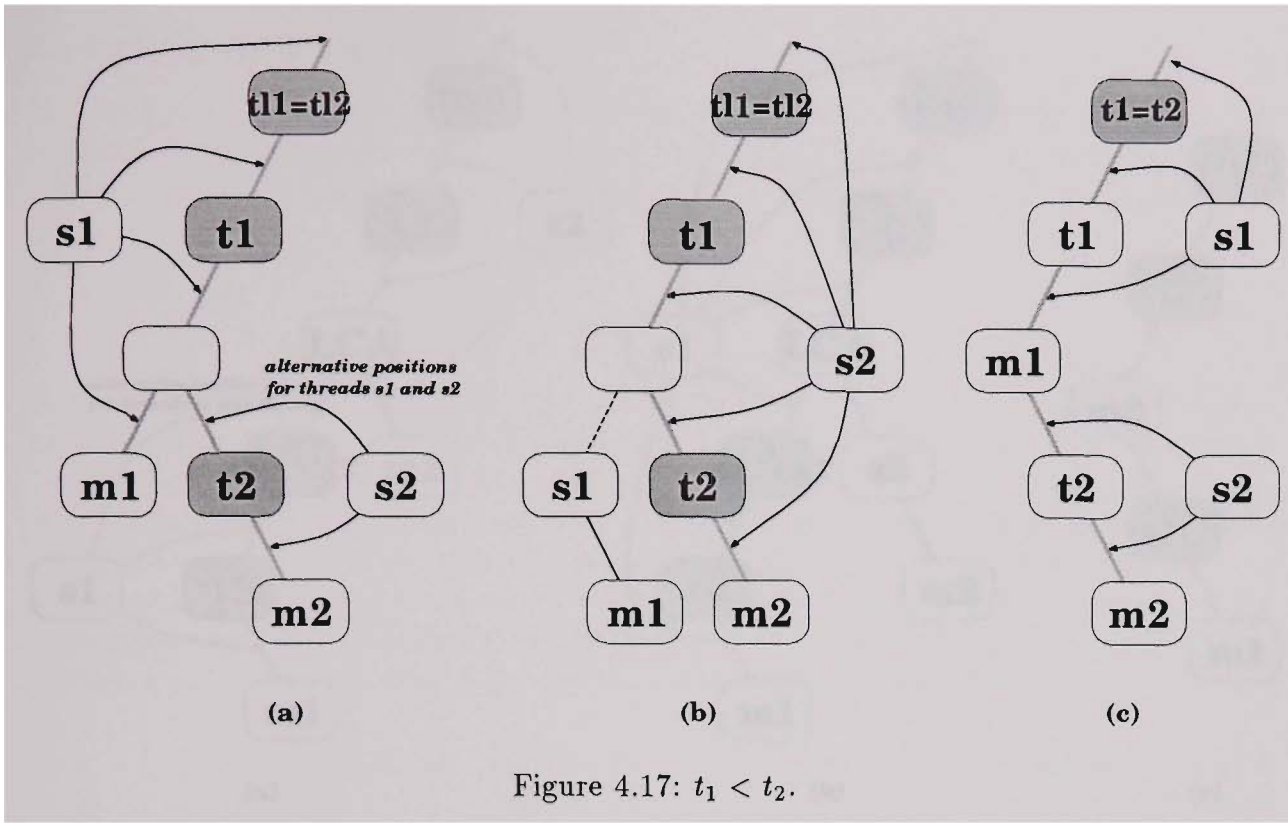

Figure 4.17: $t_1 < t_2$.

Serializability of Transactional Partial Threads: Since $t_1 < t_2$, $LCAT(m_1, m_2) = t_1$. Consider two subcases.

1. s_1/t_1 *retDep* s_2/t_1;
2. not s_1/t_1 *retDep* s_2/t_1.

s_1/t_1 *retDep* s_2/t_1: In this case, $s_1/LCAT(m_1, m_2)$ *retDep* $s_2/LCAT(m_1, m_2)$ and therefore serializability is not required.

not s_1/t_1 *retDep* s_2/t_1: In this case, m_2 is not scheduled before s_1/t_1 has finished execution. This schedule is equivalent to the schedule "$s_1/LCAT(m_1, m_2)$ before $s_2/LCAT(m_1, m_2)$".

4.4.14.2 Return Dependencies

If $s_1/LCAT(m_1, m_2)$ *retDep* $s_2/LCAT(m_1, m_2)$ then the schedulability predicate for m_2 is satisfied.

4.4.14.3 Cascading Aborts

If t_1 aborts then t_2 must be aborted, too, due to the semantics of nested transaction aborts—no cascading aborts can occur.

4.4.14.4 Concurrency

Consider two subcases.

1. s_1/t_1 *retDep* s_2/t_1;
2. not s_1/t_1 *retDep* s_2/t_1.

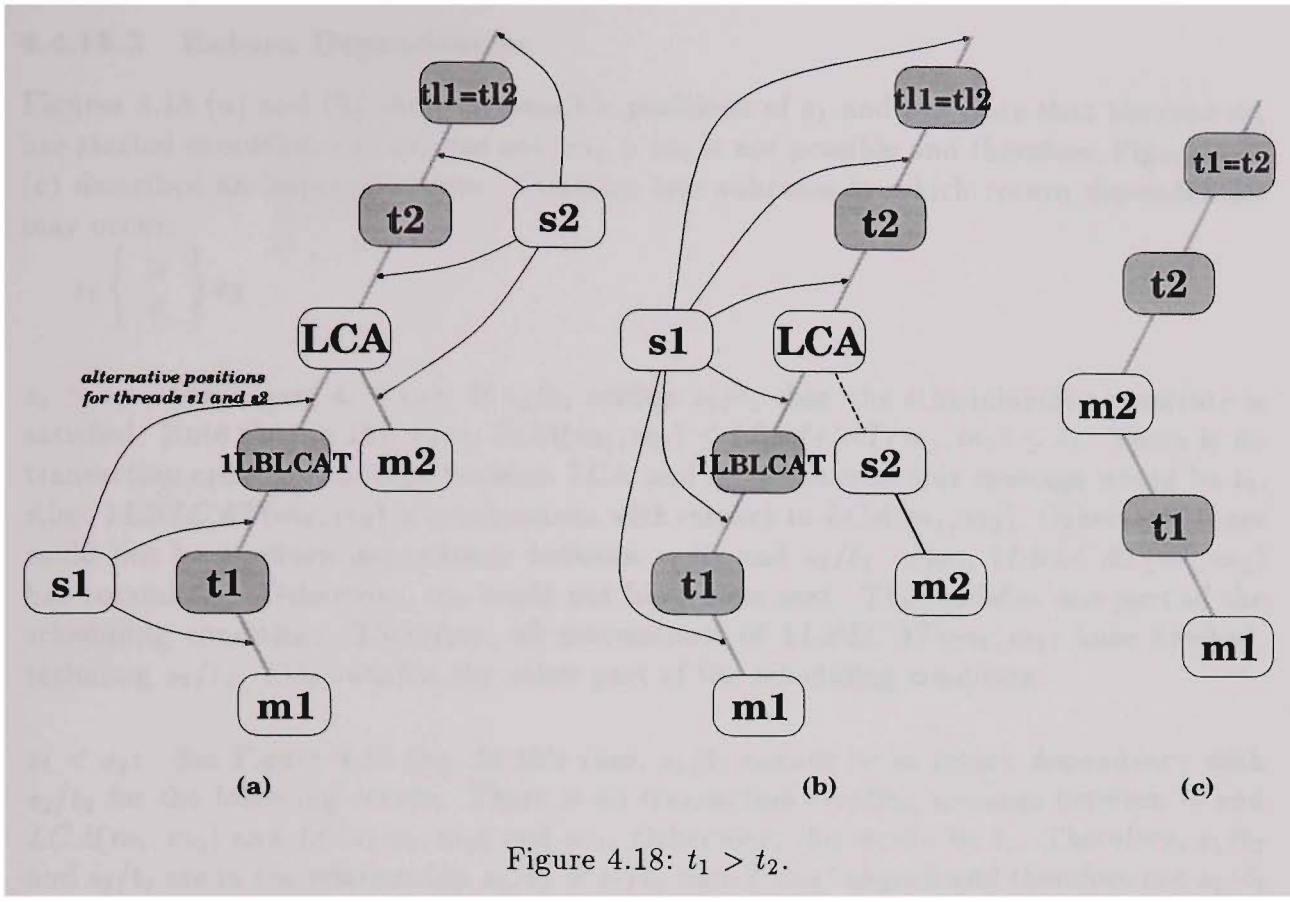

Figure 4.18: $t_1 > t_2$.

s_1/t_1 *retDep* s_2/t_1: In this case, m_2 is schedulable immediately which is trivially the earliest possible schedule.

not s_1/t_1 *retDep* s_2/t_1: Consider a weaker schedulability predicate, i.e. m_2 is schedulable before s_1/t_1 has finished execution. Then, the serializability of $s_1/LCAT(m_1, m_2)$ and $s_2/LCAT(m_1, m_2)$ cannot be ensured without using application-specific knowledge. For example, m_1 may write to a variable which m_2 overrides and another message m_1' belonging to s_1/t_1 subsequently reads this variable.

4.4.15 $t_1 > t_2$

In this case ($s_1 \neq s_2$, not m_1 *retDep* m_2, m_1 and m_2 transactional, $tl_1 = tl_2$, $t_1 > t_2$), the schedulability predicate for m_2 is satisfied if $1LBLCAT(m_1, m_2)$ has committed and the execution of s_1/t_2 has finished (Condition 3(b)iii of the schedulability predicate). See Figure 4.18.

4.4.15.1 Schedules

In this case, serializability of top-level transactions, synchronization of non-transactional messages and top-level transactions and synchronization of non-transactional messages are not an issue since $tl_1 = tl_2$. See Figure 4.18.

Serializability of Transactional Partial Threads: Since $t_1 > t_2$, $LCAT(m_1, m_2) = t_2$. m_2 is not scheduled unless s_1/t_2 has finished execution. This schedule is equivalent to the serial schedule "$s_1/LCAT(m_1, m_2)$ before $s_2/LCAT(m_1, m_2)$".

4.4.15.2 Return Dependencies

Figures 4.18 (a) and (b) show all possible positions of s_1 and s_2. Note that because m_1 has started execution and m_2 has not, $m_1 > m_2$ is not possible and therefore, Figure 4.18 (c) describes an impossible case. Consider two subcases in which return dependencies may occur.

$$s_1 \left\{ \begin{array}{c} > \\ < \end{array} \right\} s_2$$

$s_1 > s_2$: See Figure 4.18 (a). If s_2/t_2 *retDep* s_1/t_2 then the schedulability predicate is satisfied. Note that in this case, $LCA(m_1, m_2) \leq 1LBLCAT(m_1, m_2) \leq s_1$. There is no transaction creating message between LCA and t_2. Otherwise this message would be t_2. Also, $1LBLCAT(m_1, m_2)$ is synchronous with respect to $LCA(m_1, m_2)$. Otherwise, there could not be a return dependency between s_2/t_2 and s_1/t_2. Also, $1LBLCAT(m_1, m_2)$ has committed. Otherwise, m_2 could not have been sent. This satisfies one part of the scheduling condition. Therefore, all descendants of $1LBLCAT(m_1, m_2)$ have finished, including s_1/t_2. This satisfies the other part of the scheduling condition.

$s_1 < s_2$: See Figure 4.18 (b). In this case, s_1/t_2 cannot be in return dependency with s_2/t_2 for the following reason. There is no transaction creating message between t_2 and $LCA(m_1, m_2)$ and $LCA(m_1, m_2)$ and m_2. Otherwise, this would be t_2. Therefore, s_1/t_2 and s_2/t_2 are in the relationship $s_2/t_2 = s_1/t_2$ *non-Trans* asynch* and therefore not s_1/t_2 *retDep* s_2/t_2.

4.4.15.3 Cascading Aborts

Since it is ensured that $1LBLCAT(m_1, m_2)$ has committed before m_2 is scheduled, cascading aborts are avoided.

4.4.15.4 Concurrency

Assume a weaker schedulability predicate, i.e. m_2 is schedulable before $1LBLCAT(m_1, m_2)$ has committed or before s_1/t_2 has finished. Then, in both cases, it cannot be rules out without employing application-specific knowledge, that scheduling properties are violated.

1. Assume that m_2 is scheduled before $1LBLCAT(m_1, m_2)$ has committed. Then, m_2 can read variables that have been written by m_1. If then t_1 aborts subsequently due to the abort of $1LBLCAT(m_1, m_2)$ or any of its descendant transactions, then t_2 must be aborted as well because it has seen uncommitted state of t_1—a cascading abort.

2. Assume that m_2 is scheduled before s_1/t_2 has finished. Then, m_2 could overwrite variables written by m_1 which are subsequently read by another message m_1' that belongs to s_1/t_2. Then, serializability of $s_1/LCAT(m_1, m_2)$ and $s_2/LCAT(m_1, m_2)$ is defied.

4.4.16 $t_1 <> t_2$

In this case ($s_1 \neq s_2$, not m_1 *retDep* m_2, m_1 and m_2 transactional, $tl_1 = tl_2$, $t_1 <> t_2$), the schedulability predicate for m_2 is satisfied if $1LBLCAT(m_1, m_2)$ has committed and either the execution of $s_1/LCAT(m_1, m_2)$ has finished or $s_1/LCAT(m_1, m_2)$ retDep $s_2/LCAT(m_1, m_2)$ (Condition 3(b)iv of the schedulability predicate). See Figure 4.19.

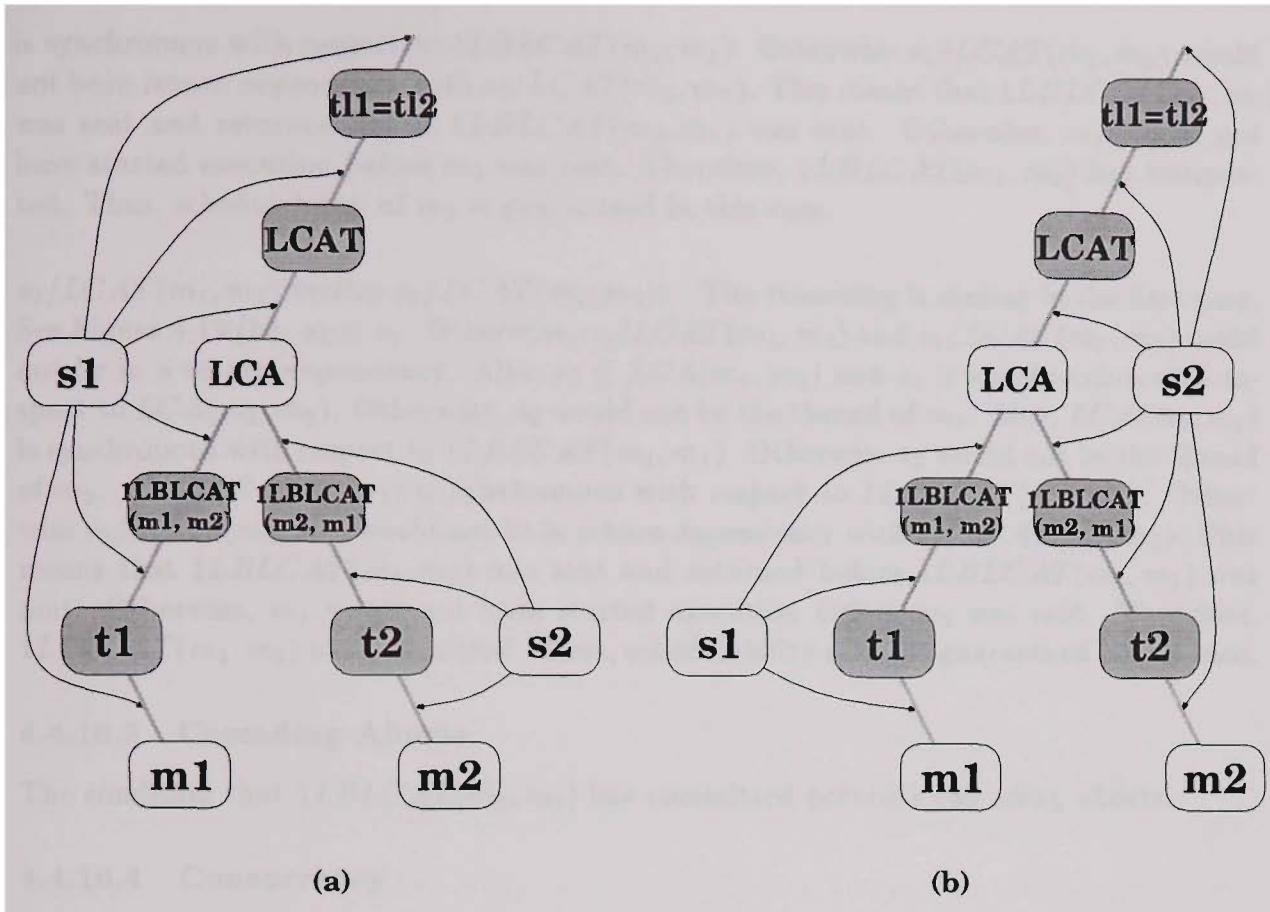

Figure 4.19: $t_1 <> t_2$.

4.4.16.1 Schedules

In this case, serializability of top-level transactions, synchronization of non-transactional messages and top-level transactions and synchronization of non-transactional messages are not an issue since $tl_1 = tl_2$.

Serializability of Transactional Partial Threads: If $s_1/LCAT(m_1, m_2)$ *retDep* $s_2/LCAT(m_1, m_2)$ then serializability of $s_1/LCAT(m_1, m_2)$ and $s_2/LCAT(m_1, m_2)$ is not required.

If not $s_1/LCAT(m_1, m_2)$ *retDep* $s_2/LCAT(m_1, m_2)$ then m_2 is not scheduled before $s_1/LCAT(m_1, m_2)$ has finished execution. This schedule is equivalent to the schedule "$s_1/LCAT(m_1, m_2)$ before $s_2/LCAT(m_1, m_2)$".

4.4.16.2 Return Dependencies

Consider two subcases.

1. $s_1/LCAT(m_1, m_2)$ *retDep* $s_2/LCAT(m_1, m_2)$;
2. $s_2/LCAT(m_1, m_2)$ *retDep* $s_1/LCAT(m_1, m_2)$.

$s_1/LCAT(m_1, m_2)$ *retDep* $s_2/LCAT(m_1, m_2)$: See Figure 4.19 (a). Then, $s_1 < s_2$. Otherwise, $s_1/LCAT(m_1, m_2)$ and $s_2/LCAT(m_1, m_2)$ could not be in a return dependency. Also, $s_1 \leq LCA(m_1, m_2)$ and s_1 is synchronous with respect to $LCA(m_1, m_2)$. Otherwise, s_1 would not be the thread of m_1. Also, $LCA(m_1, m_2)$ is synchronous with respect to $1LBLCAT(m_1, m_2)$. Otherwise s_1 would not be the thread of m_1. Also, $LCA(m_1, m_2)$

is synchronous with respect to $1LBLCAT(m_2, m_1)$. Otherwise $s_1/LCAT(m_1, m_2)$ would not be in return dependency with $s_2/LCAT(m_1, m_2)$. This means that $1LBLCAT(m_1, m_2)$ was sent and returned before $1LBLCAT(m_2, m_1)$ was sent. Otherwise, m_1 would not have started execution before m_2 was sent. Therefore, $1LBLCAT(m_1, m_2)$ has committed. Thus, schedulability of m_2 is guaranteed in this case.

$s_2/LCAT(m_1, m_2)$ *retDep* $s_1/LCAT(m_1, m_2)$: The reasoning is similar to the first case. See Figure 4.19 (b). $s_2 < s_1$. Otherwise, $s_2/LCAT(m_1, m_2)$ and $s_1/LCAT(m_1, m_2)$ could not be in a return dependency. Also, $s_2 \leq LCA(m_1, m_2)$ and s_2 is synchronous with respect to $LCA(m_1, m_2)$. Otherwise, s_2 would not be the thread of m_2. Also, $LCA(m_1, m_2)$ is synchronous with respect to $1LBLCAT(m_2, m_1)$. Otherwise s_2 would not be the thread of m_2. Also, $LCA(m_1, m_2)$ is synchronous with respect to $1LBLCAT(m_1, m_2)$. Otherwise $s_2/LCAT(m_1, m_2)$ would not be in return dependency with $s_1/LCAT(m_1, m_2)$. This means that $1LBLCAT(m_1, m_2)$ was sent and returned before $1LBLCAT(m_2, m_1)$ was sent. Otherwise, m_1 would not have started execution before m_2 was sent. Therefore, $1LBLCAT(m_1, m_2)$ has committed. Thus, schedulability of m_2 is guaranteed in this case.

4.4.16.3 Cascading Aborts

The condition that $1LBLCAT(m_1, m_2)$ has committed prevents cascading aborts.

4.4.16.4 Concurrency

Assume a weaker schedulability predicate, i.e. m_2 is scheduled either before $1LBLCAT(m_1, m_2)$ has committed or, in case that not $s_1/LCAT(m_1, m_2)$ *retDep* $s_2/LCAT(m_1, m_2)$ before $s_1/LCAT(m_1, m_2)$ has finished. Then, in both cases, it cannot be ruled out without employing application-specific knowledge, that scheduling properties are violated.

1. If m_2 is scheduled before $1LBLCAT(m_1, m_2)$ has committed then m_2 might read a variable that has been written by m_1. If $1LBLCAT(m_1, m_2)$ aborts subsequently, then t_1 must be aborted as well since it is a descendant transaction. In this case, t_2 must be aborted as well since it has seen uncommitted state of t_1—a cascading abort.

2. If not $s_1/LCAT(m_1, m_2)$ *retDep* $s_2/LCAT(m_1, m_2)$ and s_2 is scheduled before $s_1/LCAT(m_1, m_2)$ has finished execution then serializability between $s_1/LCAT(m_1, m_2)$ and $s_2/LCAT(m_1, m_2)$ may be violated. For example, m_2 may override a variable that has been written by m_1 and subsequently, another message m_1' that belongs to $s_1/LCAT(m_1, m_2)$ may read this variable.

4.5 Implementation of the Scheduling Mechanism

This section presents the design for an efficient implementation of the scheduling mechanism. All objects and methods described in this section are implemented as part of the Hermes/ST transaction handler. However, only objects and methods that are relevant to scheduling are described here. For other aspects of the Hermes/ST transaction handler refer to [FHR93c].

Since remote messages are much more expensive than local messages (see Section 5.7), this design minimizes network communications that are needed for scheduling. This is achieved via *lazy information propagation* and *caching* techniques.

Section 4.5.1 presents some important classes of this design. Section 4.5.2 shows how objects of these classes interact.

4.5.1 System Objects for Scheduling

4.5.1.1 Transactions

Transaction handlers are modelled as Hermes/ST objects of class `Transaction`. There is exactly one `Transaction` object for each transaction created in the execution of a system. This `Transaction` resides on one node of the network and coordinates the possibly distributed transaction. `Transactions` have the following variables[27].

`path` represents an identifier for the transaction and its position in the transaction tree, including references to the parent transaction and top-level transaction (if the transaction is not top-level itself).

`status` indicates the status of a transaction at a particular point in time, represented by the symbols `#executing` and `#committed`[28].

`threads` is a dictionary that includes the partial thread of the transaction creating message and all threads that belong to the transaction[29]. The keys of this dictionary are thread identifiers and the values are the status symbols `#executing` and `#finished`.

`subtransactions` is a set of references to `Transactions` if the transaction has any subtransactions.

4.5.1.2 TransactionCaches

There is exactly one `Transaction` object per transaction in the execution of a system. However, there may be many `TransactionCache` objects for one transaction, but at most one per node. As their name suggests, `TransactionCaches` cache information of a `Transaction`—information that is needed to determine the schedulability of messages. `TransactionCaches` have a subset of variables of `Transactions`.

`status` contains information about the transaction's status represented by the symbols `#?`, `#executing` and `#committed`.

`threads` is a dictionary with thread identifiers as keys and status symbols (`#?`, `#executing` and `#finished`) as values.

Additionally, `status` contains a set `objectsToInform` of local Hermes/ST objects that requested to be informed about the commit of the transaction. This is the case if the schedulability of messages that are sent to these objects depend on this transaction to have committed (Conditions 3(b)iii and iv of the schedulability predicate).

Status symbol `#?` represents the lack of information. This information must first be obtained from the `Transaction` which the `TransactionCache` represents. Status symbol `#executing` indicates that the `Transaction` has been asked about its `status` and the reply was `#executing`. It also indicates that the `Transaction` will inform the

[27] `Transactions` in Hermes/ST have many more variables. However, only the variables relevant for scheduling are described here.

[28] Again, there are more states which are not discussed here.

[29] Threads that belong to descendant transactions are not included here but are stored in the descendant `Transactions`.

`TransactionCache` whenever its status changes from `#executing` to `#committed`. Status symbol `#committed` indicates that the status of the `Transaction` is `#committed`. This is known either via direct inquiry or via informing by the `Transaction`.

The meaning of the status symbols for threads is analogous. Also, the `TransactionCache` keeps sets `objectsToInform` for `status` and each `thread` entry. `objectsToInform` contains a set of Hermes/ST objects whose `ConcurrencyControllers` requested to be informed about the commit of the transaction or the finish of execution of a thread.

Analogously, the `Transaction` keeps sets `nodesToInform` for `status` and each `thread` entry. `nodesToInform` contains a set of nodes that requested to be informed about the commit of the transaction or the finish of execution of a thread.

4.5.1.3 ConcurrencyControllers

Hermes/ST objects can have concurrency controllers that schedule incoming messages according to the schedulability predicate. A concurrency controller is an instance of class `ConcurrencyController`. It has two variables.

`pending` is a queue[30] that contains messages which are not schedulable at a particular point in time and are waiting to become schedulable.

`granted` is a set of schedulable messages that have started execution[31].

4.5.1.4 Messages, MessagePaths and MessagePathElements

Messages are represented by `Message` objects which encapsulate the following variables.

`messagePath` is a `MessagePath` object, a structure that identifies a message and indicates its position in a message tree.

`receiver` refers to the receiver object of the message.

`methodName` is a symbol that represents the name of the method to be invoked.

`arguments` is a list of method arguments. The length of this list must match the number of arguments requested by method `methodName`.

`lock` is the lock specification of the message. This can be a `ProgrammableLock` object (see Section 3.4.2).

A `MessagePath` is a list of `MessagePathElements`. A `MessagePathElement` has three variables.

`identifier` is a symbol or a number that identifies a message.

`kind` is a symbol that describes the kind of a message, `#synchronous` or `#asynchronous`.

`transactionCharacteristics` is a symbol that describes the transaction characteristics of a message: either `#transactionCreating` or `#nonTransactionCreating`.

[30] The `pending` queue differs from the queue data type in that elements can be de-queued from any position of the queue, not only from the head.

[31] They might even have finished execution.

`MessagePathElements` for top-level messages have system-wide unique `identifiers`, e.g. the IP address of the node on which the message is sent, concatenated with a node-wide unique number. All children are identified uniquely, e.g. via following numbers. The same applies for their children and so on. Thus, every `MessagePath` is system-wide unique.

Whenever an asynchronous message or a transaction creating message or a top-level message is sent, a new `MessagePath` is created. Nested synchronous, non-transaction creating messages are identified by their parent's `MessagePath`[32].

From a `MessagePath`, one can determine whether a message is transactional or non-transactional. Furthermore, one can deduce the `MessagePaths` of the thread, transaction and top-level transaction, a message belongs to.

4.5.2 Interaction of System Objects

To demonstrate the interactions of the system objects for scheduling, each possible scenario of sending and executing a message is examined in detail. See Figure 4.20. Consider that a `Message m` with `receiver fred` and `methodName print` is sent on a node called `#harpo`. `m` can be either sent from a client, e.g. a graphical user interface or from another message whose receiver object resides on `#harpo`. In the first case, `m` is top-level. In the second case, `m` is nested. If `m` is top-level, transaction creating or asynchronous, then a new `MessagePath mp` is created. Otherwise, `m` "inherits" its senders `MessagePath`.

4.5.2.1 Transaction Creation

Consider the case that `m` is transaction creating. Then, a new `Transaction` is created. In case that `m` is top-level or its sender is non-transactional, a new `TopLevelTransaction`[33] is created via `newTopLevelTransaction:m`. Otherwise, the message `newSubtransaction: m` is sent to the parent `Transaction`[34]. The parent `Transaction` then makes a new `Subtransaction` and includes it in its set of `subtransactions`. The new `Transaction` has its variables initialized to the following values.

`path` is initialized with `mp`.

`status` is initialized with `#executing`.

`threads` : is initialized with a `Dictionary` that contains one entry. The `key` of this entry is `mp`[35] and the value is `#executing`.

`subtransactions` is initialized with an empty `Set`.

4.5.2.2 Thread Creation

Consider the case that `m` is asynchronous. If `m` is also transactional, but not transaction creating then the message `executionStarted:m` is sent to its `Transaction`, i.e. the object that represents the transaction that `m` belongs to. It includes `mp` into its dictionary `threads` with status `#executing`.

[32] Section 4.6.2.1 shows why the schedulability predicate can be implemented correctly although *synchnonTrans* messages are not assigned a new `MessagePath`.

[33] Class `Transaction` has two subclasses, `TopLevelTransaction` and `Subtransaction`.

[34] Note that the parent transaction's `MessagePath` can be generated from `m`'s `MessagePath mp`. A reference to the parent `Transaction` can be created from its `MessagePath` so that messages can be sent to the `Transaction`.

[35] In this case, `mp` represents the partial thread that is created by `m`.

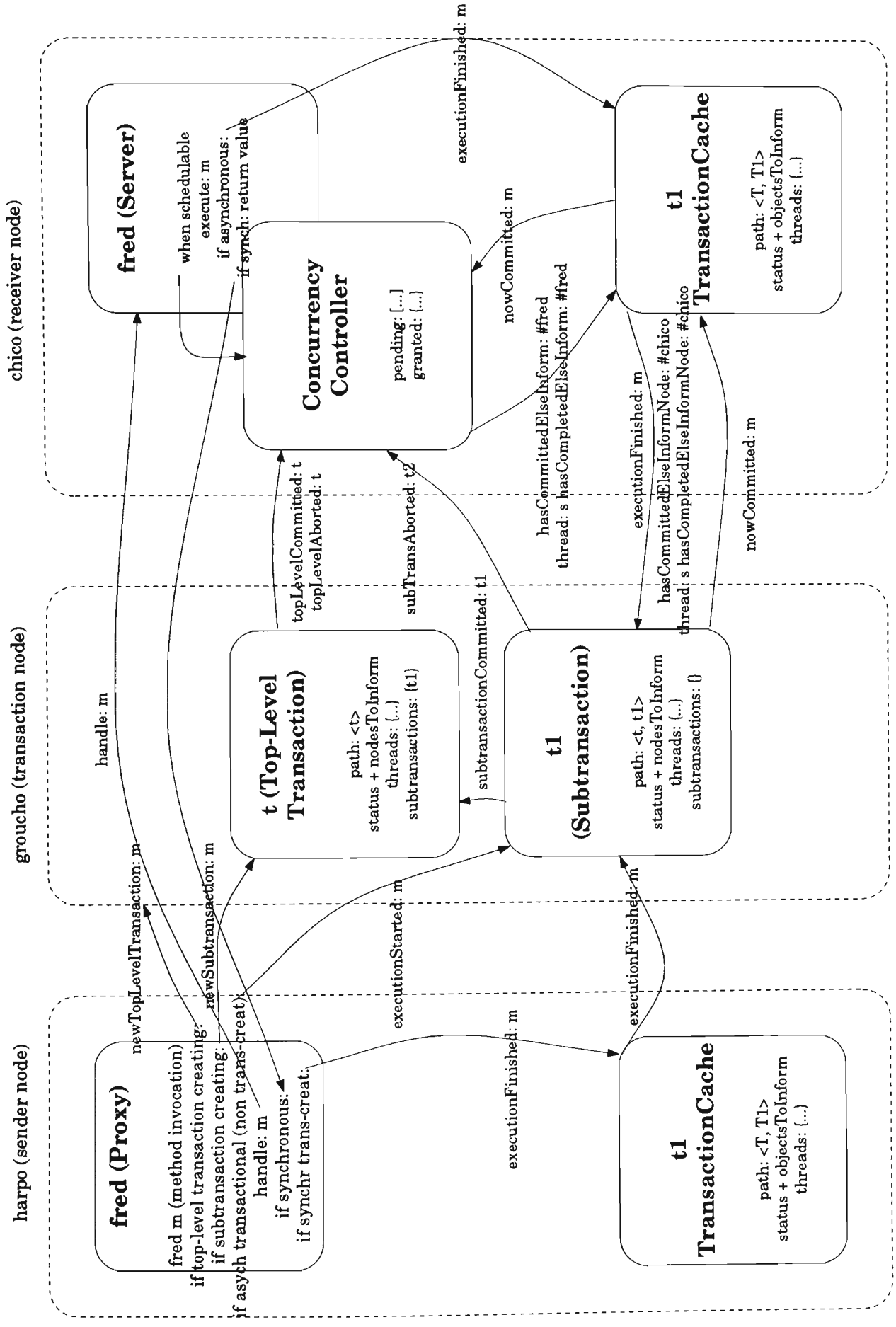

Figure 4.20: Scenario of sending and executing a message.

For transaction creating messages, mp is included into threads at initialization of the new Transaction, as explained above. Thus, executionStarted:m need not be sent in this case.

The sender of the asynchronous message is now allowed continue its execution. Note that it is important that asynchronous transactional messages make a Transaction or register their thread identifier with their Transaction *before* their sender is allowed to continue its execution. This is necessary for the following two reasons.

1. Consider the case of a subtransaction creating an asynchronous message. If the sender is allowed to continue before the creation of the subtransaction is known to the parent Transaction then the the following race condition can happen. The parent Transaction can then commit before the subtransaction has committed, even before it has started execution.

2. Consider the case of a transactional, asynchronous, but non-transaction creating, message. If the sender is allowed to continue before the new thread is registered with its Transaction then the following race condition can happen. The Transaction can commit before the thread has finished execution, even before it has started execution.

4.5.2.3 Sending the Message

In case m is local, i.e., fred resides on #harpo, m is handed to fred's Concurrency-Controller for schedulability testing. In case m is remote, e.g., fred resides on #chico, a Proxy object for fred on #harpo handles the remote invocation transparently[36].

fred's Proxy hands m to a CommunicationsHandler. The CommunicationsHandler marshals m[37] and sends it to #chico where it is unmarshaled.

4.5.2.4 Concurrency Control

Once m arrives at fred, it is passed on to fred's ConcurrencyController to check for schedulability. Schedulability testing is performed by comparing the incoming message m against all messages in granted. m is schedulable if for all messages m2 in granted that are conflicting (i.e., whose locks are incompatible with m's lock), message m isSchedulableWithRespectTo:m2 returns true. Section 4.6 describes the implementation of isSchedulableWithRespectTo: in detail.

To determine the schedulability of m with respect to a conflicting granted message, the MessagePaths of the two messages are compared. Two types of information may have to be obtained remotely: information about the commit of a transaction and the finish of execution of a thread. To find out whether a transaction t1 has committed, t1's local TransactionCache on #chico is sent the message hasCommittedElseInform: #fred. If such a local TransactionCache does not yet exist then it is now created. The TransactionCache has three options to respond to this request.

1. status = #?. In this case, the TransactionCache has no information about the commit status of the transaction it represents and has not yet attempted to obtain any information. It then sends the message hasCommittedElseInform:#chico to

[36] In fact, fred or fred's Proxy also initiate the transaction creation and thread creation as described above.

[37] Marshalling refers to the transformation of an object into a representation which can be sent over the network, typically a byte stream.

the Transaction, which resides on #groucho. If the transaction actually has committed, i.e., its status is #committed, then the Transaction returns true to the TransactionCache. The TransactionCache then sets its status to #committed and returns true to fred's ConcurrencyController.

Otherwise, i.e., if the Transaction's status is not #committed, then the Transaction inserts #chico into its set nodesToInform for status and returns false to the TransactionCache. The TransactionCache then inserts fred into its set objectsToInform for status and returns false to fred's ConcurrencyController.

2. status = #executing. This indicates that the Transaction has already been asked whether it has committed and false was returned. It also implies that the Transaction has included #chico into its set of nodesToInform for status. Therefore, a further access to the Transaction is not necessary. Instead, fred is added to the TransactionCache's set objectsToInform for status and false can be returned to fred's ConcurrencyController immediately.

3. status = #committed. This indicates that the Transaction has been asked whether it has committed and true was returned. In this case, no further access to the Transaction is necessary and true can be returned immediately to fred's ConcurrencyController.

When t1 finally commits, it sends the message nowCommitted:t1 to the TransactionCaches on all nodes specified in nodesToInform. These TransactionCaches then set their status variable to #committed. In turn, they send the message nowCommitted:t1 to all objects specified in objectsToInform.

Requests about the finish of execution of a particular thread are processed by the TransactionCache of the Transaction that the thread belongs to, in an analogous way, via messages hasThread:s finishedElseInform:fred, hasThread:s finishedElseInform:#chico, and threadNowFinished:s.

4.5.2.5 Scheduling

If m is not schedulable then it is enqueued in pending and possibly re-tested for schedulability at a later time. Otherwise, i.e., if m is schedulable, then m is added to granted and its execution is started. After m has finished execution, the following operations are performed.

If m is non-transactional then it is removed from granted and the result is returned to the sender (in case m is synchronous). The removal of a non-transactional message from granted after it has finished execution is compatible with the schedulability predicate. Note that in the schedulability predicate, a finished non-transactional message m_1 never causes a message m_2 not to be schedulable.

If m is transactional, then two cases are distinguished.

1. m is asynchronous;

2. m is synchronous.

m is asynchronous: In this case, the message executionFinished:m is sent to its Transaction via its local TransactionCache. Both the TransactionCache and Transaction objects change the status for thread m from #executing to #finished.

m is synchronous: In this case, the result is returned to the sender. If m additionally is transaction creating, then the message executionFinished:m is sent to the Transaction after the value has been returned.

The Transaction starts the prepare phase of the 2PC protocol when the following two conditions are satisfied.

1. All subtransactions have committed.

2. All threads, including the partial thread that created the transaction, have finished.

For this reason, it is important that for synchronous, transaction creating messages, the result is returned first before executionFinished:m is sent. Otherwise the following race condition could happen. The Transaction could commit without the result of the message actually being delivered[38].

When a Transaction t commits or aborts then all objects that belong to t and all its descendant transactions are informed about this event via the messages topLevelTransactionCommit:t, topLevelTransactionAbort:t or subtransactionAbort:t. Apart from recovery related activity, these messages provide scheduling information for the visited object's ConcurrencyControllers. All messages that belong to t or any of its descendant transactions are removed from both pending and granted.

The removal of these messages from granted is compatible with the schedulability predicate. Note that in the schedulability predicate, a message m_1 that belongs to a committed or aborted top-level transaction or that belongs to an aborted subtransaction never causes a message m_2 not to be schedulable.

4.5.2.6 Rescheduling Pending Requests

There are four situations in which messages are removed from pending.

1. A top-level transaction commits.

2. A top-level transaction aborts.

3. A subtransaction aborts.

4. A non-transactional message finishes execution.

Furthermore, there are two events that Transactions inform ConcurrencyControllers about via TransactionCaches.

1. A transaction commits.

2. A (partial) thread finishes execution.

All six events may have an impact on the schedulability of messages in pending. Therefore, they all trigger re-testing of messages in pending for schedulability. This test is performed from the head to the tail of the queue pending so that messages that have been waiting the longest are tested first.

[38] Another approach is to notify the Transaction about the two events independently, namely the finish of execution of the partial thread that created the Transaction and the delivery of the result. However, this approach requires an additional network communication and is therefore not preferable.

4.5.2.7 Broadcasting versus Asking

The lazy information propagation and caching techniques presented in this section have the potential for large savings in network communications for obtaining scheduling information remotely. Scheduling information is information about the commit of subtransactions and the finish of execution of transactional (partial) threads. Note that if scheduling information is needed by a `ConcurrencyController` on a particular node then this information is obtained exactly once. If scheduling information is not needed by any `ConcurrencyController` on a particular node, then it is not obtained at all.

To obtain scheduling information, either one or two messages need to be sent. Only one message is needed if the awaited event[39] has already happened. If this is not the case then two messages are needed. The first message gets the negative reply and ensures that `ConcurrencyControllers` are informed after the event has happened. The second message informs `ConcurrencyControllers` that the event has happened.

The alternative to an asking mechanism is a broadcast mechanism. Whenever such an event happens then scheduling information is broadcast to all nodes that are potentially interested in it. In the scheduling context, broadcasting is not a viable alternative to asking. This is because it is very hard to determine the group of nodes that are potentially interested in scheduling information.

Take the event that a subtransaction t has committed. This information might be needed to determine the schedulability of a message m_2 that belongs to the same top-level transaction than t. This is the case if $t = 1LBLCAT(m_1, m_2)$ where m_1 is a message that belongs to t or any of its descendant transactions. In order to ensure that the `ConcurrencyController` that schedules m_2 obtains the information about t's commit locally, t must broadcast this information at least to all nodes of all objects that have been visited by t's top-level transaction and any of its descendant transactions—a potentially large number of nodes. However, it still does not cover the set of all nodes that are potentially interested in this scheduling information. This is because t's top-level transaction is still executing and more nodes can be visited after the commit event. In short, in order to ensure that all nodes are informed that potentially need this scheduling information, this information must be broadcast to all nodes in the entire network. Similar arguments hold for the scheduling information about the finish of execution of a transactional (partial) thread.

Thus, broadcasting scheduling information is not a workable approach if the distributed system contains a large number of nodes. This is even more so considering the fact that, from the experience of real-world applications like the distributed bank, obtaining scheduling information remotely is rarely necessary.

4.6 Implementation of the Schedulability Predicate

This section describes algorithms for the schedulability predicate that are both efficient and easy to implement. Two algorithms `schedulable` and `returnDependent` are presented in pseudo code. `schedulable` implements the predicate "schedulable with respect to" and `returnDependent` implements the return dependency predicate.

[39] The transaction has committed or the transactional (partial) thread has finished execution.

4.6.1 The Algorithms

4.6.1.1 Data Structures

The main data structure used in schedulable and returnDependent is the MessagePath, an array of MessagePathElements. An individual MessagePathElement e of a MessagePath m is accessed by e := m[i] where i is an Index running from 1..depth(m). MessagePathElements can be compared for equality (=), it can be checked whether they are synchronous (synch) or asynchronous (asynch), transaction creating (trans) or non-transaction creating (nonTrans).

4.6.1.2 schedulable

```
01 schedulable(m1, m2: MessagePath)
02{
03  LCAT := 0;    (* index of LCAT(m1, m2) *)
04  LCA := 0;     (* index of LCA(m1, m2) *)
05  1LBLCAT := 0;(* index of 1LBLCAT(m1, m2) *)
06  LCAS := 1;    (* index of least common thread of m1 and m2 *)
07  S := 0;       (* index of last thread of m1 which is not shared by m2 *)

08  for i := 1 to depth(m1) do
    (* 1st phase: descend common subpath between m1 and m2 *)
09    if m1[i] = m2[i] then  (* elements are the same *)
10    {
11      LCA := i;
12      if trans(m1[i]) then LCAT := i;
13      if asynch(m1[i]) then LCAS := i;
14    } else  (* elements are different *)
15    {
16      if LCAT = 0 then
        (* either m1, m2 not both transactional or not tl1 = tl2 *)
17        return false;
18      else  (* m1, m2 both transactional with t1 = t2 *)
19      {
20        for j := i to depth(m1) do
          (* 2nd phase: descend subpath of m1 not shared by m2 *)
21        {
22          if trans(m1[j]) and 1LBLCAT = 0 then 1LBLCAT := j;
23          if asynch(m1[j]) then S := j;
24        };
          (* 3rd phase: descend subpath of m2 which is not shared by m1 *)
25        if S = 0 and returnDependent(m2, LCA) then
          (* return dependency between s1/LCAT and m2 *)
26          return true
27        else (* no return dependency between s1/LCAT and m2 *)
28          return finishedExecution(max(LCAS, S, LCAT), m1)
                    and (1LBLCAT=0 or committed(1LBLCAT, m1))
29      } (* end else *)
30    } (* end else *)
31  }; (* end for loop -> m1 < m2 *)
```

```
    (* 3rd phase: descend subpath of m2 which is not shared by m1 *)
32  if returnDependent(m2, LCA) then
33    return true;  (* return dependency between m1 and m2 *)
34  else  (* no return dependency between  m1 and m2 *)
35    if LCAT = 0 then  (* m1 is non-transactional *)
36      return false;
37    else  (* tl1 = tl2 *)
38      return finishedExecution(max(LCAS,LCAT), m1)
39 }
```

committed(idx:Index, m:MessagePath) and finishedExecution(idx:Index, m: MessagePath) perform potential network communications to ask a Transaction whether it has committed or whether a (partial) thread has completed. Caching on the node level is performed as described in the previous section.

4.6.1.3 returnDependent

Conceptually, returnDependent has two MessagePaths m1 and m2 as arguments with m1 < m2. Because of the real ancestor relationship, it is enough to pass m2 and an index idx such that m1 = m2[1]...m2[idx].

```
returnDependent(m: MessagePath, idx: Index)
  {
01  for i := idx + 1 to depth(m) do
02  {
03    if synch-trans(m[i]) then return true;
04    if asynch(m[i]) then return false
05  }
06  return true
  }
```

4.6.2 Correctness of the Schedulability Algorithm

4.6.2.1 Long MessagePaths versus Short MessagePaths

A message path as defined in Section 4.1.3 includes message path elements for all messages from the root of a message tree down to the message that the path describes. The definition of the schedulability predicate and its correctness analysis are based on this definition of a message path. However, a MessagePath object, as described in Section 4.5.1 is shorter. It only contains MessagePathElements for transaction creating messages or asynchronous messages. The fact that MessagePaths are short is important for the efficiency of the scheduling algorithms, both in space and time. This section shows why it is enough to use short MessagePaths and still be able to implement the schedulability predicate correctly.

All rules of the schedulability predicate deal only with threads and transactions except the test for return dependency. Therefore, in this section, it is analyzed whether the return dependency test for *long message paths* (i.e. paths including *synch-nonTrans* elements) is equivalent to the return dependency test for respective *short paths* (i.e. paths with *synch-nonTrans* elements removed). Unfortunately, this is not the case. However, it can be shown that in the context of the schedulability predicate and its implementation, differences do not lead to wrong results in the schedulability test.

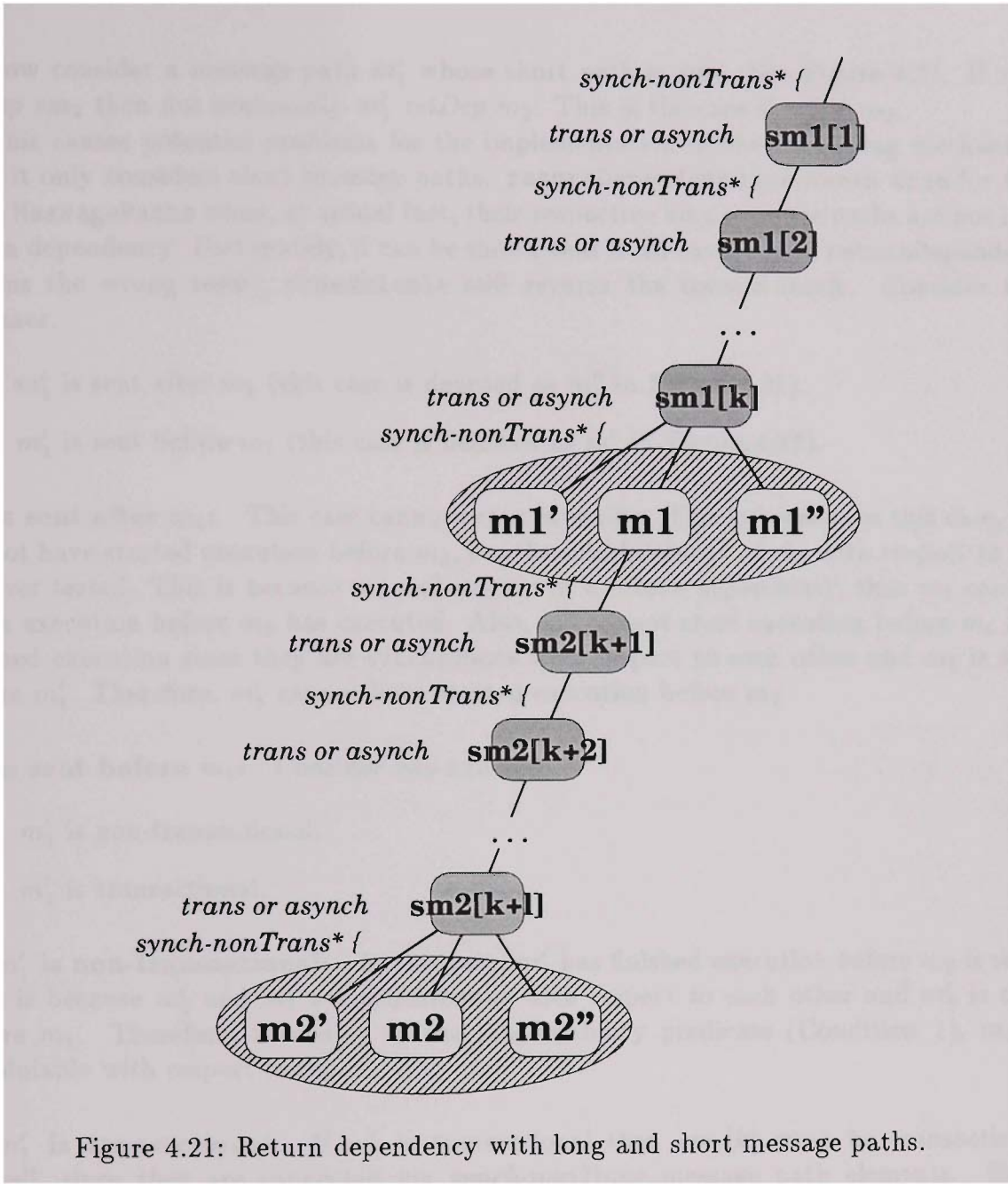

Figure 4.21: Return dependency with long and short message paths.

Consider two conflicting messages m_1 and m_2 where m_1 has started execution before m_2 is sent. Assume that m_1 *retDep* m_2. Let $sm_1 = sm_1[1]...sm_1[k]$ and $sm_2 = sm_1[1]...sm_1[k]sm_2[k+1]...sm_2[k+l]$ be the corresponding short message paths, i.e. the paths with all *synch-nonTrans* message path elements removed. See Figure 4.21. It is easy to see that since m_1 *retDep* m_2, also sm_1 *retDep* sm_2. This is because the relationship between return dependent message paths m_1 and m_2 is $m_2 = m_1$ *synch-nonTrans* [synchTrans any*]*. If *synch-nonTrans* messages are deleted that match the part *synch-nonTrans** of the regular expression then this does not cause the predicate not to be satisfied. The same is true if *synch-nonTrans* messages are deleted that match the part *any* of the regular expression.

Now, it is examined whether the opposite is true. Assume that sm_1 *retDep* sm_2. Is m_1' *retDep* m_2' for all long message paths m_1' and m_2' whose corresponding short paths are sm_1 and sm_2?

Consider a message path m_2' whose short path is sm_2. See Figure 4.21. If sm_1 *retDep* sm_2 then m_1 *retDep* m_2'. This is because of the transitivity of the return dependency relationship and the fact that $sm_2[k+l]$ is return dependent on m_2, since they are connected via *synch-nonTrans* messages.

Now consider a message path m_1' whose short path is sm_1. See Figure 4.21. If sm_1 *retDep* sm_2 then not necessarily m_1' *retDep* m_2. This is the case if $m_1' \not\leq m_2$.

This causes potential problems for the implementation of the scheduling mechanism since it only considers short message paths. `returnDependent` may return `true` for two short **MessagePaths** when, in actual fact, their respective long message paths are not in a return dependency. Fortunately, it can be shown that in all cases where `returnDependent` returns the wrong result, `schedulable` still returns the correct result. Consider two subcases.

1. m_1' is sent after m_1 (this case is denoted as m_1'' in Figure 4.21).
2. m_1' is sent before m_1 (this case is denoted as m_1' in Figure 4.21).

m_1' is sent after m_1: This case cannot occur in reality. This is because in this case, m_1' cannot have started execution before m_2, i.e. the schedulability of m_2 with respect to m_1' is never tested. This is because m_1 and m_2 are in a return dependency, thus m_1 cannot finish execution before m_2 has executed. Also, m_1' cannot start execution before m_1 has finished execution since they are synchronous with respect to each other and m_1 is sent before m_1'. Therefore, m_1' cannot have started execution before m_2

m_1' is sent before m_1: Consider two subcases.

1. m_1' is non-transactional.
2. m_1' is transactional.

m_1' is non-transactional: In this case, m_1' has finished execution before m_2 is sent. This is because m_1' and m_1 are synchronous with respect to each other and m_1' is sent before m_1. Therefore, according to the schedulability predicate (Condition 1), m_2 is schedulable with respect to m_1'.

m_1' is transactional: If m_1' is transactional then $sm_1[k]$ must be transactional as well, since they are connected via *synch-nonTrans* message path elements. Since $sm_1[k] \leq m_1 \leq m_2$, m_1 and m_2 are transactional as well. Let t_1' be the transaction of m_1', t_1 the transaction of m_1 and t_2 the transaction of m_2. Then, $t_1' = t_1 \leq t_2$. Thus, $LCAT(m_1', m_2) = t_1$. Let s_1' be the thread of m_1' and s_1 the thread of m_1. Then, $s_1' = s_1$ since m_1' and m_1 are synchronous with respect to each other. Recall that $sm_1[k]$ *retDep* m_2. Since s_1'/t_1 is synchronous with respect to $sm_1[k]$, also s_1'/t_1 *retDep* m_2. Therefore, m_2 is schedulable with respect to m_1' (Conditions 3(b)i and ii of the schedulability predicate).

4.6.2.2 The First Phase

In this section and the following sections, it is shown that for conflicting messages m_1 and m_2, where m_1 has started execution before m_2 is sent, `schedulable(m1, m2)` returns `true` exactly if m_1 is schedulable with respect to m_2.

`schedulable` consists of three main phases. In the first phase (8-14)[40] m_1 and m_2 are descended on their common subpath if such a common subpath exists. See Figure 4.22. The first phase finishes when the first element $m_1[i]$ is found which is not equal to $m_2[i]$ (14). During this loop, `LCA` is set to the loop index `i` (11). Therefore, after the finish of

[40] Numbers in brackets refer to line numbers in the code for `schedulable` and `returnDependent`.

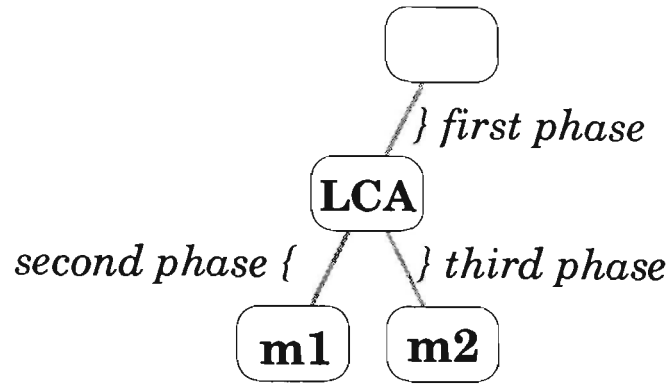

Figure 4.22: Phases of the schedulability algorithm.

the first phase, LCA can have two kinds of values. If m_1 and m_2 belong to different message trees then LCA is 0 (the initial value). Otherwise, LCA is the index of the last element that is common to m_1 and m_2, hence $LCA(m_1, m_2)$. Analogously, LCAT is assinged an index when a common transaction creating message is detected (12) and otherwise is still 0. LCAS is assigned an index if a common asynchronous message is detected (13) and otherwise is still 1[41].

When the first element is detected where m_1 and m_2 are different then the first phase finishes (14). If no transaction creating request has been detected during the first phase (LCAT = 0) then false is returned. This is compatible with the schedulability predicate for the following reasons.

The fact that there is no least common ancestor transaction between m_1 and m_2 indicates that either not both messages are transactional or their top-level transactions tl_1 and tl_2 are different. Since m_1 has not been removed from set granted of the ConcurrencyController, none of the following conditions have happened yet.

- m_1 is non-transactional and has finished execution.
- m_1 is transactional and its transaction has aborted.
- m_1 is transactional and its top-level transaction has committed.

Conditions 1, 2 and 3a of the schedulability predicate are not satisfied in this case. This is, provided, that m_1 and m_2 are neither synchronous nor in a return dependency. Both cases can be ruled out as is shown below.

If m_1 and m_2 were synchronous then m_1 would have been removed from granted at the latest when its ancestor returned to $LCA(m_1, m_2)$. This is true no matter whether m_1 is transactional or not. If m_1 is non-transactional then it has finished before its ancestor has returned to $LCA(m_1, m_2)$. Non-transactional messages are removed from granted after they have finished execution. If m_1 is transactional, then its top-level transaction is a descendant of $LCA(m_1, m_2)$. Otherwise, $LCA(m_1, m_2)$ would have been transactional, too. This top-level transaction has committed or aborted before its ancestor has returned to $LCA(m_1, m_2)$. Thus, m_1 has been removed from granted in this case.

m_1 and m_2 cannot be in a return dependency in this case, since $m_1 \not\leq m_2$. This is because there is at least one message path element in m_1 which is not shared by m_2.

[41] LCAS is initialized to 1 for reasons outlined below.

4.6.2.3 The Second Phase

If there is a common transaction between m_1 and m_2 then the second phase of the algorithm starts. In the second phase (20-24), the subpath of m_1 which is not shared by m_2 is descended, if such a subpath exists. See Figure 4.22.

While descending this subpath, the index of the first transaction creating message is stored in `1LBLCAT`. The criterion for assigning the index `j` to variable `1LBLCAT` is that $m_1[j]$ is transaction creating and `1LBLCAT` has not been assigned a value yet except the initial value `0`. If `1LBLCAT` is assigned a value then it is the index of $1LBLCAT(m_1, m_2)$. This is because it is the subtransaction of $LCAT(m_1, m_2)$ and it is an ancestor of m_1.

Furthermore, the index of the last asynchronous message in this subpath is stored in variable `S`. This is performed by assigning the index `j` to `S` whenever $m_1[j]$ is asynchronous.

4.6.2.4 The Third Phase

The third phase of `schedulable` is performed by function `returnDependent` (25). It descends the subpath of m_2 which is not shared by m_1 to detect a return dependency between $LCA(m_1, m_2)$ and m_2.

`returnDependent` has two arguments, a `MessagePath m` and an `Index idx`. It returns whether $m[1]...m[idx]$ *retDep* $m[1]...m[idx]m[idx+1]...m[depth(m)]$. It is easy to see that `returnDependent` returns the correct result.

`m` is descended from `idx` (excluding) to its last element (including) to check whether this subpath matches the regular expression *synch-nonTrans* [synch-trans any*]*. If a *synch-trans* element is detected (3) then `true` is returned since any message type is allowed to follow. If an *asynch* element is detected (4) then `false` is returned since the regular expression is not matched. If a *synch-nonTrans* element is detected then descending continues (implicit). When the for loop finishes then this indicates that all messages in the subpath are *synch-nonTrans*. `true` is returned in this case (6).

4.6.2.5 `S = 0 and returnDependent(m2, LCA)`

In Line 25 of function `schedulable`, is tested whether `S = 0 and returnDependent(m2, LCA)`. This condition is equivalent to $s_1/LCAT(m_1, m_2)$ *retDep* m_2. This is shown in two parts.

1. If `S = 0 and returnDependent(m2, LCA)` then $s_1/LCAT(m_1, m_2)$ *retDep* m_2.
2. If not (`S = 0 and returnDependent(m2, LCA)` then not $s_1/LCAT(m_1, m_2)$ *retDep* m_2

`S = 0 and returnDependent(m2, LCA)`: `S = 0` indicates that there is no asynchronous message in the subpath of m_1 which is not shared by m_2. Thus, $s_1/LCAT(m_1, m_2) \leq LCA(m_1, m_2)$. Note that $s_1/LCAT(m_1, m_2)$ *retDep* $LCA(m_1, m_2)$. This is because there are only *synch-nonTrans* messages between $s_1/LCAT(m_1, m_2)$ and $LCA(m_1, m_2)$. Since `returnDependent(m2, LCA)` returns true, $LCA(m_1, m_2)$ *retDep* m_2. With transitivity of the return dependency relationship it follows that $s_1/LCAT(m_1, m_2)$ *retDep* m_2.

not (`S = 0 and returnDependent(m2, LCA)`: Consider two subcases.

1. not `S = 0`.
2. `S = 0` but `returnDependent(m2, LCA)` returns `false`.

not `S = 0`: The fact that `S` has been assigned an index in the third phase of **schedulable** indicates that there is an asynchronous message in the subpath of m_1 which is not shared by m_2. Therefore, $s_1 > LCA(m_1, m_2)$ and therefore, $s_1/LCAT(m_1, m_2) = s_1$. Also, $s_1 \not\leq m_2$. Therefore, not $s_1/LCAT(m_1, m_2)$ *retDep* m_2.

`S = 0` **but** `returnDependent(m2, LCA)` **returns** `false`: Examine messages in the subpath between $s_1/LCAT(m_1, m_2)$ and $LCA(m_1, m_2)$. There cannot be a transaction creating message in this subpath, otherwise this message would be $LCAT(m_1, m_2)$. Also, there cannot be an asynchronous message in this subpath, otherwise this message would be s_1. Hence, all messages between $s_1/LCAT(m_1, m_2)$ and $LCA(m_1, m_2)$ are *synch-nonTrans*. Therefore, testing for return dependency can start from $LCA(m_1, m_2)$, since an arbitrary number of *synch-nonTrans* message path elements can be ignored by the return dependency test. Since `returnDependent(m2, LCA)` returns `false` in this case, not $s_1/LCAT(m_1, m_2)$ *retDep* m_2.

If there is a return dependency between $s_1/LCAT(m_1, m_2)$ and m_2 then `true` is returned (26). This is compatible with the schedulability predicate. Consider two subcases.

1. `1LBLCAT = 0`.
2. not `1LBLCAT = 0`.

`1LBLCAT = 0`: Since there is no transaction creating message in the subpath of m_1 which is not shared by m_2, $t_1 \leq t_2$. Since $s_1/LCAT(m_1, m_2)$ *retDep* m_2, Conditions 3(b)i and ii of the schedulability predicate are satisfied in this case[42].

not `1LBLCAT = 0`: In this case, t_1 may be incomparable with t_2. Condition 3(b)iv requires that in addition to $s_1/LCAT(m_1, m_2)$ *retDep* m_2, $1LBLCAT(m_1, m_2)$ must have committed. This can be ensured for the following reasons. There is no asynchronous message in subpath of m_1 which is not shared by m_2, since `S = 0`. Also, the child of $LCA(m_1, m_2)$ which is an ancestor of m_1 has been invoked before the child of $LCA(m_1, m_2)$ which is an ancestor of m_2. This is because otherwise m_1 could not have started execution before m_2 was sent[43]. Since synchronous transaction creating messages return only after the transaction has committed[44], $1LBLCAT(m_1, m_2)$ must have committed.

4.6.2.6 Not [`S = 0` and `returnDependent(`m_2`, LCA)`]

Now consider the case that not $s1/LCAT(m_1, m_2)$ *retDep* m_2 (27). In this case, `finished-Execution(max(LCAS, S, LCAT), m1)` and `(1LBLCAT=0 or committed(1LBLCAT, m1))` is returned. The following two observations can be made. First, m_1 cannot be in return dependency with m_2 since the two paths have different elements and hence $m_1 \not< m_2$. Second, m_1 cannot be synchronous with m_2 since then $s_1/LCAT(m_1, m_2)$ would be in a return dependency with m_2. Consider two subcases.

1. `1LBLCAT = 0`.
2. not `1LBLCAT = 0`.

[42] Note that in Condition 3(b)i, the disjunction s_1/t_1 *retDep* m_2 has been omitted since it can not occur in this particular case.

[43] Note that $LCA(m_1, m_2)$ *retDep* m_2.

[44] Note that if $1LBLCAT(m_1, m_2)$ aborts then m_1 is removed from granted.

1LBLCAT = 0: In this case there is no subtransaction creating message in the subpath of m_1 which is not shared by m_2. Then, $t_1 = LCAT(t_1, t_2)$. In this case, t_1 and t_2 can be in either of the two relationships.

1. $t_1 = t_2$. This is the case if there is also no transaction creating message in the subpath of m_2 which is not shared by m_1.

2. $t_1 < t_2$. This is the case if there is a transaction creating message in the subpath of m_2 which is not shared by m_1.

Under the assumption that not $s_1/LCAT(m_1, m_2)$ *retDep* m_2, the schedulability predicate for both cases is the same (Conditions 3(b)i and ii): m_2 is schedulable with respect to m_1 if the execution of s_1/t_1 (hence $s_1/LCAT(m_1, m_2)$) has finished.

If `S = 0` then there is no asynchronous message in the subpath of m_1 which is not shared by m_2. Then, `LCAS` is the the index of s_1. Note that `LCAS` is initialized to `1`. This reflects the fact that the thread of a message is the top-level message if there is no asynchronous element in its message path. If not `S = 0` then `S` is the index of the last asynchronous message in the subpath of of m_1 which is not shared by m_2. Hence `S` is the index of s_1 in this case.

Note that if not `S = 0` then its value is larger than `LCAS`. This is because it represents the index of an element further down the message path of m_1. Therefore, `max(LCAS, S)`, which computes the maximum of both indices, is the index of s_1.

Recall the definition of s/t. If $s \leq t$ then $s/t = t$ otherwise $s/t = s$. Therefore, the index of s/t can be determined by the maximum of the index of s and the index of t.

Therefore, the index of $s_1/LCAT(m_1, m_2)$ is `max(max(LCAS, S), LCAT) = max (LCAS, S, LCAT)`. `schedulable` returns `finishedExecuting(max(s1, s3, LCAT)` in this case since `1LBLCAT = 0` (28).

not 1LBLCAT = 0: If not `1LBLCAT = 0` then there is a transaction creating message in the subpath of m_1 which is not shared by m_2. In this case, t_1 and t_2 can be in either of the two relationships.

1. $t_1 > t_2$. This is the case if there is no transaction creating message in the subpath of m_2 which is not shared by m_1.

2. $t1 <> t2$. This is the case if there is a transaction creating message in the subpath of m_2 which is not shared by m_1.

Considering that $s_1/LCAT(m_1, m_2)$ and m_2 are in no return dependency, the Properties 3(b)iii and iv of the schedulability predicate are satisfied if the execution of $s_1/LCAT(m_1, m_2)$ is finished and $1LBLCAT(m_1, m_2)$ has committed.

As argued above, `finishedExecution(max(LCAS, S, LCAT), m1)` returns `true` if $s_1/LCAT(m_1, m_2)$ has finished execution. Since 1LBLCAT is the index of $1LBLCAT(m_1, m_2)$, `committed(1LBCAT, m1)` returns `true` if $1LBLCAT(m_1, m_2)$ has committed.

4.6.2.7 $m_1 < m_2$

The `for` loop of Lines 8–31 can only terminate without being pre-empted by a return statement if the `else` statement in Line 14 is never reached. This is the case if all elements of m_1 are shared by m_2 and hence $m_1 < m_2$[45]. In particular, $LCA(m_1, m_2)$ is

[45] Per definition, $m_1 \neq m_2$.

the last element of m_1. In this case, the third phase of the algorithm is started by invoking the `returnDependent` function (32). In the third phase, the subpath of m_2 which is not shared by m_1 is descended to check for return dependency between m_1 and m_2. If such a return dependency is detected then `true` is returned (33), according to the schedulability predicate.

Note that if such a dependency cannot be detected then m_1 and m_2 cannot be synchronous with respect to each other since $m_1 < m_2$. Two subcases are distinguished.

1. `LCAT = 0;`
2. not `LCAT = 0`.

`LCAT = 0`: This indicates that m_1 is non-transactional. In this case, Condition 1 of the schedulability predicate requires the execution of m_1 to be finished. Since m_1 is not yet removed from `granted`, it can be deduced that this is not yet the case. Therefore, `false` is returned in this case (36).

not `LCAT = 0`: This indicates that m_1 is transactional. In this case, either $t_1 = t_2$ or $t_1 < t_2$. This is because $m_1 < m_2$. Considering that there is no return dependency between $s_1/LCAT(m_1, m_2)$ and m_2, Properties 3(b)i and ii of the schedulability predicate are identical. m_2 is schedulable with respect to m_1 if the execution of s_1/t_1 has finished. Since `LCAT` is the index of t_1, `executionFinished(max(LCAS, LCAT), m1)` returns `true` in this case (38).

4.6.2.8 Termination and Complexity

It is easy to see that `schedulable` returns a boolean result for every pair of message paths m_1 and m_2. `schedulable` has three distinct phases that are performed at most once. Phase one descends the subpath of m_1 that is shared by m_2 if such a subpath exists. Phase two descends the subpath of m_1 which is not shared by m_2 if such a subpath exists. Phase three descends the subpath of m_2 which is not shared by m_1 if such a subpath exists. At the end of phases one and two, either a result is returned or another phase is started. At the end of phase three, a result is returned.

The analysis above has shown that whenever the algorithm returns a value it is compatible with the schedulability predicate. Since the algorithm returns a value for all pairs of message paths m_1 and m_2, the algorithm is correct with respect to the schedulability predicate.

It is easy to see that the algorithm is linear in the sum of the depth of m_1 and m_2. This is because the first phase is linear in the minimum of the depth of m_1 and m_2. The second phase is linear in the length of m_1. The third phase is linear in the depth of m_2. Furthermore, phases one, two and three are performed at most once. More precisely, the schedulable algorithm descends the `MessagePaths` of m_1 and m_2 at most once.

4.6.2.9 Additional Optimizations

Section 4.5.1 describes a generator for `NamePathElement` identifiers. Top-level messages are assigned a network-wide unique identifier, e.g., composed of the IP address of the node where it is sent and a node-wide unique number. Its children are named by successive integer number, in the order in which they are sent. The same strategy is used for their children and so on.

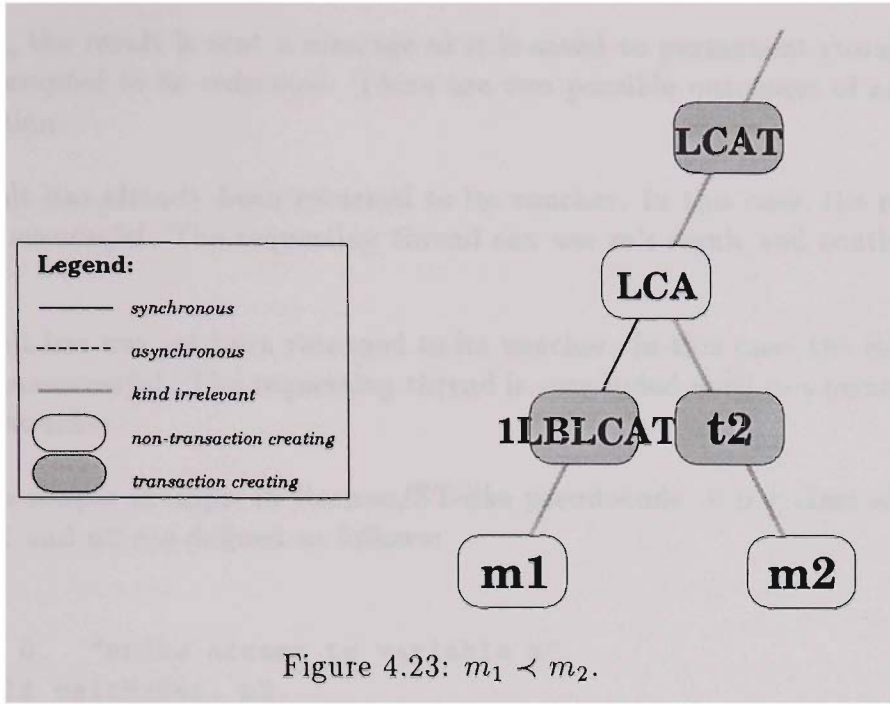

Figure 4.23: $m_1 \prec m_2$.

For **MessagePaths** constructed in such a way, a lexicographic order $\prec$[46] defines a total order over all messages belonging to a particular message tree. Note that for all messages m_1 and m_2 that are synchronous with respect to each other and $m_1 \prec m_2$, m_2 is sent after m_1 has finished execution. Particularly, if m_1 is transaction creating then m_2 is sent after m_1 has committed.

The ordering of messages in a message tree is a piece of information that neither the schedulability predicate nor its implementation utilize as described so far. Ordering information can be used by the schedulability algorithm to further reduce network communications for obtaining scheduling information.

Consider the example in Figure 4.23. Property 3(b)iv of the schedulability predicate requires that m_2 is not schedulable unless $1LBLCAT(m_1, m_2)$ has committed. By reasoning over the message paths of m_1 and m_2 it can be deduced that $1LBLCAT(m_1, m_2)$ has committed. This is because $m_1 \prec m_2$ and $LCA(m_1, m_2)$ and $1LBLCAT(m_1, m_2)$ are synchronous with respect to each other. For these reasons, the ancestor of $1LBLCAT(m_1, m_2)$ must have returned to $LCA(m_1, m_2)$ before the ancestor of m_2 was sent by $LCA(m_1, m_2)$. Since transaction creating messages do not return before they have committed, $1LBLCAT(m_1, m_2)$ must have committed. Note that this information can be deduced locally, i.e. without asking the **Transaction**. Therefore, this approach has the potential of avoiding network communications.

4.7 The Wait-By-Necessity Extension

4.7.1 Scheduling and Return Dependencies

A wait-by-necessity message m is an asynchronous message that returns a result. The sender of m is not suspended at the time m is sent. Synchronization takes place when m's result is first used. Immediately after m is sent, a voucher object is returned to the sender. The result of m is eventually returned into its voucher. Note that there is a one-to-one relationship between m and its voucher. m's voucher cannot be shared by another wait-by-necessity message and m cannot have more than one voucher. When m's result is

[46] $x_1 x_2 ... x_n \prec y_1 y_2 ... y_m$ if $x_i = y_i$ $(1 \leq i < k)$ and $x_k < y_k$ for some $k \leq m, n$; or if $x_i = y_i$ $(1 \leq i \leq n)$ and $n < m$ [Knu73].

first used (e.g, the result is sent a message or it is saved to permanent storage) then the voucher is *attempted to be redeemed.* There are two possible outcomes of an attempted redeem operation.

1. m's result has already been returned to its voucher. In this case, the redeem operation is *successful.* The requesting thread can use m's result and continue immediately.

2. m's result has not yet been returned to its voucher. In this case, the redeem operation is *unsuccessful.* The requesting thread is suspended until m's result is returned into its voucher.

Consider a simple example in Hermes/ST-like pseudocode. `C` is a class with methods `m1` and `m2`. `m1` and `m2` are defined as follows:

```
m1
    self x: 0.  "write access to variable x"
    v := self waitByNec; m2.
    ...some time consuming task...
    v display  "send message to v"

m2
    ^self x "perform read access and return"
```

Now consider the following scenario. Some object `o` of class `C` is sent the message m_1 in `o transactionCreating; m1`. m_1 sends message m_2 in `o waitByNec; m2`. Since m_1 writes to one of `o`'s variables, its lock type is `WriteLock`. Since m_2 reads one of `o`'s variables but does not write to any of `o`'s variables, its lock type is `ReadLock`. Thus, m_1 and m_2 are conflicting.

Since m_1 and m_2 create different transactional threads s_1 and s_2, the schedulability properties requires s_1 and s_2 to be serialized. Note that there is no return dependency between m_1 and m_2 *per se*. However, a return dependency occurs dynamically at runtime when `v` is attempted to be redeemed. m_1 cannot finish execution before m_2 has finished execution since it waits for m_2 to return a value. Note that such a *dynamic return dependency* cannot be detected statically before the execution of a message, e.g.,at compile time. This is because the redeem operation may depend on conditions, e.g.,user input, which cannot be anticipated.

This dynamic return dependency between m_1 and m_2 causes a deadlock situation. m_1 cannot finish execution since it waits for m_2 to return a value. m_2 is not schedulable since it is conflicting with m_1, and m_1 has not finished execution.

In accordance with Scheduling Property 2, serializability is not required in this case because of the dynamic return dependency relationship between m_1 and m_2. Note that as with static return dependencies, there is no problem of interleaving accesses when schedulability is guaranteed. This is for the following reasons:

- Before the dynamic return dependency occurs, i.e.,before m_2's voucher is attempted to be redeemed, serializability between s_1 and s_2 is maintained. Since m_1 and m_2 are conflicting, m_2 is not schedulable.

- When the dynamic return dependency occurs, i.e, when m_2's voucher is attempted to be redeemed, then serializability is not required. m_2 can then be scheduled. This schedule cannot lead to interleaving executions of m_1 and m_2 since m_1 is suspended

until m_2 has finished execution and has returned its result. This is analogous to a sender of a synchronous message being suspended until the synchronous message returns a result.

4.7.2 A General Form of Wait-By-Necessity

The term "wait-by-necessity" has been invented by Caromel [Car90] in a concurrent, but not distributed context. Also, transactions are not supported in Caromel's model. Since wait-by-necessity is the only kind of message passing supported, vouchers[47] that have not been redeemed can be returned as message results and passed as arguments to other messages. Vouchers are only attempted to be redeemed when they are first used, e.g. when the result is sent a message.

In this section it is shown that such a general form of wait-by-necessity, although elegant and useful in the concurrent context, is not suitable for a transactional, distributed context. This is because returning vouchers and actual results over node boundaries independently and maintaining serializability between wait-by-necessity threads that are not return dependent on each other *is* very expensive. An example below demonstrates this. For these reasons, a restricted form of wait-by-necessity is presented in Section 4.7.3 that can be implemented efficiently in a transactional, distributed context.

Consider the example of class `C` with four method `m1`, `m2`, `m3` and `m4:`. `m1`, `m2` and `m3` have no arguments. `m4:` has one argument. The definitions of the methods are shown below in Hermes/ST-like pseudocode. `barney` and `fred` are two instances of `C`.

```
m1
  self x: 0.  "write access"
  v2 := barney waitByNec; m2.  "send m2 to barney"
  v4 := fred waitByNec; m4: v2.  "send m4 with argument v2 to fred"
  v4 display  "send a message to v4"

m2
  v3 := fred waitByNec; m3.  "send m3 to fred"
  ^v3  "return voucher v3"

m3
  ^self x  "return result of read access"

m4: v
^v + 1  "send message to v. Return something"
```

Now consider the scenario of message m_1 being sent to `fred` in `fred transactionCreating; m1`. See Figure 4.24. After performing a write access to `fred`'s variable `x`, m_1 sends a message m_2 to `barney` in `barney waitByNec; m2`. m_2, in turn, sends message m_3 back to `fred` in `fred waitByNec; m3`. m_3 performs a read access to `fred`'s variable `x`. This access is conflicting to m_1's write access to `x`. Thus, m_1 and m_3 are conflicting. This read access is also the result of m_3 and is therefore returned to voucher `v3`. `v3` is the result of m_2 and is therefore returned to voucher `v2`. `v2` is then passed as an argument to message m_4 in `fred waitByNec; m4:v2`. There, it is sent a message (`+`) which finally causes `v2` to be redeemed. The result of message m_4 is returned to voucher `v4`. `v4` is then sent a message (`display`) in m_1 and is therefore redeemed.

Two observations can be made from this scenario:

[47] Vouchers are called "awaited objects" in [Car90].

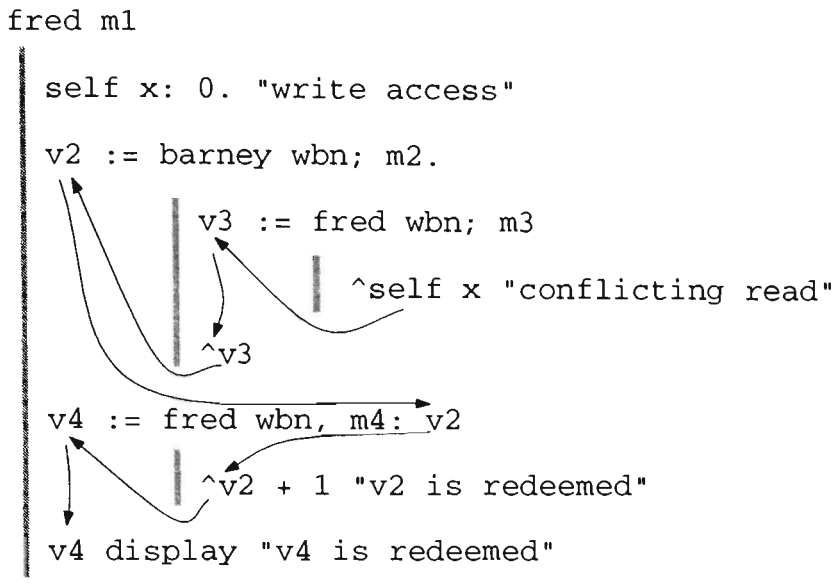

Figure 4.24: A scenario of wait-by-necessity messages.

1. There can be chains of voucher returns of arbitrary length. The result of the read access to **fred**'s variable **x** is performed in m_3. It is then returned to **v3**, then returned to **v2**, then passed as an argument to m_4 and then redeemed in m_4. See Figure 4.24.

   Assume that **fred** and **barney** reside on different nodes. Further assume that m_3 is computationally expensive such that m_4 attempts to redeem **v2** before m_3 has returned a result. Then, unredeemed vouchers must be passed from **fred**'s node (where m_3 is executed) to **barney**'s node (where m_2 is executed) and back to **fred**'s node (where m_1 and m_4 are executed). When m_3 finally returns a result then this result must be passed along the same route. This doubles the number of network communications necessary for returning the result of m_3 to m_4. In this case it is conceivable that passing m_3's result from **fred**'s to **barney**'s node and back can be avoided if such a cycle in a return chain is detected. However, detecting such a cycle requires at least the same amount of network communications as passing the actual result.

2. There can be chains of return dependencies of arbitrary lengths. There is no serial schedule for threads s_1 and s_3 that are created by m_1 and m_3, respectively. This is because m_1 and m_3 are conflicting and there is a dynamic return dependency between m_1 and m_3 occurring when **v2** is attempted to be redeemed in m_4. The dynamic return dependency is due to the following dependency chain. m_1 is suspended since it depends on the redeem of **v4**. The redeem of **v4** depends on the return of m_4. The return of m_4 depends on the redeem of **v2** and in turn on the redeem of **v3** since **v3** is assigned to **v2**. **v3** depends the return of m_3 which, of course, depends on the finish of execution of m_3.

This example demonstrates that, unlike static return dependencies, dynamic return dependencies can occur between messages that are not in an ancestor/descendant relationship. . with static return dependencies, m_3 can be scheduled safely without the danger of interleaving execution after the dynamic return dependency has been detected. However, the detection of dynamic return dependencies can be very expensive. It requires full knowledge of all unsuccessful redeem attempts in the execution of a system. This knowledge can be used to construct a graph where messages form the nodes and waits-for

relationships form the arcs. Cycles in this graph indicate dynamic return dependencies. Since such cycles can be of arbitrary length, cycle detection is NP-complete.

What makes this approach particularly unattractive is the fact that a high expense for detecting dynamic dependency relationships wold have to be paid even in the absence of dynamic return dependencies. Note that unsuccessful redeem attempts as such do by no means indicate a cycle. For example, there would not have been a return dependency between m_1 and m_3 if m_4 had been asynchronous or if m_1 had not redeemed **v4** but had returned it to its sender to redeem it instead. Unsuccessful redeem attempts can occur in the absence of dynamic return dependencies if wait-by-necessity messages takes a long time to return a result.

For these reasons, a less general form of wait-by-necessity is introduced below. It still provides a useful programming abstraction but can be implemented efficiently in a transactional, distributed context.

4.7.3 A Less General Form of Wait-By-Necessity

The less general form of wait-by-necessity requires that the voucher of a wait-by-necessity message m can only be redeemed within the * This means that unredeemed vouchers cannot be returned as results of messages and cannot be passed as arguments to messages.

This wait-by-necessity construct still provides a useful programming abstraction. The sender of a wait-by-necessity message m can continue to perform potentially computationally expensive tasks while m executes other, potentially computationally expensive tasks concurrently. This is particularly useful if m is remote, thus executing on a different processor than its sender. The redemption of m's voucher, and thus the synchronization of m's sender with the return of m's result, is performed as in the general wait-by-necessity model.

With this less general form of return dependency, dynamic return dependencies can only form between ancestor and descendant messages—like static return dependencies and unlike dynamic return dependencies in the general model. To see this, consider the five rules about message dependencies from Section 4.4.1 with extensions for the wait-by-necessity case. Extensions are emphasized by italics.

1. A message sending a synchronous message *or sending a wait-by-necessity message and attempting to redeem its voucher* waits until the submessage returns a result. Therefore, the finish of execution of a message depends on the return of synchronous submessages and *on the return of wait-by-necessity submessages if their vouchers are attempted to be redeemed.*

2. A message sending an asynchronous message is not suspended. Therefore, the finish of execution of a message does not depend on the finish of execution of asynchronous submessages.

3. A *wait-by-necessity or* synchronous non-transaction creating message returns immediately after it has finished execution. Therefore, the return of a *wait-by-necessity or* synchronous transaction creating message depends only on the finish of execution.

4. A *wait-by-necessity or* synchronous transaction creating message returns after the transaction it creates has committed or aborted. Therefore, the return of a *wait-by-necessity or* synchronous transaction creating message depends on the commit or abort of the transaction it creates.

* body of the message that sends m.

5. Transaction commit entails the finish of execution of the message itself, finish of execution of all threads that belong to it and the commit or abort of all descendant transactions. Therefore, the commit of a transaction depends on the finish of execution of all descendant messages.

The less general form of wait-by-necessity can be implemented efficiently in a transactional, distributed context. The extension of the scheduling mechanism to include wait-by-necessity messages is straightforward. The idea is that in terms of schedulability testing, wait-by-necessity messages are treated like asynchronous messages before they finish execution and before their voucher is attempted to be redeemed. However, after they finish execution or after their vouchers are attempted to be redeemed, they are treated like synchronous messages.

Then, serializability of transactional wait-by-necessity messages is ensured just as the serializability of transactional threads is ensured (Scheduling Property 1). Also, dynamic return dependencies are handled exactly in the way, static return dependencies are handled. If a voucher is attempted to be redeemed, then its wait-by-necessity message is treated like a synchronous message thus allowing the detection of return dependencies with the mechanisms introduced in Section 4.6. Therefore, schedulability is guaranteed in the face of return dependencies (Scheduling Property 2).

Obtaining the information whether a voucher is attempted to be redeemed is performed analogously to obtaining other scheduling information. Lazy information propagation and caching techniques are used. Since a voucher cannot be returned from a message and cannot be passed as an argument to other messages, the information whether a voucher is attempted to be redeemed is available local to the sender of the wait-by-necessity message. The sender's node serves requests about the redeem status of vouchers it has created. With a set `nodesToInform` and local caches with sets `objectsToInform`, it can be achieved that voucher redeem information is obtained only when it is needed and then only once. Note that to implement this informing mechanism, a message path must encode the location where a wait-by-necessity message is sent.

4.8 The Non-Serialized Transactional Thread Extension

Serializability of transactions is a useful property when transactional threads can interleave. However, if, due to the semantics of a particular application, transactional threads never interleave, it is desirable to avoid the expense involved in ensuring serializability. Take the example of a bank transfer which is performed via asynchronous withdraw and deposit operations. Assume that the deposit and withdraw operations only access their respective account objects. Since a transfer of funds is always performed from one account to a *different* account, the withdraw and deposit operations never interleave. Therefore, ensuring serializability of the two operations with respect to each other is unnecessary and wasteful. Note that the withdraw and deposit operations are both serialized with other transactions via the enclosing transfer transaction.

Also, non-serialized threads allow threads to communicate forth and backwards via shared data if this is required in an application. Note that with synchronized threads, this is not possible.

This is why existing transactional systems such as Avalon/C++ provide transactional threads that are not serialized with respect to each other. The Hermes/ST generalized message scheme is extended to include such non-serialized transactional threads. A new message parameter, `nonSerialized`, indicates the creation of such a thread.

The extension of the scheduling mechanism to deal with non-serialized threads is straight_forward. If an asynchronous non-serialized message is sent then asynchrony is created as for normal asynchronous messages. However, in terms of scheduling, an asynchronous non-serialized message is treated like a synchronous message. Note that messages which are synchronous with respect to each other are schedulable with respect to each other. Although non-serialized threads are not serialized with each other, they are still serialized with other serialized threads.

4.9 The Top-Level Extension

In the generalized message scheme presented so far, every message sent by another message is a submessage of its sender. Also, every transaction send by a transactional message is a subtransaction of its sender's transaction. From the experience with nested transactional systems it has emerged that in some applications it is advantageous to provide less strict semantics. Therefore, many nested transactional systems provide mechanisms for leaving the scope of an invoking transaction.

For example, Argus provides the `enter topaction...end` construct that allows the creation of top-level transactions from within (nested) transactions. Avalon/C++ allows the creation of top-level transactions and top-level threads from within (nested) transactions via the `toplevel` construct. It is stressed by the developers of both systems that these constructs should be used with care and only in situations where they are necessary. This is because they allow non-committed transactions to exchange data and therefore may defy transactional properties.

The generalized message scheme is extended to include the creation of top-level messages from within (nested) messages. This construct, although very simple, is more general than the constructs provided by Argus and Avalon/C++. In addition to creating top-level transactions and threads from within (nested) transactions, it allows the creation of top-level messages of all kinds and transaction characteristics: transaction creating, non-transaction creating, synchronous, asynchronous, and wait-by-necessity.

A new message parameter `topLevel` specifies that a message is defined outside its sender's scope. For example, the message `branch topLevel; transactional; addInterest` creates a top-level transaction like the `enter topaction...end` construct in Argus. Message `branch topLevel; asynchronously; updateView` creates a top-level thread like the `toplevel` construct in Avalon/C++. Message `branch topLevel; waitByNec getStatistics` creates a wait-by-necessity message outside the scope of its sender.

The implementation of top-level messages is straight_forward. Whenever a top-level message is sent then it is assigned a new top-level message path as if it was sent by a client. The scheduling mechanism then treats this message like a message sent by a client, i.e., independent from the scope in which it was actually sent.

Chapter 5

Discussion

In this thesis, linguistic mechanisms to specify the application of transaction and thread semantics to messages independently via parameters have been presented. Chapter 3 argues that such linguistic mechanisms are useful in terms of reusability, extensibility and maintainability. Chapter 4 specifies the semantics of independent threads and transactions in terms of scheduling properties. Although these scheduling properties are relatively complex to describe, they are intuitive and easily understood by application programmers. Basically, the interleaving of all kinds of transactional threads in any conflicting manner is avoided while their progress is guaranteed. Also, properties such as cascading abort free schedules and high concurrency are ensured. These properties do not affect the semantics but rather the performance of programs. A schedulability predicate and its implementation that satisfy the scheduling properties have been presented. Although the description and correctness discussions of both the schedulability predicate and its implementation are relatively complex, the algorithms are both efficient and easy to implement by system programmers.

The schedulability predicate has been implemented in Hermes/ST and aspects of the design that concern scheduling are described in Chapter 4. Hermes/ST employs single-version, pessimistic concurrency control based on locking. Hermes/ST's object model is fine-grained and deadlocks are detected via timeouts. However, the schedulability predicate and its implementation are independent of quite a number of these and other aspects.

Lock Mode: The schedulability predicate and its implementation can be used in combination with all kinds of lock modes including read/write locks, mutual exclusion locks, and user-defined type-specific locks. This separation of concerns is achieved via the use of the predicate "conflicting" in the definition of "schedulable with respect to". "conflicting" refers to the lock compatibility matrix of any lock mode that is used.

Deadlock Handling: The schedulability predicate and its implementation are independent to the deadlock handling mechanism employed. They can be used in combination with mechanisms that may lead to deadlocks, such as for example "general waiting" as well as in a combination with mechanisms that prevent or avoid deadlocks, such as, for example, "no waiting", "cautious waiting", "wound-wait," and "wait-die" [RSL87]. When "general waiting," is used then a negative outcome of the schedulability test causes the execution of a message to be delayed. When "no waiting" is used then a negative outcome of the schedulability test causes the abort of a transaction. When "cautious waiting", "wound-wait", or "wait-die" is used then

a negative outcome of the schedulability test either causes the delay of a message execution or a transaction abort.

Level of Concurrency Control Granularity: The schedulability predicate and its implementation are equally applicable to large-grained objects, medium-grained objects and small-grained objects. Also, they are independent of whether concurrency control is performed for whole objects or for objects' individual variables.

The independence of the schedulability predicate and its implementation from these parameters facilitates the comparison of mechanisms presented in this thesis with the respective mechanisms employed in other systems. This forms a major part of this chapter. A number of models and systems are selected that are representative of different scheduling approaches. Since, for example, Argus and Camelot/Avalon employ similar scheduling approaches, only one of these two important systems is compared. Moss' model is compared in Section 5.1, Argus in Section 5.2, Eden in Section 5.3, downward lock inheritance as used in LOCUS in Section 5.4, Venari/ML in Section 5.5 and KAROS in Section 5.6.

Section 5.7 presents the second part of this chapter. It shows some performance figures, obtained from the implementation of the scheduling mechanisms in Hermes/ST.

5.1 Moss' Model

Four years after submission of his thesis [Mos81], Moss published a book "Nested Transactions — An Approach to Reliable Distributed Computing" [Mos85]. This book is based on his thesis with only minor modifications and additions. In terms of scheduling, [Mos85] describes a slightly simpler model than [Mos81] in order to simplify the presentation of the mechanisms. For the same reason, the scheduling mechanisms presented in this thesis are first compared against the model presented in Moss' book. Most of the following sections present variations of this scheme, including the model presented in Moss' thesis.

5.1.1 Transactions

The transaction model of [Mos85] is as follows. Transactions can access (read or write) data items, which Moss calls "objects", and can create an arbitrary number of subtransactions. Subtransactions can execute synchronously or asynchronously. Transactions that have not been created by another transaction are called "top-level transactions". All transactions created in the execution of a system form a forest of transaction trees with top-level transactions as roots. The following restrictions are made.

- Only leaf transactions of transaction trees, i.e., transactions that do not create subtransactions are allowed to read or write objects.
- The only way of creating concurrency in the execution of a system is by creating asynchronous subtransactions.
- The model covers transactional operations only—non-transactional operations are not included.

5.1.2 Scheduling

When a leaf transaction accesses an object then it must acquire a lock. A lock must be acquired in read mode for a read access and in write mode for a write access. When the lock is granted then the transaction *holds* it until commit or abort.

Locks in Moss' terminology have a slightly different connotation from locks as used throughout this thesis. The model described in Chapter 4 includes concurrency controllers that are uniquely associated with objects. An individual lock has one particular mode and is uniquely associated with one particular message. In Moss' terminology, a lock itself is uniquely associated with a particular object—analogous to a concurrency controller in the terminology of Chapter 4. Various transactions can hold this lock in various modes—analogous to various transactions whose locks have been granted by the same concurrency controller in the terminology of Chapter 4.

At top-level transaction commit or abort, all locks held by the top-level transaction are released, after the respective recovery operations have been performed. At subtransaction commit, the parent of the committing transaction *"upward inherits"*[1] all locks, the subtransaction has held. Inherited locks act as placeholders. On the one side, they prevent transactions outside the holder's "universe" (i.e., non-descendant transactions) from interleaving in a conflicting way. On the other hand, they allow transactions inside this universe to acquire locks so that they have a chance of finishing successfully. Four locking rules are described.

1. A transaction can acquire a write lock if all transactions holding this lock are ancestors[2].

2. A transaction can acquire a read lock if all transactions holding this lock in write mode are ancestors.

3. When a transaction aborts then all the locks it holds are released. Ancestor transactions holding the same lock are not affected.

4. When a transaction commits then all the locks it holds are upward inherited by its parent transaction (if any).

Moss presents an optimization which is based on the fact that he considers read/write locking only. When a transaction that holds a lock in some mode upward inherits the same lock in another mode, then it does not have to hold the lock in two modes. Rather, the transaction only has to hold the lock in the maximum of the two modes. The maximum is defined by the total ordering $none < read < write$ of the lock modes where *none* denotes that the lock is not held at all.

Many aspects of Moss' model are described in an "algorithmic" way. For example, holding a lock only in the maximum of two lock modes is purely an optimization. It reduces the number of locks to compare against for schedulability testing. The semantics of the transaction model does not change, whether this optimization is performed or not. Lock upward inheritance is a mechanism that serves two purposes. It ensures serializability of transactions and avoids cascading aborts. The *mechanism* is described, rather than the *semantics* it aims to ensure. Nevertheless, Moss sees his model as purely conceptual. A particular implementation is not described.

[1] This inheritance mechanism is unrelated to the inheritance concept in object-orientation. Different terms are used for this concept, including "anti-inheritance", "upward lock inheritance" and simply "inheritance". In [Mos85], the simple term "inheritance" is used since its counterpart, "downward lock inheritance" is not discussed there. However, downward lock inheritance is discussed in Section 5.4. In order to make presentation unambiguous, the term "upward lock inheritance" is therefore used throughout this chapter.

[2] Moss uses the term "superior" instead of "ancestor".

5.1.3 Comparison

5.1.3.1 Terminology

What are termed "objects" in [Mos85] are not objects in the object-oriented sense. Rather, they are data items in the database sense. An object in [Mos85] holds one value and does not encapsulate a set of variables. Such a value is visible to clients via read and write access functions. The internals of objects are not hidden from clients. Functions, procedures, and transactions are invoked by clients and inspect and manipulate objects directly. In contrast, objects in the object-oriented sense can only be accessed via messages. The implementations of objects and their messages are hidden from clients.

These are important differences with respect to software engineering issues. However, since this thesis is mainly concerned with scheduling, these differences are not further discussed. To make a proper comparison of the scheduling semantics that the two mechanisms provide, a simple mapping of the concepts can be made. Functions and procedures in Moss' model are mapped to methods in the generalized message scheme. Function and procedure calls are mapped to messages. Transactions are mapped to transaction creating messages. Moss' objects are mapped to objects of the generalized message scheme.

In the generalized message scheme, all three restrictions of Moss' model are removed.

1. Every message can access its receiver object's variables and can send other messages. Thus, data accesses are not restricted to leaf transactions.

2. Non-transaction creating messages can be asynchronous. Thus, concurrency can be created other than by subtransactions only.

3. Transactional and non-transactional messages are included. Thus the model is not restricted to transactions only.

Moss' model can be seen as a subset of the generalized message scheme. Every program in Moss' model can be expressed directly in the generalized message scheme. For example, synchronous transactions are expressed by synchronous transaction creating messages. Asynchronous transactions are expressed by asynchronous transaction creating messages. The opposite is not true. For example, there is no equivalent to non-transactional messages, non-transaction creating transactional threads and wait-by-necessity messages in Moss' model.

For the subset of the generalized message scheme that is identical to Moss' model, scheduling properties are identical. Serializability is provided between top-level transactions and between asynchronous subtransactions. For the extensions of Moss' model, the semantics of his model have been extended in a natural manner. Consider, for example, transactional threads. In Moss' model, transactional threads are always associated with subtransactions. Serializability semantics are provided. The generalized message scheme extends the concept of transactional threads by additionally introducing non-transaction creating transactional threads. Again, serializability semantics are provided as for their transaction creating counterparts. Recall the discussion of the scheduling properties in Section 4.2.2.

5.1.3.2 Separation of Concerns

Chapter 3 presents in detail the advantages of the generalized message scheme. They can be paraphrased and summarized as follows.

Transactions are useful abstractions for reliable computing when the integrity of critical data is concerned. However, ensuring transactional semantics comes at a considerable

expense. Therefore, non-transactional operations are more efficient and sufficient when the integrity of data is not important. For example, in the banking domain, transactions should be used for account operations like deposits and withdraws while transactions should not be used for gathering statistical information.

Synchronous, asynchronous, and wait-by-necessity execution and their various variations are well-established and widely used mechanisms in concurrent and distributed programming. Moss' model combines the transaction aspect of an operation with its kind, i.e, synchronous or asynchronous. This forces application programmers to make compromises between the two concepts. They must, for example, use subtransactions if they want to create a new thread.

In contrast, the generalized message scheme allows the kind of operations to be specified independently from their transaction characteristics. This separation of concerns gives application programmers the full advantage of both concepts. If they want to create transactions then they can use transaction creating messages. If they want to create threads then they can use asynchronous messages. Also, as pointed out in Chapter 3, separation of concerns supports reusability, extensibility and maintainability.

5.1.3.3 Level of Concurrency

The use of serialized transactional but non-transaction creating threads in the generalized message scheme allows higher concurrency than asynchronous subtransactions in Moss' model. As pointed out in Section 4.2.2, these threads allow the application programmer to explicitly trade off the level of concurrency with the level of recovery provided by the system.

Recall the example for Scheduling Property 3 as described in Section 4.2.1 and shown in Figure 4.1. In this example, M_{11} and M_{15} have the same receiver object O and are conflicting. M_{11} has started execution before M_{15} is sent. Moss' upward lock inheritance mechanism handles this case in the following way. Assume that M_{11} reads O and M_{15} attempts to write O. Further assume that M_{11} is still executing. At this point in time, M_{15} is not schedulable since it cannot acquire a write lock. This is because T_{11} (the transaction of M_{11}) holds a read lock on O and T_{11} is not an ancestor of T_{15}. Thus, Locking Rule 1 is not satisfied. When T_{11} commits then its read lock is upward inherited by its parent transaction T_{10}. M_{15} still cannot acquire a write lock since T_{10} is not an ancestor of T_{15}. The same happens when T_{10} commits. However, when T_9 commits then the read lock is upward inherited by its parent transaction T_8. T_8 is an ancestor transaction of T_{15}. Thus, T_{15} can now acquire the write lock and is schedulable.

Recall that $T_8 = LCAT(M_{11}, M_{15})$. $T_9 = 1LBLCAT(M_{11}, M_{15})$. This example demonstrates that when the transaction one level below the least common ancestor commits, then locks acquired by it and all descendant transactions are upward inherited to the least common ancestor transaction. At this point in time, other descendants of the least common ancestor transaction can acquire conflicting locks. One could say that upward lock inheritance and Locking Rules 1 and 2 "implement" the schedulability test that checks whether the transaction one level below the least common ancestor has committed. This can be showed easily via induction over the nesting levels of a transaction tree.

Thus, Moss' scheduling mechanism provides the same level of concurrency as the scheduling mechanisms for the subset of his model of the generalized message scheme. This is because conflicting messages become schedulable exactly under the same condition.

Now recall that non-transaction creating transactional threads are outside Moss' model. They allow higher concurrency than subtransaction creating transactional threads, as pointed out in Section 4.2.2. The same argument holds for a comparison of non-

transaction creating transactional threads of the generalized message scheme and asynchronous subtransactions in Moss' model. Thus, the extensions to Moss' model provide a higher level of concurrency than Moss' model does. Non-transaction creating transactional threads allow application programmers to explicitly trade-off the level of concurrency with the level of recovery.

5.1.3.4 Serializability between Ancestor and Descendant Transactions

Before comparing the individual mechanisms, let us define what is meant by serializability between asynchronous ancestor and descendant transactions. As usual, a serializable schedule is defined as a schedule whose effects are equivalent to a serial schedule. However, there cannot be a serial schedule between an ancestor and a descendant transaction. This is because an ancestor transaction cannot commit before all descendants have committed[3], and therefore before all descendants have started execution. On the other hand, a descendant transaction cannot commit before one of its ancestor transactions has started execution. This is because the descendant is created by the ancestor.

Therefore, a serial schedule between asynchronous ancestor and descendant transactions is defined as a schedule which is equivalent to a serial schedule of the two threads that include all data accesses of the two transactions but exclude their commit procedures.

In Moss' model, only leaf transactions are allowed to access objects. This means that no ancestor transaction ever performs any work other than creating subtransactions. With this access restriction, there is no problem with the synchronization of asynchronous ancestor and descendant transactions. This is because ancestor transactions never perform "real" work. Moss concedes that the access restriction severely limits programming in his model. The justification for the access restriction is to simplify the presentation of his mechanisms. He offers the following range of practical approaches to get around this restriction that have been adopted by various systems.

1. All data accesses are turned into subtransactions.
2. Parent transactions are always suspended while child transactions execute.
3. Conflicting data accesses of ancestor and descendant transactions are treated as errors.
4. Ancestor and descendant transactions can interleave in an uncontrolled way.

Data Accesses Turned into Subtransactions: In this approach, all data accesses performed by a non-leaf transaction are turned into synchronous subtransactions. These additionally created subtransactions perform nothing but data accesses. Thus, they are leaf transactions and the access rule is not violated. To avoid subtransactional overhead, Moss proposes that a real implementation can treat these additional transactions in a special way.

First consider the option that the conversions from data accesses into synchronous subtransactions are performed automatically by the system and invisibly to the application programmer. Then, serializability between asynchronous ancestor and descendant transactions cannot be guaranteed. Consider the example shown in Figure 5.1. Transaction T_1 creates an asynchronous subtransaction T_2. T_1 performs two write accesses $write_1$ and $write_2$ to an object. T_2 performs a write access $write_3$ to the same object. The timing

[3]The abort case is not considered here in order to give transactions the chance of finishing successfully.

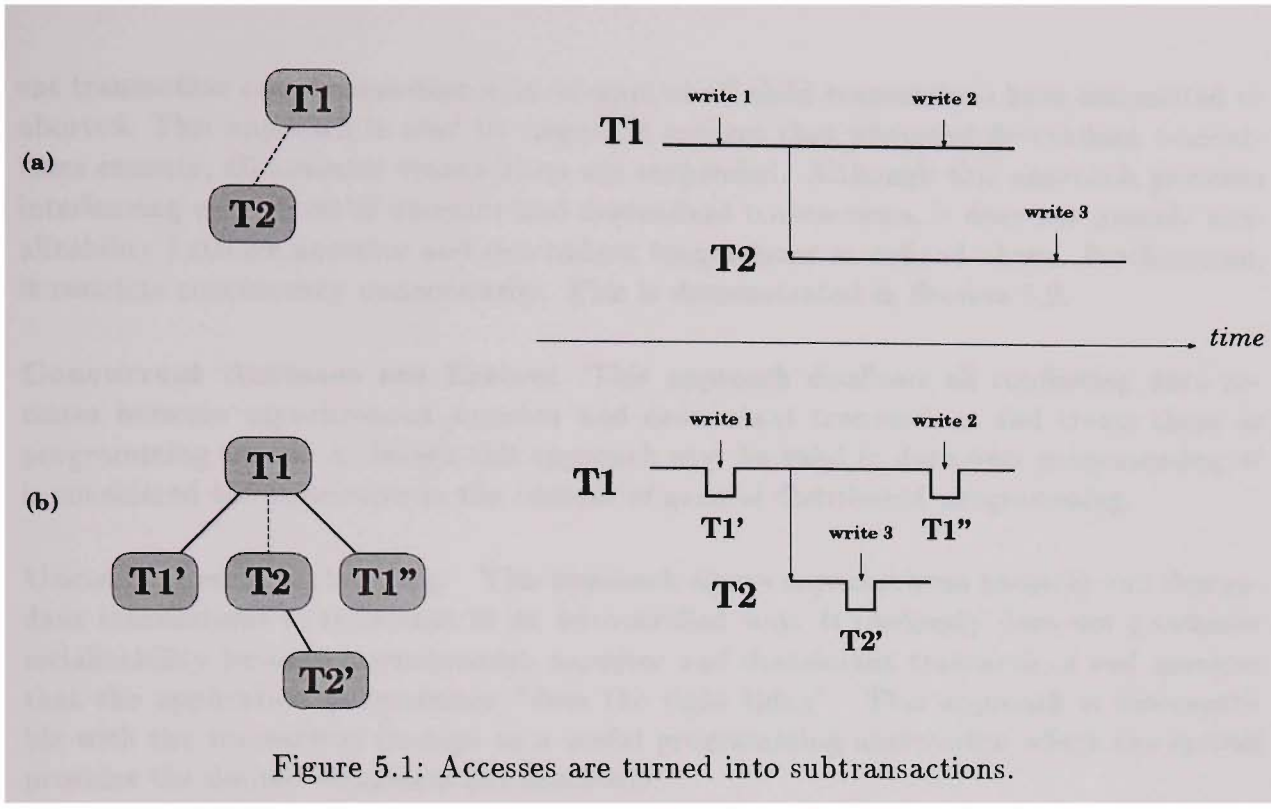

Figure 5.1: Accesses are turned into subtransactions.

diagram in Figure 5.1 (a) shows a possible serializable schedule between T_1 and T_2 where the write accesses are performed in order $write_1$, $write_2$, $write_3$.

Figure 5.1 (b) shows how the variable accesses of T_1 and T_2, $write_1$, $write_2$ and $write_3$, are turned into synchronous subtransactions T_1', T_1'' and T_2', respectively. Note that this transformation makes T_2 (the subtransaction specified by the application programmer) and T_1' and T_1'' (the subtransactions created by the system) sibling transactions. Using Moss' locking rules, the following schedule is allowed. T_1 starts execution and creates T_1'. T_1' acquires a write lock, performs $write_1$ and commits. T_1 then upward inherits the write lock. T_2 is created asynchronously. Assume that it creates T_2' before T_1 creates T_1''. Then, T_2' tries to acquire a write lock. This lock is granted since the holder of the write lock, T_1, is an ancestor. T_2' can then perform $write_3$ and commit. Successively, T_2 can commit and T_1'' can perform $write_2$. Thus, the locking rules allow a non-serialized schedule between asynchronous ancestor and descendant transactions T_1 and T_2 with the write accesses performed in order $write_1$, $write_3$, $write_2$.

This example demonstrates that an automatic conversion of data accesses to subtransactions does not guarantee serializability between ancestor and descendant transactions. Thus, application programmers must manually convert variable accesses into subtransactions. They must reason over the application semantics in order to guarantee serializability. This defies the purpose of transactions. Recall that transactions have been introduced as a programming abstraction where the underlying system ensures semantics like serializability.

Reconsider the example of Figure 5.1. The application programmer must ensure that T_2 is created after T_1'' has committed. Note that this modification of the program may not only obscure the definition of T_1 in an unnatural way. It may also restrict the concurrency considerably. Assume that T_1 performs time-consuming computations between $write_1$ and $write_2$, possibly remotely. Furthermore assume that T_2 performs time-consuming computations before $write_3$, possibly remotely. Then, the delay of T_2 may lead processors to be idle that could have, otherwise, performed operations concurrently.

Parent Transactions Suspended: The second approach to avoid the access restriction is to always suspend parent transactions while child transactions execute. The par-

ent transaction can resume execution as soon as all child transactions have committed or aborted. This approach is used by Argus. It ensures that whenever descendant transactions execute, all ancestor transactions are suspended. Although this approach prevents interleaving executions of ancestor and descendant transactions, it does not provide serializability between ancestor and descendant transactions as defined above. Furthermore, it restricts concurrency unnecessarily. This is demonstrated in Section 5.2.

Concurrent Accesses are Errors: This approach disallows all conflicting data accesses between asynchronous ancestor and descendant transactions and treats them as programming errors. Although this approach may be valid in data_base programming, it is considered too restrictive in the context of general distributed programming.

Uncontrolled Interleaving: This approach allows asynchronous ancestor and descendant transactions to interleave in an uncontrolled way. It obviously does not guarantee serializability between asynchronous ancestor and descendant transactions and assumes that the application programmer "does the right thing". This approach is incompatible with the transaction concept as a useful programming abstraction where the system provides the desired semantics automatically.

In contrast to all these approaches, the schedulability predicate for the generalized message scheme always provides serializability between ancestor and descendant transactions. This is because scheduling decisions are not only made on the basis of transaction commits. They are also made on the basis of the finish of execution of threads. Reconsider the example shown in Figure 5.1. The schedulability predicate ensures that $write_3$ is not performed before T_1's thread has finished execution, hence after $write_2$ has been performed.

5.1.3.5 Efficiency

Since Moss does not describe an implementation of his mechanisms, a comparison of the two mechanisms can only be performed on a conceptual level. In this section, it is shown that scheduling for the subset of the generalized message scheme that implements Moss' model is not more expensive than scheduling in Moss' model.

Both mechanisms need a data structure that resembles the position of a transaction in a transaction tree. The length of this structure is determined by the transactional nesting depth[4]. The implementation of the scheduling mechanism for the generalized message scheme uses the message path data structure. The length of a message path is determined by the number of transaction creating and/or asynchronous messages. For the subset of Moss' model, the length of a message path is the transactional nesting depth. This is because, in his model, transactional operations are considered only. Thus, the data structure describing the position of a transaction in a transaction tree has the same length for both mechanisms.

Both mechanisms require schedulability testing of requested operations compared to operations that have started execution. They are called "granted messages" in the scheduling mechanism for the generalized message scheme and "other transactions holding the lock" in Moss' model. In both mechanisms, this test is linear in the number of

[4]There are implementations of Moss' model that use fixed length nested transaction identifiers. For example, Camelot employs such an approach. For transactions whose nesting depth is within the limit of this fixed length, the performance discussions above apply. For deeper nested transactions, caching and informing techniques are used.

executing operations. Furthermore, for both mechanisms, the individual compatibility tests are linear in the transactional nesting depth. For the generalized message scheme, this has been shown in Section 4.6.2.8. For Moss' mechanisms this is easy to see since a test for ancestor relationship is performed.

In a naive implementation of upward lock inheritance, all locks held by a committing subtransaction are informed to perform upward inheritance. This approach potentially requires a large number of network communications since a committing transaction must not only communicate with all locks it has acquired but also all locks, all its descendant transactions have acquired. A more realistic strategy is called "lazy-evaluation anti-inheritance"[Lis84]. It describes a caching and informing mechanism. Scheduling information is only requested when needed. In this case, the question whether or not a transaction "has committed up to the least common ancestor" is asked. This strategy is equivalent to asking the transaction one level below the least common ancestor whether it has committed—the strategy used in the scheduling mechanisms for generalized message scheme.

Assuming the subset of Moss model, the scheduling mechanisms for generalized message scheme does not need more communications to obtain scheduling information than Moss' mechanism, using lazy-evaluation anti-inheritance. Note that in Moss' model, concurrency is only created via subtransactions, hence asynchronous messages are always transaction creating. This means that once $1LBLCAT(m_1, m_2)$ has committed, the thread of m_1 must also have finished. This information need not be requested separately.

The fact that in the generalized message scheme, transaction creation and thread creation are unified with the message concept allows further reduction of network communications than is possible in Moss' model. This is because message paths contain information about the message kind, synchronous or asynchronous. This information allows, in some cases, the deduction of the commit status of transactions which otherwise would have to be acquired remotely. Recall Section 4.6.2.9.

On aborts, both mechanisms perform the same operations. All locks held by an aborting transaction are released.

5.2 Argus

5.2.1 The Model

The scheduling mechanism adopted in Argus [Lis82, LS83, LCJS87, Lis88] is similar to Moss' mechanism. Since it is a major design goal of Argus to make distributed programming easier, the restriction that only leaf transactions can access objects is removed. The Argus approach to dealing with the interleaving of concurrent ancestor and descendant transactions is to disallow ancestor/descendant concurrency completely. Parent transactions are always suspended while asynchronous child transactions execute. Although there may be concurrency between sibling transactions, there is no concurrency between ancestor and descendant transactions. This approach is expressed in the linguistic constructs that Argus provides for creating transactions. Top-level transactions and synchronous transactions are created via the `enter action...end` construct. Concurrent subtransactions are created via the `coenter...end` construct. The `coenter...end` construct ensures that the invoking transaction is suspended until all child transactions have either committed or aborted; only then it is resumed.

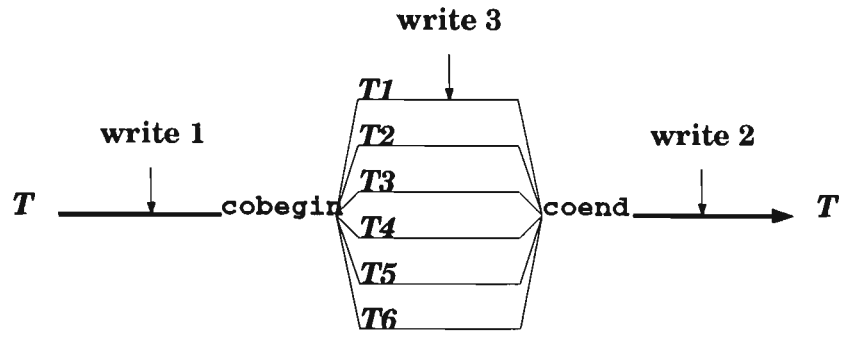

Figure 5.2: Ancestor/descendant synchronization in Argus.

5.2.2 Generality of the Model

Like Moss' model, the transactional model of Argus has a number of restrictions that are removed by the generalized message scheme.

- Concurrency can only be created via subtransactions.
- There is no ancestor/descendant concurrency.
- Every handler call implicitly creates a transaction. Therefore, non-transactional operations are not included in the model.
- A wait-by-necessity type construct is not provided.

The creation of top-level transactions from within subtransactionsis provided via the enter `topaction...end` construct. The generalized message scheme supports such a construct via the extension described in Section 4.9.

5.2.3 Scheduling

Scheduling in Argus is based on the locking rules of Moss' book as described in Section 5.1. Transactions can acquire a lock if all transactions holding the lock in a conflicting mode are ancestors. At transaction commit, locks are upward inherited to the parent transaction (if any). At transaction abort, locks are released.

5.2.4 Serializability of Ancestor and Descendant Transactions

Since Argus does not allow concurrency between ancestor and descendant transactions, there is no interleaving execution between ancestor and descendant transactions' threads. However, note that the Argus approach does not provide serializability between ancestor and descendant transactions as defined in Section 5.1. The reason is that there cannot be a serial schedule between a parent transaction and its subtransactions if they are created via the `coenter...end` construct. This is obvious from the fact that the parent transaction always starts before its subtransactions start and always finishes after the subtransactions have finished. In short, there cannot be serial schedules of ancestor and descendant transactions if there is no concurrency between ancestor and descendant transactions. Consider the example in Figure 5.2. A transaction T performs two write accesses $write_1$ and $write_2$ to the same data item. T creates a number of asynchronous subtransactions via the `coenter...end` construct. The subtransactions are created between the two accesses. One of the subtransactions, T_1, performs a write access $write_3$ to the same data item. Argus schedules the write accesses in the order $write_1$, $write_3$, $write_2$. This is not a

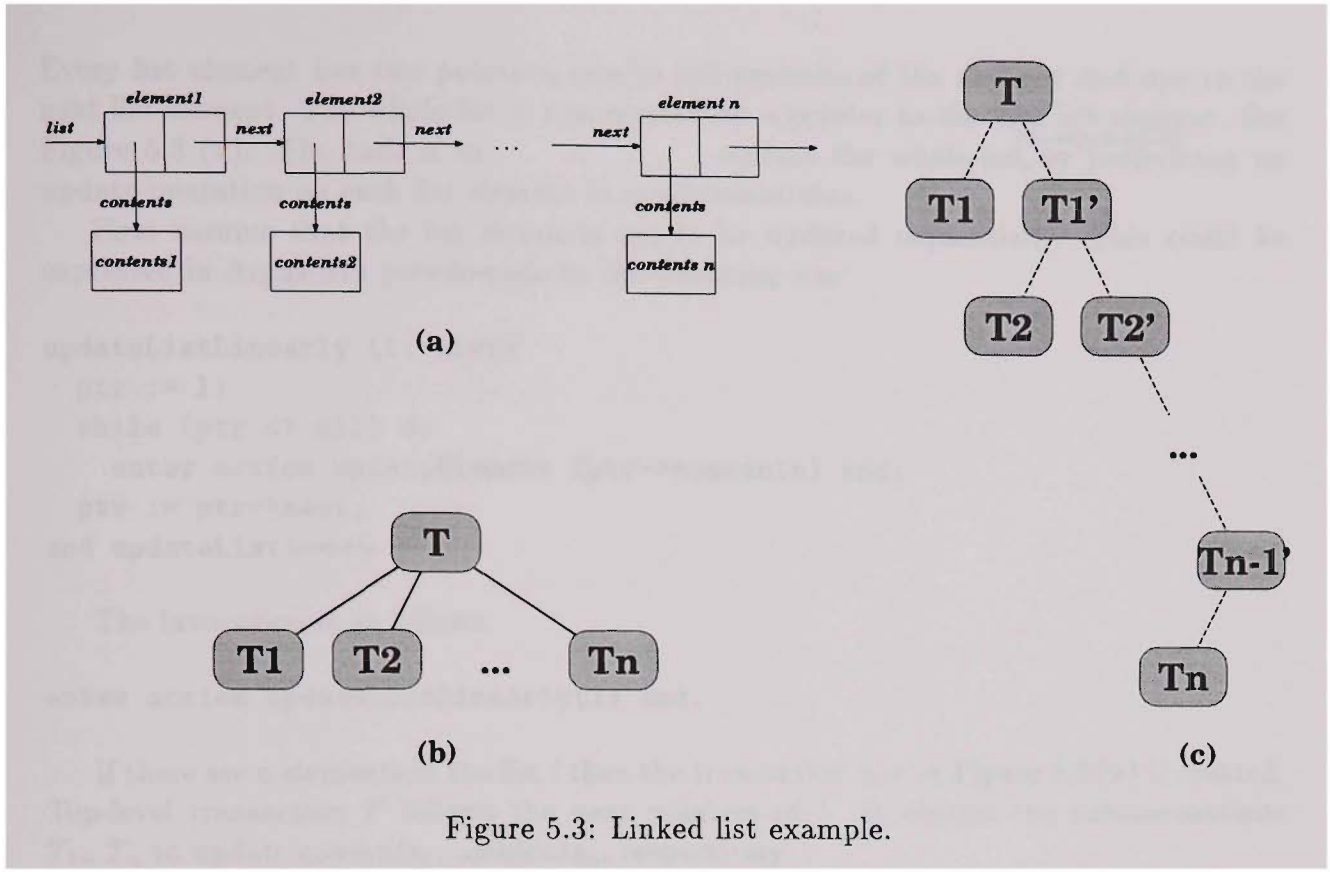

Figure 5.3: Linked list example.

serializable schedule between the ancestor transaction T_1 and the descendant transaction T_2. In a serializable schedule, the write accesses are either performed in order $write_1$, $write_2$, $write_3$ or $write_3$, $write_1$, $write_2$.

5.2.5 Level of Concurrency

An obvious disadvantage of the Argus approach to suspend parent transactions while subtransactions execute is that it restricts concurrency unnecessarily. It could be argued that higher concurrency can always be achieved in Argus by turning the transaction code after `coenter...end` into an additional concurrent subtransaction. Such a conversion could be performed either automatically by the system or manually by the application programmer. There are various drawbacks of this conversion approach which are discussed below.

5.2.5.1 Readability, Reusability and Maintainability

If such a conversion is left to the application programmer then the application program becomes unnecessarily obscured. Efficiency concerns have to be reflected in the structure of the programs. This adversely affects the readability and maintainability of code, but also the reusability of transactions in various contexts.

5.2.5.2 Increased Transactional Nesting Depth and Overheads

hether this conversion is performed manually by the application programmer or automatically by the system, it involves subtransactional overheads. These overheads include the activation of transaction handlers, recovery-related operations, commit notification, upward lock inheritance, and, in case of early writing, disk accesses.

Furthermore, the transactional nesting depth may be increased considerably by this approach, leading to higher expense in scheduling. Consider the example shown in Figure 5.3, which has been adapted from [HR93]. The data structure used is a linked list.

Every list element has two pointers, one to the contents of the element and one to the next list element. The whole list is represented by a pointer to the first list element. See Figure 5.3 (a). The task is to update the whole list transactionally by performing an update operation on each list element in a subtransaction.

First assume that the list elements are to be updated sequentially. This could be expressed in Argus-like pseudo-code in the following way:

```
updateListLinearly (l: list)
  ptr := l;
  while (ptr <> nil) do
    enter action updateElement (ptr->contents) end;
  ptr := ptr->next;
end updateListLinearly
```

The invocation is as follows.

```
enter action updateListLinearly(l) end.
```

If there are n elements in the list l then the transaction tree in Figure 5.3 (b) is created. Top-level transaction T follows the next pointers of l. It creates the subtransactions $T_1 ... T_n$ to update $contents_1 ... contents_n$, respectively.

Now assume that all update operations are to be performed concurrently. This cannot be achieved directly in Argus. This is because the parent transaction that follows the **next** pointers cannot execute concurrently with its subtransactions that perform the update operations for the individual elements. However, the approach proposed above shows how the Argus program can be restructured to provide the desired concurrency. The update operation of each element is performed in a concurrent subtransaction, using the `coenter...end` construct. The remaining code of the parent transaction, i.e. following the next pointer and updating the rest of the list, are turned into a sibling subtransaction. An implementation of this strategy can be expressed by the following recursive Argus-like pseudo code.

```
updateListConcurrently (l: list)
  if (l <> nil) then
    coenter ...  %two concurrent subactions:
      updateElement (l->contents) ...  % first subaction
      updateList(l->next) ...  %second subaction
    end
end updateListConcurrently
```

The invocation is as follows.

```
enter action updateListConcurrently(l) end.
```

This execution creates the transaction tree shown in Figure 5.3 (c). Transactions $T, T_1, ..., T_n$ correspond to transactions $T, T_1, ..., T_n$ in Figure 5.3 (b). Top-level transaction T performs the update of the whole list and subtransactions $T_1, ..., T_n$ perform the updates for $contents_1 .. contents_n$, respectively. $T'_1, ..., T'_n$ are the additional transactions created in order to achieve the desired level of concurrency. The level of transaction nesting has been increased dramatically from the constant number 2 to the length of the list, n.

In contrast, the concurrent update of a linked list can be expressed elegantly and efficiently in the generalized message scheme, without the need for additional transactions. Consider the following Hermes/ST like pseudo code for method `updateConcurrently` of class `LinkedList`.

```
updateConcurrently
  ptr := hermesSelf.
  [ptr notNil] whileTrue: [
    ptr contents asynchronously; transactionCreating; update.
    ptr := ptr next]
```

The invocation is as follows.

```
list transactionCreating; updateConcurrently.
```

The sequential version can be obtained by simply omitting the `asynchronously` parameter for the `update` message. The execution of `list transactionCreating; updateConcurrently` creates the same transaction tree as shown in Figure 5.3 (b), where the solid lines are replaced by dashed lines.

5.2.5.3 Synchronization of Ancestor and Descendant Transactions

As pointed out above, concurrent subtransactions created by `coenter...end` are not serialized with their parent transactions. However, Argus provides other, clean semantics for the order of execution of parent and subtransactions. First, the first part of the parent transaction is executed. Then, all subtransactions are executed in a serializable schedule. Finally, the second part of the parent transaction is executed.

This clean semantics is lost when the mechanism for achieving higher concurrency is used. Although serializability between subtransactions is guaranteed, it is non-deterministic when the second part of the parent transaction is scheduled. Thus, applying this mechanism not only changes the performance but may also change the semantics of an implementation.

In contrast, the scheduling mechanism for the generalized message scheme always provides the highest level of concurrency. Application programmers do not have to modify their code in order to achieve a desired level of concurrency. They do not have to risk software errors due to the semantic changes that these modifications may involve. Furthermore, serializability semantics are always guaranteed for asynchronous ancestor and descendant transactions. This is because scheduling decisions are not only made on the basis of transaction commits and aborts but also on the basis of the finish of execution of threads.

5.3 Eden

5.3.1 The Model

Eden is a distributed programming environment that supports nested transactions [PN85, ABLN85]. In Eden's transaction model, all transactions can access data items, called "*Eden objects*" or "*Ejects*". Every transaction can create a number of synchronous and asynchronous subtransactions. Concurrency can be created via the `COBEGIN...COEND` construct. Concurrent threads created via `COBEGIN...COEND` can, but do not have to, create transactions. Unlike the `coenter...end` construct in Argus, the thread performing

a **COBEGIN...COEND** construct is not suspended during the execution of the concurrent subthreads. Thus, ancestor/descendant concurrency is provided. Unlike Argus, Eden does not provide a construct to leave the scope of a transaction, e.g., to create a top-level transaction from within a subtransaction.

5.3.2 Scheduling

Eden allows non-leaf transactions to access Ejects. Furthermore, ancestor/descendant transaction concurrency is provided. Thus, the scheduling rules specified in Moss' book as described in Section 5.1 are not sufficient to provide serializability between ancestor and descendant transactions. Thus, Eden employs the scheduling mechanism described in Moss' thesis [Mos81].

5.3.2.1 Holding Locks versus Retaining Locks

In Moss' thesis, a distinction is made between a transaction *holding* a lock and a transaction *retaining* a lock. A transaction holds a lock if the transaction itself has acquired the lock because it performs data accesses. A transaction holds a lock until it commits or aborts. A transaction retains a lock if one of its descendants has held this lock and the lock has been upward inherited to this transaction. The explicit distinction between holding and retaining locks allows other transactions to distinguish whether a lock has been acquired by an ancestor or by a non-ancestor that belongs to the same transaction tree. The locking rules are as follows.

1. A transaction can hold a lock in write mode if no other transaction holds the lock and all transactions retaining the lock are ancestors.
2. A transaction can hold a lock in read mode if no other transaction holds the lock in write mode and all transactions retaining the lock in write mode are ancestors.
3. At subtransaction commit, the parent transaction retains all locks held or retained by the committing subtransaction.
4. At transaction abort and top-level commit, all locks held or retained are released.

These locking rules provide serializability between ancestor and descendant transactions. Assume that there is a lock conflict between an ancestor and a descendant transaction. Consider two cases.

1. The descendant transaction has acquired the lock before the ancestor transaction.
2. The ancestor transaction has acquired the lock before the descendant transaction.

Descendant Before Ancestor: In this case, Locking Rules 1 and 2 ensure that the ancestor transaction cannot acquire the lock unless the descendant transaction has committed and its lock has been upward inherited to the ancestor transaction. This schedule is equivalent to the serial schedule "descendant before ancestor" and is therefore serializable.

Ancestor Before Descendant: In this case, Locking Rules 1 and 2 ensure that the descendant transaction cannot acquire the lock before the ancestor transaction has committed. However, the ancestor transaction cannot commit unless all of its descendant transactions have either committed or aborted. A deadlock situation occurs that can only be resolved by aborting either the descendant or the ancestor transaction.

If the ancestor transaction acquires the lock before the descendant transaction is created then there is no point in retrying the failed transaction. Every retry will lead to the same deadlock situation. Thus, such a transaction is *de facto* regarded as a programming error.

5.3.2.2 Non-Transaction Creating Transactional Threads

Transactions can create non-transaction creating threads via the `COBEGIN...COEND` construct. No serializability semantics is provided for such threads. They can interleave in an unrestricted way with respect to each other. However, since serializability between transactions is ensured, these threads cannot interleave with other transactions in a conflicting way.

5.3.3 Comparison

The scheduling mechanism in Eden always leads to an ancestor/descendant deadlock if an ancestor transaction acquires a lock before a descendant transaction tries to acquire the same lock. In contrast, the scheduling mechanism for the generalized message scheme never deadlocks in such a case. This is because scheduling decisions are not only based on transaction commits and aborts but also on the finish of execution of threads.

Non-transaction creating transactional threads in Eden are treated like non-serialized threads in the generalized message scheme as described in Section 4.8. There are no serialized non-transaction creating threads in Eden.

5.4 Downward Lock Inheritance

5.4.1 Simple Downward Lock Inheritance

The concept of downward inheritance of locks is an extension of the scheduling mechanism described in Moss' thesis [Mos81]. Recall the distinction between holding and retaining locks and the locking rules, as described in Section 5.3.2.1. A linguistic construct is introduced that allows ancestor transactions explicitly *offer* locks that they are holding to descendant transactions. Descendant transactions can then acquire the offered lock. This is expressed in the following additional locking rule.

- A transaction holding a lock can offer the lock to descendant transactions. After offering the lock, the transaction retains the lock in the same mode it held it.

When the transaction later wants to hold the lock again then it has to wait until descendants holding the lock have committed or aborted, i.e., until the lock has been upward inherited back to the transaction. Such a downward lock inheritance mechanism has been implemented in LOCUS [MMP83].

5.4.2 Controlled Downward Lock Inheritance

In [HR93], the simple downward lock inheritance concept is extended to a concept called "controlled downward lock inheritance". The concepts of *upgrading* and *downgrading* locks

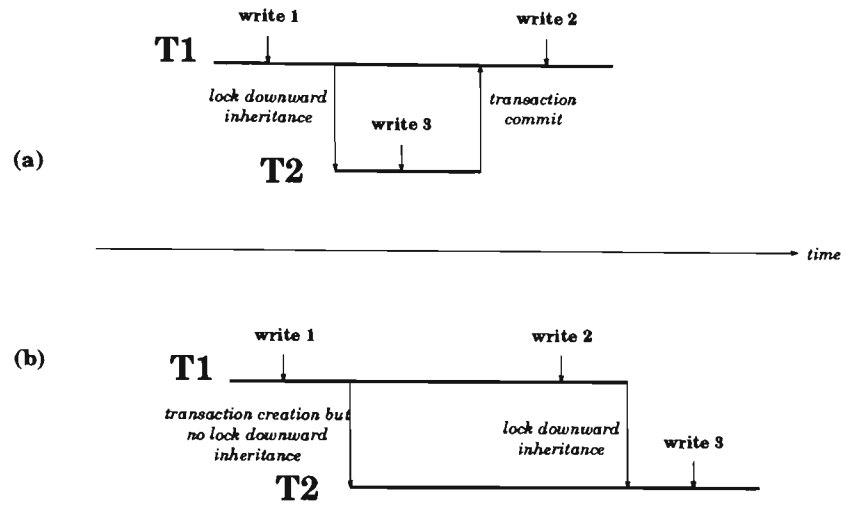

Figure 5.4: Lock downward inheritance.

are introduced. They allow explicit specification of the type of mode in which descendants are allowed to hold a lock. The following two locking rules express this concept.

- A transaction may upgrade a lock from read mode to write mode if no other transaction holds this lock and any transaction retaining this lock is an ancestor.
- A transaction may downgrade a lock it holds from write mode to read mode. It then retains the lock in write mode.

5.4.3 Analysis

Downward lock inheritance and its extensions, upgrading and downgrading of locks, allows the application programmer to explicitly modify transactional scheduling semantics on a per-transaction basis. Note that downward lock inheritance allows application programmers to explicitly defy serializability between ancestor and descendant transactions. Consider the example shown in Figure 5.4 (a). T_1 performs a write operation $write_1$ and therefore acquires a write lock. It offers this write lock to its descendant T_2. T_2 performs another write operation $write_3$ to the same data item and acquires the offered lock. After T_2 has committed, the lock is upward inherited by T_1. It can then re-acquire the lock to perform a write operation $write_2$. The explicit lock offer allows, in this case, the non-serializable schedule $write_1$, $write_3$, $write_2$.

However, downward lock inheritance can also be used to ensure serializability between asynchronous ancestor and descendant transactions. This can be achieved if a transaction offers all the locks it holds to descendant transactions after it has performed its last data access. Since the transaction does not perform any further data accesses, serializability is ensured.

Consider the example shown in Figure 5.4 (b). Transaction T_1 acquires a write lock and performs two write operations $write_1$ and $write_2$. After both write operations have been performed and it is ensured that no further data is accessed, T_1 offers the lock to its descendants. T_2 can then acquire the lock to perform the write operation $write_3$ to the same data item. The serializable schedule $write_1$, $write_2$, $write_3$ is achieved.

5.4.4 Comparison

Lock downward inheritance and the extension to include upgrading and downgrading of locks allow application programmers to explicitly modify the performance and scheduling

characteristics of applications. As shown above, they can explicitly violate serializability as well as ensure serializability. Furthermore, using application specific knowledge, they can ensure serializability and achieve higher concurrency than is possible with the schedulability predicate for the generalized message scheme. This is the case if an ancestor transaction offers locks to its descendants before it finishes execution but after it is sure, via the semantics of the application, that these locks are not further used by the ancestor transaction. In order to make such optimizations, the application programmer must carefully reason over both application semantics and the semantics of the scheduling mechanisms.

In contrast, the schedulability predicate for the generalized message scheme always ensures serializability for all kinds of serialized threads. The highest possible concurrency is achieved without using application-specific knowledge. However, the application programmer does not have the ability to use application-specific knowledge in order to increase concurrency further. Also, the application programmer cannot defy serializability in the way that downward lock inheritance allows.

To paraphrase, downward lock inheritance is an explicit mechanism where the application programmer is responsible for ensuring the desired semantics. The scheduling mechanism for the generalized message scheme is an implicit mechanism where the system is responsible for ensuring the desired semantics.

Other important aspects of the the generalized message scheme such as, e.g., the interplay of transactional and non-transactional messages, serialized transactional threads that do not create a subtransaction, non-serialized threads, and return dependencies are not addressed by the downward lock inheritance mechanism and can, therefore, not be compared.

5.5 Venari/ML

Venari/ML [WFMN92, NW91, HKM+94] is the only system the author is aware of that extends the traditional nested transaction model in a similar fashion to Hermes/ST. Not only are transaction semantics separated from thread semantics, but individual transactional properties can also be applied independently. As in Hermes/ST, the separation of concerns is a key idea of Venari/ML and Venari/ML goes even further than Hermes/ST. Even though Venari/ML is not object-oriented and not yet distributed, its similar design goals make it well worth comparing with Hermes/ST.

The notations used in various Venari/ML publications differ slightly. In this section, the notations of [HKM+94] are used. The term "transaction" is redefined to describe a thread or a group of threads. Every transaction can be invoked synchronously or asynchronously. The predicates "*persist*", "*undo*" and "*locking*" can be applied independently to transactions. Persist, undo and locking roughly correlate to the transactional properties permanence, atomicity, and serializability, respectively. A transaction that has all three properties is called a "*regular transaction*". Regular transactions have the semantics of transactions in the traditional sense. The other seven combinations provide weaker semantics but are also less expensive than regular transactions. Transactions can be arbitrarily nested, thus providing nested transactions and a wide range of other useful semantics.

Venari is implemented on top of the functional programming language Standard ML [MTH90]. Transaction creation and thread creation is specified via higher order functions. The following syntax is used. `f a` denotes the application of function `f` to argument `a`. No transaction or thread is created. `(transact f) a` first applies the higher order function `transact` to `f` which returns a function with regular transaction semantics. This function

is then applied to `a`. `transact` can be applied to any function `f` regardless of its semantics and implementation.

Thread creation is specified similarly. `(fork f) a` specifies that function `f` is applied to argument `a` asynchronously.

5.5.1 Generality of the Model

Venari/ML's transaction model is more general than the generalized message scheme. Like the generalized message scheme, it includes the following extensions of the transactional nested transaction model:

- Transactional and non-transactional operations are included.
- Every transaction can access data, not only leaf transactions.
- `fork` creates concurrency without necessarily creating a subtransaction.
- Ancestor/descendant concurrency is supported.
- The model includes both transactional threads that are serialized with respect to each other and transactional threads that are not serialized with respect to each other.

Additionally, Venari/ML allows various transactional features to be applied independently.

5.5.2 Scheduling

In terms of scheduling, however, Venari/ML is much less sophisticated than Hermes/ST. Venari/ML provides two kinds of locking:

1. Read/write locking is used for transactions. Locks are explicitly acquired in the function code and are released by the system according to 2PL.
2. Mutual exclusion locking is typically used for non-transactions. Locks are explicitly acquired and released in the function code.

With mutual exclusion locking, application programmers are responsible for ensuring the desired scheduling semantics. Mutual exclusion locks can, for example, be used to synchronize non-transactional messages.

With read/write locking, the system ensures serializability semantics. Venari/ML uses the simple locking rules of Moss' book as described in Section 5.1. No distinction is made between holding and retaining locks. Locks can be acquired if all conflicting locks are held by ancestor transactions. At transaction commit, all locks are upward inherited. At top-level transaction commit and transaction abort, locks are released.

5.5.3 Serializability of Ancestor and Descendant Transactions

Venari/ML does not restrict data accesses to leaf transactions. Ancestor/descendant concurrency is not restricted. Also, no distinction is made between holding and retaining locks. This means on the one hand that no deadlock between ancestor and descendant transactions can occur as in Eden. However, on the other hand it means that ancestor and descendant transactions can interleave in an uncontrolled manner. Serializability of

asynchronous ancestor and descendant transactions is not provided. If application programmers want to guarantee serializability in this case, they must implement it explicitly via mutual exclusion locks.

In contrast, the scheduling mechanism for the generalized message scheme always ensures serializability between asynchronous ancestor and descendant transactions.

Transaction and thread semantics in Venari/ML are applied to functions—analogously to the generalized message scheme. Thus, return dependencies can arise between ancestor and descendant transactions in exactly the same way. In contrast to the scheduling mechanism for the generalized message scheme, this issue is not addressed by Venari/ML's scheduling mechanism.

5.5.4 Level of Concurrency

Although Venari/ML provides non-transaction creating transactional threads that provide serializability semantics, it schedules them like regular transactions. This is because Moss' locking rules are generally used for all kinds of transactions. This takes away some of the attraction of such threads since concurrency is unnecessarily restricted. This is because Moss' scheduling rules require transactions to commit all the way up to the least common ancestor. This not only ensures serializability but also avoids cascading aborts. However, serializability is already ensured when conflicting threads have finished execution. Recall the example for Scheduling Property 3 as described in Section 4.2.1.

In contrast, the scheduling mechanism for the generalized message scheme provides serializability of threads under the highest concurrency that can be achieved without using application-specific knowledge. This allows application programmers to explicitly trade-off the level of concurrency with the level of recovery in transactional threads.

5.6 KAROS

KAROS [GCLR92] is an object-oriented concurrent, but not distributed, programming system that supports nested transactions. KAROS is implemented in C++ [Str86]. It is the only transactional system the author is aware of that provides wait-by-necessity constructs in combination with transactions. For this reason, it is compared with the scheduling mechanism for the generalized message scheme.

5.6.1 The Transaction Model

Transactions in KAROS are implicit in that every message creates a new transaction. Three types of asynchronous messages are supported: `Apply`, `Call`, and `Send`.

Apply: The syntax for `Apply` is as follows:

```
res = Apply(server, class, method) << Arg1..<< ArgN;
```

When `Apply` is used, an *implicit future* object is returned to the sender immediately. An implicit future object is analogous to a voucher object as described in Section 4.7. The actual result is eventually awaited when the implicit future is first used or when it is sent an explicit `wait` message. Two subtransactions are always created when `Apply` is used: one for `method` and one for the remaining code of the sender's message. The first subtransaction commits after the execution of `method` has finished. The second subtransaction commits when the implicit future is awaited.

Call: The syntax for `Call` is as follows.

```
res = Call(server1, class, method) << Arg1 ... << ArgN;
if Failure(res))  /*alternative code */
  res = Call(server2, class, method) << Arg1 ... << ArgN;
else ..... /* normal code */
```

`Call` behaves exactly like `Apply` if there is no failure in the invocation of `method`. In case of a failure, a failure code is returned and the sender can perform some alternative action.

Send: The syntax for `Send` is as follows.

```
Send(server, class, method) << Arg1 ... << ArgN;
```

After issuing a `Send` message, the sender continues to execute in its current transaction. The sender does not expect any result from `method`. `method` is executed in an independent top-level transaction outside the scope of the sender's transaction.

5.6.2 Scheduling

KAROS uses the simple locking rules of Moss's book for synchronization (see Section 5.1).

5.6.3 Serializability

The scheduling mechanism for the generalized message scheme provides stronger semantics for wait-by-necessity messages than KAROS does for `Apply` and `Call` messages. See Figure 5.5 (a) which has been adapted from [GCLR92]. A transaction t_1 sends a message using `Apply` or `Call`. This creates two subtransactions $t_{1.1}$ for the remaining code of the sender and $t_{1.2}$ for the new message. Assume that there are three conflicting write accesses being performed by t_1, $t_{1.1}$ and $t_{1.2}$ as shown in Figure 5.5 (a). Since KAROS treats the remaining code of the sending transaction t_1 as subtransaction $t_{1.1}$, the order of accesses $write_2$ and $write_3$ is non-deterministic. Thus, both schedules $write_1$, $write_2$, $write_3$ and $write_1$, $write_3$, $write_2$, are possible.

In contrast, the scheduling mechanism for the generalized message scheme provides stronger serializability semantics in this case. Only the schedule $write_1$, $write_2$, $write_3$ is allowed. This is because the scheduling mechanism ensures serializability between the whole of the sending thread and the whole of the wait-by-necessity thread unless a dynamic return dependency is established. Such a return dependency is only established at the point where the sender awaits the result of the wait-by-necessity message.

5.6.4 Efficiency and Concurrency

Since KAROS creates a transaction for every message, application programmers do not have the option to save transactional expense when transactional semantics are not required. Also, higher concurrency for non-transaction creating transactional threads cannot be achieved.

The method of creating two subtransactions for `Apply` and `Call` messages is similar to the mechanism discussed in Section 5.2 to increase concurrency in Argus. It has the same drawback of creating deeply nested transaction trees where the application program suggests only a constant level of nesting. Consider the following example in KAROS-like pseudo code:

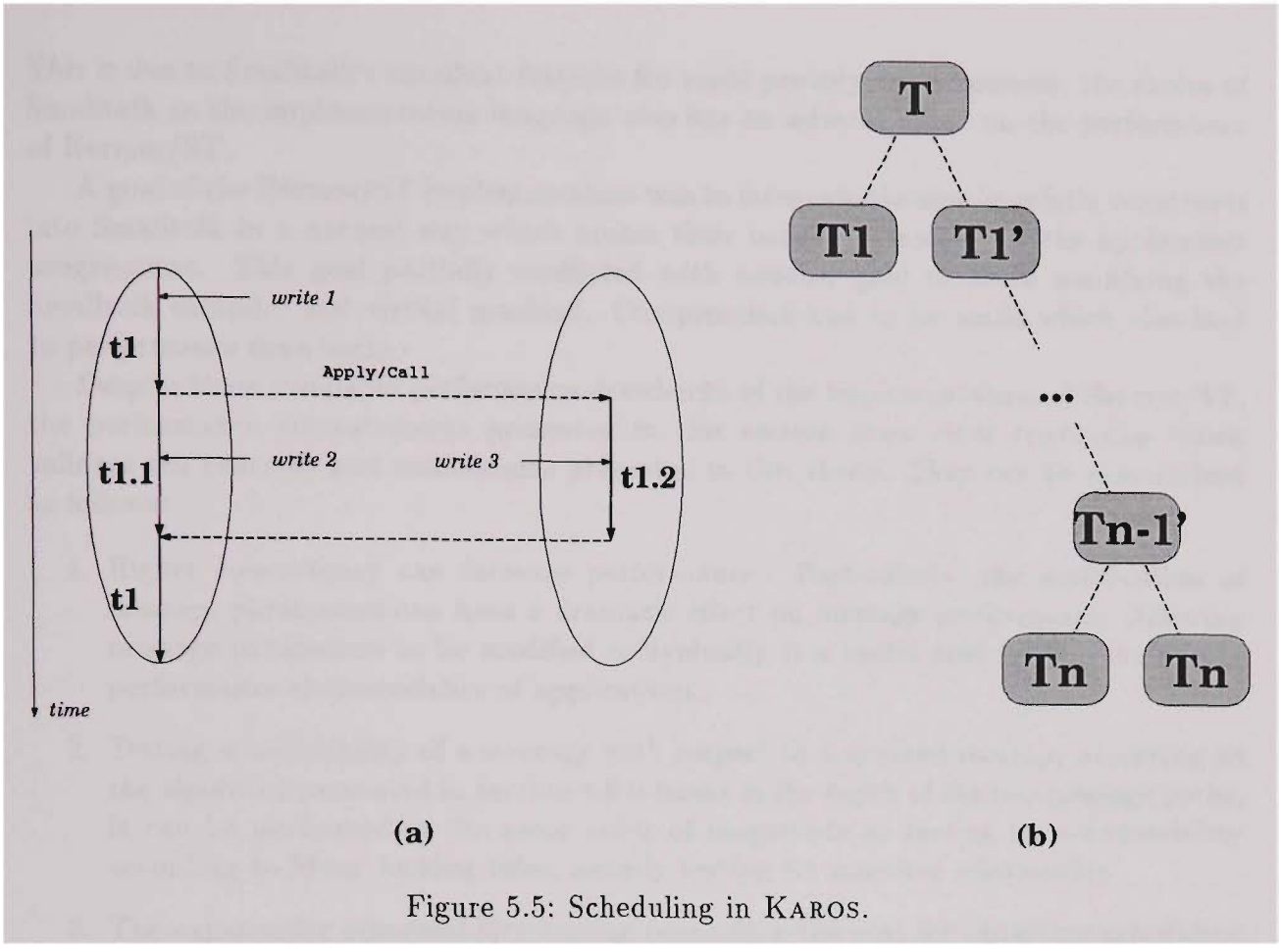

Figure 5.5: Scheduling in KAROS.

```
x1 = Apply(server1, class1, method1) << Arg1.1 ... « Arg1.M1;
x2 = Apply(server2, class2, method2) << Arg2.1 ... « Arg2.M2;
...
xN = Apply(serverN, classN, methodN) << ArgN.1 ... « ArgN.MN;
x = x1 + x2 + ... xN;      /* usage of implicit futures */
```

There is no nesting of messages in this example and thus the code suggests a flat transaction tree. However, the KAROS system creates a transaction tree of depth $n+1$ as shown in Figure 5.5 (b). In this figure, transaction T refers to the transaction sending all the `Apply` messages. $T_1 ... T_n$ refer to the subtransactions for `method1...methodN`. $T'_1 ... T'_n$ refer to the subtransactions that are additionally created by the KAROS system. Such a deeply nested transaction increases the expense for scheduling considerably and therefore affects the performance of programs in a negative way.

In contrast, an equivalent example can be programmed in Hermes/ST using transaction creating wait-by-necessity messages. The Hermes/ST system does not create more subtransactions than specified in the application program.

5.7 Performance Analysis

This section presents the second part of Chapter 5. It gives some performance figures for the implementation of the scheduling mechanism in Hermes/ST. Hermes/ST is a prototype implementation of concepts and mechanisms introduced in this thesis and various other publications [FHR93b, Faz94, Ran94]. Due to limited manpower, many obvious and well-known optimizations, e.g., for crash and abort recovery and 2PC, have not been implemented. The choice of Smalltalk as the implementation language facilitated the implementation of a complete and complex system in a relatively short period of time.

This is due to Smalltalk's excellent features for rapid prototyping. However, the choice of Smalltalk as the implementation language also has an adverse affect on the performance of Hermes/ST.

A goal of the Hermes/ST implementation was to integrate the new linguistic constructs into Smalltalk in a natural way which makes their usage convenient for the application programmer. This goal partially conflicted with another goal to avoid modifying the Smalltalk compiler and virtual machine. Compromises had to be made which also lead to performance drawbacks.

Despite these avoidable performance drawbacks of the implementation of Hermes/ST, the performance measurements presented in this section show clear tendencies which validate the concepts and mechanisms presented in this thesis. They can be summarized as follows:

1. Higher concurrency can increase performance. Particularly, the modification of message parameters can have a dramatic effect on message performance. Allowing message parameters to be modified individually is a useful tool for fine-tuning the performance characteristics of applications.

2. Testing schedulability of a message with respect to a granted message according to the algorithm presented in Section 4.6 is linear in the depth of the two message paths. It can be performed in the same order of magnitude as testing for schedulability according to Moss' locking rules, namely testing for ancestor relationship.

3. The expense for schedulability testing (excluding the cost for obtaining scheduling information remotely) is negligible compared to overall transaction costs.

4. Network communications are expensive and should be avoided if possible.

5.7.1 Modifying Message Parameters

Recall the bank transfer example, as described in Section 3.1, and its implementation in Hermes/ST, as listed in Appendix A.5.1. The transfer method is always invoked as a transaction. The withdraw and deposit methods may be sent synchronously, asynchronously, transaction creating or non-transaction creating, depending on their message parameters. The transaction created by the transfer message ensures that the semantics of the transfer operation are not changed, no matter what parameter setting is chosen for the deposit and withdraw messages. Message parameters for the message kind and transaction characteristics can be set independently. This allows four possible combinations which are shown in Table 5.1.

	asynch	synch
nonTrans	0.86 s	1.31 s
trans	1.73 s	2.18 s

Table 5.1: Transactional bank transfer with varying message parameters for deposit and withdraw.

The table shows the execution times for the whole transfer transaction, where the `Teller` object and the two `Account` objects all reside on different nodes. The parameters *nonTrans* and *trans* for the rows and *asynch* and *synch* for the columns specify the message parameters for both deposit and withdraw messages.

Performing the transfer operation synchronously and with two subtransactions (*synch trans*) is certainly the slowest option. It does not sufficiently utilize system resources,

namely the processors of the nodes on which the particular objects reside. Also, it provides an unnecessarily high level of recovery since a transfer operation is always aborted if either of the deposit or withdraw operations fails.

Two kinds of optimizations can be made: increasing concurrency and cutting down transactional nesting depth. The first optimization (*trans asynch*) reduces the execution time by over 20%. The second optimization (*nonTrans synch*) reduces the the execution time by 40%. Since the generalized message scheme allows message parameters to be modified independently, both optimizations can be performed together (*nonTrans asynch*). This reduces the the execution time by over 60%.

This example shows that the performance impacts of changing message parameters can be dramatic. Note that changing these parameters does not affect the semantics of the program. This makes the generalized message scheme a most useful tool for fine-tuning transactional applications.

5.7.2 Performance of Schedulability Testing

Section 4.6.2.8 shows that the complexity of the algorithm for the schedulability predicate is linear in the length of the message paths. The testing of the ancestor relationship, as performed in Moss' locking rules, is also linear in the length of the nested transaction identifier. Both the message paths and the nested transaction identifiers have the same length for all cases within the subset of Moss' model.

To validate this theoretical result, the performance of both algorithms has been monitored for a large number of pairs of message paths out of randomly generated message trees of various depths and breadths. The results are listed in Table 5.2.

depth	Hermes/ST	Moss
1	28.9 μs	25.5 μs
2	55.7 μs	27.8 μs
3	93.4 μs	30.4 μs
4	118.5 μs	32.5 μs

Table 5.2: Comparison of the performance of schedulability testing.

Column "depth" indicates the average nesting depth of the message paths compared. Column "Hermes/ST" shows the average execution time for schedulability testing according to the algorithm of Section 4.6. Column "Moss" shows the average execution time for performing a test for the ancestor relationship.

Table 5.2 shows that both figures rise monotonically and roughly linearly. Furthermore, the expense of schedulability testing for the generalized message scheme is in the same order of magnitude as testing for ancestor relationship.

5.7.3 Schedulability Testing versus Overall Transaction Cost

This section puts the results of the last section, namely the cost of individual schedulability tests, into the context of overall transaction costs. Measurements have been taken from executions of the banking system, as specified in Appendix A.5. For these tests, the transactional nesting depth was in the range 1–4 and the number of granted messages that an incoming message had to be compared with was in the range 0–10. Table 5.3 shows the average time for transactional transfer operations and the respective time spent on schedulability testing, excluding the time needed for obtaining scheduling information remotely.

overall transaction	schedulability testing	percentage
857 ms	0.11 ms	0.013 %

Table 5.3: Cost for schedulability testing in comparison to overall transaction costs.

The table suggests that the cost for schedulability testing is negligible compared to the overall transaction cost. The two main contributors to the transaction cost are disk accesses and network communications. That network communications are well worth avoiding if possible is shown in the next section.

5.7.4 Caching versus Asking Scheduling Information

From executions of the banking system, measurements have been taken to compare the run-time cost involved in obtaining scheduling information remotely, i.e., via asking a `Transaction` object, or locally, i.e., via its `TransactionCache`. Scheduling information includes information about the commit of transactions and the finish of execution of (partial) threads. The result is shown in Table 5.4.

	remote	local
scheduling information	58.8 ms	5.6 ms

Table 5.4: Obtaining scheduling information remotely and locally.

The table shows clearly that caching and obtaining scheduling information locally has enormous performance benefits compared to obtaining scheduling information remotely.

To summarize, the performance figures indicate the validity of concepts and mechanisms, described in this thesis. Particularly, the more general transaction model allows the performance tuning of applications in a way which is not possible in the traditional, less general transaction model.

The cost for scheduling in the more general transaction model is a small component of the overall transaction cost. However, what does affect the overall transaction cost is the number of network communications needed for obtaining scheduling information. This is why it is important that scheduling in the general model does not require more network communication than scheduling with the traditional mechanisms for the subset of the less general model. It can even be shown that, in some cases, even less network communications are needed (recall Section 4.6.2.9).

Chapter 6

Conclusions

In this thesis, novel linguistic constructs for distributed systems programming have been introduced. They include a generalized message scheme that allows transaction creation and thread creation to be specified independently over messages in the object-oriented sense. The generalized message scheme provides a richer set of programming abstractions than does the traditional nested transaction model. For this reason, the scheduling semantics of the traditional nested transaction model have been extended in a natural way to cover all abstractions provided by the generalized message scheme. An implementation-independent scheduling mechanism is presented that satisfies these scheduling semantics. Also, an efficient implementation of this scheduling mechanism is described.

The generalized message scheme has advantages over the traditional nested transaction model with respect to both system development and system execution. It facilitates a flexible "pick-and-choose" approach. Application programmers can pick the programming abstraction which is most suitable for a particular application, both in terms of semantics and performance. This is particularly important in the area of distributed systems programming where concurrency and the possibility of failures add enormous complexity and performance constraints are often hard.

The flexibility of the approaches presented has been achieved by consequent separation of concerns. Orthogonal concepts, such as, for example, transaction creation and thread creation that have been combined in the traditional nested transaction model, can be applied independently of each other and independent of the application code. Separation of concerns supports typical advantages of object-orientation like reusability, extensibility and maintainability. Particularly, it allows fine-tuning of the performance of existing applications without modifying their structure or semantics.

Although the definition of the scheduling semantics is relatively complex, their properties are intuitive and easy to understand by application programmers. Basically, serializability is provided for all kinds of transactional threads if possible and unless specified otherwise by the application programmer. If it is impossible to ensure serializability then the progress of threads is guaranteed. The fact that the properties are conceptually simple is important to their usefulness and acceptance by application programmers. Although the semantics cover a more general model, their properties are not more complex than their counterparts for the traditional, less general model. In fact, they are in some cases even simpler. Take the example of asynchronous ancestor and descendant transactions. The property provided by the general model is simple: serializability is guaranteed in any case. In existing systems that employ the traditional model, application programmers have to understand how the particular scheduling mechanism works. They then may have to modify their applications in order to ensure serializability manually or risk failures or deadlocks.

In terms of efficiency, it has been shown that the mechanisms for the more general model are not more expensive than the mechanisms for the less general model, as far as the subset of the less general model is concerned. For transactions that cannot be expressed in the less general model, only a small amount of work is performed since the number of network communications is minimized. It can even be shown that in certain cases, network communications can be saved where such savings are not possible with the traditional mechanisms. This is due to the fact that transaction creation and thread creation are unified with the message concept and the fact that message paths include thread information.

Another important advantage of the mechanisms proposed is the following. Although reasoning over the correctness of the scheduling mechanism and its implementation is relatively complex, the algorithms themselves are not very complex and can be adopted easily by system programmers.

To summarize, the semantics and mechanisms proposed in this thesis are more general than traditional semantics, are as efficient as traditional mechanisms, and are easy to implement. The combination of these three properties makes the adoption of these mechanisms well worthwhile. Although the results of this thesis are mature, they are regarded as only one step into an area that deserves more research: the separation of orthogonal concepts that have traditionally been combined in order to achieve both more flexibility during system development and more efficiency during system execution.

Take, for example, the transaction concept itself. Transactions provide useful and strong semantics but they are also quite expensive. For many real-world applications, the performance penalties of transactions are too high. Therefore, the "right" level of reliability is often achieved via hand-coding. This approach is not only unproductive and inflexible but also error prone. There are various research efforts to provide cheaper transactions. One approach is to weaken the transactional semantics, e.g., by weakening serializability. Another interesting approach has been proposed recently by Wing [HKM+94]. The idea is, again, the separation of concerns. Transactions comprise the three properties serializability, atomicity, and permanence. The individual properties can, in part, be applied independently. Initial results have been reported as part of the Venari/ML project at Carnegie-Mellon University.

Another area where the separation of concerns may increase flexibility during system development and efficiency during system execution is concurrency control granularity. The granularity of concurrency control can be separated from both object granularity and concurrency control specification. Both areas, separation of transactional properties and separation of concurrency control granularity, are currently investigated as part of the Hermes/ST project. They are only two examples of a wide range of possible continuing research in this area.

Appendix A

Hermes/ST Code Examples

A.1 The Binary Search Tree

A.1.1 The Tree Class

---

class	**Tree**
superclass	HermesSortedCollection
instance variables	rootNode
class variables	*none*
pool dictionaries	*none*
class category	Binary Search Tree

"Classes Tree and TreeNode implement the binary search tree data type. A binary search tree is a binary tree where the contents of every node (i.e. the elements of the tree) can be compared (i.e. provide two methods = and <) and are in the following relationship: for every node, all elements in the left subtree are less than the node contents itself and all elements in the right subtree are greater than the node contens. There are no two nodes with the same contents in the tree. Traversing the tree in pre−order results in a sorted list of all elements with the smallest element first and the largest elment last. Tree is defined in the following hierarchy:

HermesObject ()
HermesCollection ()
HermesSequenceableCollection ()
HermesOrderedCollection ()
HermesSortedCollection ()
Tree (*'rootNode'*)

Tree has one instance variable:

rootNode < *TreeNode*> *or* < *UndefinedObject*>

which represents the root node of the tree. If the tree is empty then rootNode is nil. Tree supports the complete interface, Collection provides. "

testing

isEmpty

```
↑self rootNode isNil
```

enumerating

do: aBlock
"evaluates 'aBlock' for each element in the tree. Traverses the tree in pre–order"

```
hermesSelf isEmpty
  ifFalse:
    [hermesSelf left do: aBlock.
    aBlock value: hermesSelf contents.
    hermesSelf right do: aBlock]
```

adding/removing

add: anObject ifExisting: aBlock
"adds 'anObject' to the tree. If 'anObject' is already existing in the tree then evaluates the exception 'aBlock'"

```
hermesSelf isEmpty
  ifTrue: [hermesSelf rootNode: (TreeNode instantiate: hermesSelf kind withContents:
anObject)]
  ifFalse: [hermesSelf contents = anObject
      ifTrue: [aBlock value]
      ifFalse: [anObject < hermesSelf contents
          ifTrue: [hermesSelf left add: anObject ifExisting: aBlock]
          ifFalse: [hermesSelf right add: anObject ifExisting: aBlock]]]
```

find: anObject ifAbsent: aBlock
"Finds a subtree with 'anObject' as root node. Returns this subtree if such a subtree can be found and evaluates the exception 'aBlock' otherwise "

```
hermesSelf isEmpty
  ifTrue: [aBlock value]
  ifFalse: [anObject = hermesSelf contents
      ifTrue: [↑hermesSelf]
      ifFalse: [anObject < hermesSelf contents
          ifTrue: [↑hermesSelf left find: anObject ifAbsent: aBlock]
          ifFalse: [↑hermesSelf right find: anObject ifAbsent: aBlock]]]
```

findLargest
"assumes that the tree is not empty. Finds and returns the subtree with the largest element as root. This subtree is always of depth 1 "

```
↑hermesSelf right isEmpty
  ifTrue: [hermesSelf]
  ifFalse: [hermesSelf right findLargest]
```

remove: anObject ifAbsent: aBlock
"removes 'anObject' from the tree. If 'anObject' is absent then the exception block 'aBlock' is evaluted. The algorithm first finds the subtree which contains the node to be removed as root. 3 cases are distringuished:

1. the subtree has no children. Then it is simply deleted;
2. the subtree has only one child. Then, the node is removed like in a linked list;
3. the subtree has two children. Then, the largest node of the left subtree is removed (another approach is to remove the smallest node of the right subtree), and the node that is to be removed is replaced by it. "

```
| subTree oldNode replacement |
subTree := hermesSelf find: anObject ifAbsent: [↑aBlock value].
subTree children
  if: [:c | c = #noChildren]
  then:
    [subTree rootNode delete.
    subTree rootNode: nil]
  elself: [:c | c = #leftOnly]
  then:
    [oldNode := subTree rootNode.
    subTree rootNode: oldNode left.
    oldNode delete]
  elself: [:c | c = #rightOnly]
  then:
    [oldNode := subTree rootNode.
    subTree rootNode: oldNode right.
    oldNode delete]
  elself: [:c | c = #twoChildren]
  then:
    [replacement := subTree left removeLargest.
    subTree rootNode contents: replacement]
```

removeLargest
"removes the largest TreeNode in the tree. Returns the contents of the removed TreeNode "

```
| largest contents |
largest := hermesSelf findLargest.
contents := largest contents.
largest rootNode delete.
largest rootNode: nil.
↑contents
```

A.1.2 The TreeNode Class

class	**TreeNode**
superclass	HermesObject
instance variables	left
	contents

	right
class variables	*none*
pool dictionaries	*none*
class category	Binary Search Tree

"Classes Tree and TreeNode implement the binary search tree data type. Tree is defined in the following hierarchy:

HermesObject ()

TreeNode (*'left' 'contents' 'right'*)

TreeNode has three instance variables:
left, right: < Tree> referring to the left and right subtrees
contents: < Object> referring to the contens of the node "

---

class	**TreeNode class**
superclass	HermesObject class
instance variables	*none*
class variables	*none*
pool dictionaries	*none*

instance creation

instantiate: kind withContents: anObject
"instantiates a TreeNode according to kind (#volatile or #persistent) with anObject as contents"

```
| inst |
inst := super instantiate: kind.
inst left: (Tree instantiate: kind).
inst contents: anObject.
inst right: (Tree instantiate: kind).
↑inst
```

A.2 Weighted Voting for Replicated Objects

A.2.1 Methods for Concurrent Collection Enumeration

---

class	**HermesCollection**
superclass	HermesObject
instance variables	*none*
class variables	*none*
pool dictionaries	*none*
class category	Hermes-Class Library

enumerating concurrently

collectInParallel: aBlock
"Concurrently evaluates 'aBlock' with each of the values of the receiver, a collection, as the argument. Collects the resulting values into a new SharedQueue in order of arrival. Returns the SharedQueue immediately."

```
| q |
q := SharedQueue new.
hermesSelf do: [:each | hermesSelf asynchronously; evaluate: [q nextPut: (aBlock
value: each)]].
↑q
```

doInParallel: aBlock
"Concurrently evaluates 'aBlock' with each of the values of the receiver, a collection, as the argument. Returns nil immediately"

```
hermesSelf do: [:each | hermesSelf asynchronously; evaluate: aBlock with: each].
↑nil
```

doInParallelAndWait: aBlock
"Concurrently evaluates 'aBlock' with each of the values of the receiver, a collection, as the argument. Returns hermesSelf after the last message has returned."

```
| q |
q := hermesSelf collectInParallel: aBlock.
hermesSelf size timesRepeat: [q next].
↑hermesSelf
```

A.2.2 The `ReplicatedObject` Class

---

class	**ReplicatedObject**
superclass	HermesObject
instance variables	versionNumber
	r
	w
	replicas
	contents
class variables	*none*
pool dictionaries	*none*
class category	Replication

"This class implements Gifford's weighted voting mechanism [Gif79]. Every replica of a replicated object is assigned a number of votes. A transaction that reads variables must acquire a read quorum r of votes; a transaction that writes variables must acquire a write quorum w of votes. Two restrictions apply for the choice of r and w with respect to the total number of votes v.

1. $r + w > v$. This ensures that there is always a non−null intersection between every read and write quorum. This ensures that every read operation returns the current version. Timestamps determine the age of a version.

2. $w > v/2$. This ensures that there can not be two partitions that have a write quorum at the same time.

Varying r and w within the range the two restrictions allow one to change the performance and availability charateristics of the replicated object.

Variables of ReplicatedObject:

versionNumber < Integer> current version number, is incremented for every write operation

r < Integer> read quorum; fixed for every replica of the replicated object

w < Integer> write quorum; fixed for every replica of the replicated object

replicas < List of ReplicaInfo> information about all replicas including their respective votes "

read/write access

read: variableName

"reads and returns a variable of a replicated object. First, votes are collected in parallel, using method collectInParallel:. As the votes arrive, it is tested whether the read quorum r is reached. Then, the latest version is returned (which is guaranteed to be the current version); votes coming in afterwards are not considered"

```
| latestVersionNumber collectedVotes queue replicatedObject votes
latestVersionContents |
  latestVersionNumber := -1.
  collectedVotes := 0.
  queue := self replicas keys collectInParallel: [:replObj | replObj
copyVersionNumberAndContents].
  [collectedVotes < r]
    whileTrue:
      [replicatedObject := queue next.
      versionNumber := replicatedObject versionNumber.
      votes := self replicas at: replicatedObject hermesSelf.
      collectedVotes := collectedVotes + votes.
      versionNumber > latestVersionNumber
        ifTrue:
          [latestVersionNumber := versionNumber.
          latestVersionContents := replicatedObject contents]].
  ↑latestVersionContents get: variableName
```

write: anObject to: variableName

"writes 'anObject' to the variable 'variableName' in a replicated object. First, votes for the write operation are collected concurrently using method 'collectInParallel:'. While votes arrive, it is tested whether the write quorum 'w' is reached. When this has happened, then the write operation is performed on all replicas of the quorum using method 'doInParallelAndWait:'. It is ensured that the replica with the largest version number is current. To keep versions as up-to-date as possible, replicas with older version numbers get other variables updated as well. 'doInParallelAndWait:' does not continue until all update operations have been completed.

To keep the replicas as up-to-date as possible, votes arriving after the write quorum has been reached ('lateComers') are updated as well using method 'doInParallel:'. Since

```
this update is not essential for maintaining the integrity of the replicated object, write:to:
does not have to wait for the update of the 'lateComers' and can return before."

  | latestVersionNumber collectedVotes queue votes latestVersionContents quorum
updatedContents lateComers |
  latestVersionNumber := -1.
  collectedVotes := 0.
  quorum := Set new.
  queue := self replicas keys collectInParallel: [:replicatedObject | replicatedObject
copyVersionNumberAndContents].
  [collectedVotes < w]
    whileTrue:
      [| replicatedObjectCopy |
      replicatedObjectCopy := queue next.
      versionNumber := replicatedObjectCopy versionNumber.
      votes := self replicas at: replicatedObjectCopy hermesSelf.
      collectedVotes := collectedVotes + votes.
      quorum add: replicatedObjectCopy.
      versionNumber > latestVersionNumber
        ifTrue:
          [latestVersionNumber := versionNumber.
          latestVersionContents := replicatedObjectCopy contents]].
  updatedContents := latestVersionContents set: variableName to: anObject.
  quorum doInParallelAndWait: [:replicatedObjectCopy |
    replicatedObjectCopy hermesPointer replaceContentsBy: updatedContents].
  lateComers := queue nextAll.
  lateComers doInParallel: [:replicatedObjectCopy |
    replicatedObjectCopy hermesPointer replaceContentsBy: updatedContents].
  ↑#done
```

A.2.3 The `ReplicaInfo` Class

class	**ReplicaInfo**
superclass	Object
instance variables	name
	location
	votes
class variables	*none*
pool dictionaries	*none*
class category	Replication

```
"This class describes information about a replica
    name < Symbol>  a name under which the replica can be accessed
    location < Symbol>  its location
    votes < Integer>  its number of votes "
```

A.3 Specification and Overriding of Message Parameters

A.3.1 Transfer Method in Class `Teller`

class	**Teller**
superclass	HermesObject
instance variables	name
	currencyTable
	interface
class variables	*none*
pool dictionaries	*none*
class category	Distributed Bank

transfer

```
transfer: amount from: branch1 name: accountNumber1 to: branch2
name: accountNumber2
"MessageParameters transactionCreating"

    branch1
        asynchronously;
        nonTransactionCreating;
        withdraw: amount from: accountNumber1.
    branch2
        asynchronously;
        nonTransactionCreating;
        deposit: amount to: accountNumber2.
    ↑#done
```

A.3.2 Deposit And Withdraw Methods in Class `Branch`

class	**Branch**
superclass	HermesObject
instance variables	name
	accounts
class variables	*none*
pool dictionaries	*none*
class category	Distributed Bank

deposit and withdraw

```
deposit: amount to: accountName
"MessageParameters transactionCreating"
    "
    ... method body ...
    "
```

```
withdraw: amount from: accountName
"MessageParameters transactionCreating"
  "
  ... method body ...
  "
```

A.3.3 Transfer Method in Class `AutomaticTellerMachine`

class	**AutomaticTellerMachine**
superclass	Teller
instance variables	*none*
class variables	*none*
pool dictionaries	*none*
class category	Distributed Bank

transfer

```
transfer: amount from: branch1 name: accountNumber1 to: branch2
name: accountNumber2
"MessageParameters timeout: 2"

  ↑super
    transfer: amount
    from: branch1
    name: accountNumber1
    to: branch2
    name: accountNumber2
```

A.4 Programmable Lock Definition and Usage

A.4.1 The `ProgrammableLock` Class

class	**ProgrammableLock**
superclass	Object
instance variables	metaObject
class variables	*none*
pool dictionaries	*none*
class category	Hermes-Programmable Locks

"ProgrammableLock is the abstract class for all programmable lock specifications. It defines the two methods isSchedulable: and isCompatibleWith: that can be overriden by descendant classes. ProgrammableLock is defined in the following hierarchy:

Object ()
  ProgrammableLock (*'metaObject'*)
    MutualExclusionLock ()
    NoLock ()
    ReadLock ()

AccountReadLock ('account')
FairReadLock ()
PeekLock ('isEmptyMethod')
SpreadSheetReadLock ('row' 'column')
WriteLock ()
AccountWriteLock ('account')
GetLock ('isEmptyMethod')
PutLock ('isFullMethod')
SavingsAccountsWriteLock ('account' 'typeCheckMethod')
SpreadSheetWriteLock ('row' 'column')

It has one variable

metaObject *< MetaObject> refers to the persistent object being locked "*

locking

isCompatibleWith: anotherLock

↑true

isSchedulable

↑true

guard methods

performGuard: guardMethod

↑self performGuard: guardMethod withArguments: #()

performGuard: guardMethod with: anObject

↑self metaObject performGuard: guardMethod withArguments: (Array with: anObject)

performGuard: guardMethod withArguments: anArray

↑self metaObject performGuard: guardMethod withArguments: anArray

A.4.2 The `AccountWriteLock` Class

---

class	**AccountWriteLock**
superclass	WriteLock
instance variables	account
class variables	*none*
pool dictionaries	*none*
class category	Distributed Bank

"AccountWriteLock is a lock to be applied to a whole branch. Logically, however, it locks a single account in write mode. It has one variable:

account *< Symbol> the name of the account locked. "*

locking

isCompatibleWith: otherLock

```
↑(super isCompatibleWith: otherLock)
  or: [self account ~= otherLock account]
```

A.4.3 Deposit Method of Class Branch

class	**Branch**
superclass	HermesObject
instance variables	name
	accounts
class variables	*none*
pool dictionaries	*none*
class category	Distributed Bank

in account operations

deposit: amount to: accountName

```
"MessageParameters
    transactionCreating;
    lock: [AccountWriteLock account: accountName]"

"
... method body ...
"
```

A.4.4 The SavingsAccountsWriteLock Class

class	**SavingsAccountsWriteLock**
superclass	WriteLock
instance variables	typeCheckMethod
class variables	*none*
pool dictionaries	*none*
class category	Distributed Bank

"SavingsAccountsWriteLock is specified for a whole branch. Logically, however, it locks all savings accounts of the branch in write mode. It has one variables:

typeCheckMethod *< Symbol>* *a method name "*

locking

isCompatibleWith: otherLock

```
↑(super isCompatibleWith: otherLock)
  or: [(self performGuard: self typeCheckMethod with: otherLock account)
    ~= #savings]
```

A.4.5 Method addInterest in Class Branch

class	**Branch**
superclass	HermesObject
instance variables	name
	accounts
class variables	*none*
pool dictionaries	*none*
class category	Distributed Bank

in account operations

```
addInterest
"MessageParameters
    transactionCreating;
    lock: [SavingsAccountsWriteLock typeCheckMethod: #typeOf:]"

  self accounts do: [:account | account type = #savings ifTrue: [account balance:
account balance * (1 + self interestRate)]].
  ↑#done
```

A.5 The Distributed Bank Implementation

A.5.1 The Teller Class

class	**Teller**
superclass	HermesObject
instance variables	name
	currencyTable
	interface
class variables	*none*
pool dictionaries	*none*
class category	Distributed Bank

"Class Teller represents various teller types in the distributed bank. It is defined in the following hierarchy:

HermesObject ()
  Teller (*'name' 'currencyTable' 'interface'*)
    AutomaticTellerMachine ()
    BankClerk ()
    HeadOffice (*'branches' 'tellers'*)

Variables are:

name < *Symbol*> *which uniqely identifies a teller*
currencyTable < *CurrencyTable*> *used fo looking up exchange rates*
interface < *TellerInterface*> *a graphical user interface; not part of the persistent state of a teller object. "*

transfer

internationalTransferFrom: branch1 name: account1 to: branch2 name: account2
"MessageParameters transactionCreating"

```
| currency1 currency2 exactRate amount newAmount |
currency1 := self currencyOf: branch1.
currency2 := self currencyOf: branch2.
exactRate := (self headOffice) waitByNec; exchangeRate: currency1 to: currency2.
amount := branch1 balanceOf: account1.
newAmount := amount * (amount > 10000
        ifTrue: [exactRate]
        ifFalse: [self exchangeRate: currency1 to: currency2]).
branch1 asynchronously; withdraw: amount from: account1.
branch2 asynchronously; deposit: newAmount to: account2.
↑#done
```

transfer: amount from: branch1 name: accountNumber1 to: branch2 name: accountNumber2
"MessageParameters transactionCreating"

```
branch1 asynchronously; withdraw: amount from: accountNumber1.
branch2 asynchronously; deposit: amount to: accountNumber2.
↑#done
```

A.5.2 The `HeadOffice` Class

class	**HeadOffice**
superclass	Teller
instance variables	branches
	tellers
class variables	*none*
pool dictionaries	*none*
class category	Distributed Bank

head office operations

addBranch: branchName on: node
"MessageParameters transactionCreating"

```
| branch |
branch := Branch name: branchName location: node.
self branches add: branch.
↑branch
```

```
addTeller: tellerName on: node
"MessageParameters transactionCreating"

  | teller |
  teller := Teller name: tellerName location: node.
  self tellers add: teller.
  ↑teller


audit
"MessageParameters transactionCreating"

  ↑(self branches collect: [:branch | branch total]) sum


deleteBranch: branch
"MessageParameters transactionCreating"

  branch asynchronously; delete.
  self branches remove: branch.
  ↑#done


deleteTeller: teller
"MessageParameters transactionCreating"

  teller asynchronously; delete.
  self tellers remove: teller.
  ↑#done
```

A.5.3 The `AutomaticTellerMachine` Class

class	**AutomaticTellerMachine**
superclass	Teller
instance variables	*none*
class variables	*none*
pool dictionaries	*none*
class category	Distributed Bank

transfer

```
transfer: amount from: branch1 name: accountNumber1 to: branch2
name: accountNumber2
"MessageParameters timeout: 2"

  ↑super
    transfer: amount
    from: branch1
    name: accountNumber1
    to: branch2
    name: accountNumber2
```

A.5.4 The BankClerk Class

class	**BankClerk**
superclass	Teller
instance variables	*none*
class variables	*none*
pool dictionaries	*none*
class category	Distributed Bank

A.5.5 The Branch Class

class	**Branch**
superclass	HermesObject
instance variables	name
	accounts
class variables	*none*
pool dictionaries	*none*
class category	Distributed Bank

"Class Branch represents a branch of the distributed bank that contains a number of bank accounts. It is defined in the following hierarchy.

HermesObject ()

Branch (*'name' 'accounts'*)

Branch has two variables

'name' < *Symbol*> *that uniqely identifies a branch;*

'accounts' < *Tree*> *a collection of all accounts stored at this branch, sorted according to the account number.* "

account operations

addInterest

```
"MessageParameters
    transactionCreating;
    lock: [SavingsAccountsWriteLock typeCheckMethod: #typeOf:]"

  self accounts do: [:account | account type = #savings ifTrue: [account balance:
account balance * (1 + self interestRate)]].
  ↑#done
```

balanceOf: accountName

```
"MessageParameters
    transactionCreating;
    lock: [AccountReadLock account: accountName]"

  ↑(self lookUp: accountName) balance
```

```
closeAccount: accountName
"MessageParameters
    transactionCreating;
    lock: [AccountWriteLock account: accountName]"

  | account |
  account := self lookUp: accountName.
  account balance ~= 0 ifTrue: [self abortCurrentTransaction: #notEmpty].
  self accounts remove: account.
  account delete.
  ↑#done


deposit: amount to: accountName
"MessageParameters
    transactionCreating;
    lock: [AccountWriteLock account: accountName]"

  | account |
  amount < 0 ifTrue: [self abortCurrentTransaction: #negativeAmount].
  account := self lookUp: accountName.
  account deposit: amount.
  ↑#done


lookUp: accountName

  ↑self accounts detect: [:account | account name = accountName]
    ifNone: [self abortCurrentTransaction: #noSuchAccount]


openAccount: accountName type: accountType
"MessageParameters
    transactionCreating;
    lock: [AccountWriteLock account: accountName]"

  | account |
  account := Account name: accountName type: accountType.
  self accounts add: account ifExisting: [self abortCurrentTransaction: #alreadyExisting].
  ↑account


total
"MessageParameters lock: [AccountReadLock account: #allAccounts]"

  ↑self accounts collect: [:account | account balance] sum


withdraw: amount from: accountName
"MessageParameters transactionCreating;
    lock: [AccountWriteLock account: accountName]"
```

```
| account |
amount < 0 ifTrue: [self abortCurrentTransaction: #negativeAmount].
account := self lookUp: accountName.
account withdraw: amount.
↑#done
```

guard methods

typeOf: accountName

```
↑(self lookUp: accountName) type
```

---

class	**Branch class**
superclass	HermesObject class
instance variables	*none*
class variables	*none*
pool dictionaries	*none*

instance creation

name: branchName location: location

```
↑self
  instantiate: #persistent
  name: branchName
  location: location
  init: [:branch | branch name: branchName; accounts: (Tree instantiate: #persistent)]
```

A.5.6 The Account Class

---

class	**Account**
superclass	HermesObject
instance variables	name
	type
	balance
class variables	*none*
pool dictionaries	*none*
class category	Distributed Bank

"Account represents an individual bank account as stored in branches of the distributed bank. It is defined in the following hierarchy

HermesObject ()

Account (*'name' 'type' 'balance'*)

It has three variables:

'name' < *Symbol*> *a name that uniquely identifies this account within the branch*

'type' < *Symbol*> , *#cheque of #savings*

'balance' < *Integer*> *the current account balance "*

deposit/withdraw

deposit: amount

```
self balance: self balance + amount.
↑#done
```

withdraw: amount

```
self balance: self balance − amount.
↑#done
```

---

class	**Account class**
superclass	HermesObject class
instance variables	*none*
class variables	*none*
pool dictionaries	*none*

instance creation

name: accountName type: type

```
↑self instantiate: #persistent init: [:inst | inst
        name: accountName;
        type: type;
        balance: 0]
```

Bibliography

[ABLN85] Guy T. Almes, Andrew P. Black, Edward D. Lazowska, and Jerre D. Noe. The Eden system: A technical review. *IEEE Transactions on Software Engineering*, SE-11(1):43–58, January 1985.

[Arc91] Architecture Projects Management Limited, Poseidon House, Castle Park, Cambridge CB3 0RD, United Kingdom. *ANSAware 3.0 Implementation Manual, Document RM.097.01*, February 1991.

[Atk91] Colin Atkinson. *Object-Oriented Reuse, Concurrency and Distribution - An ADA-based approach.* ACM Press, New York, 1991.

[BD72] L. A. Bjork and C. T. Davies. The semantics of the preservation and recovery of integrity in a data system. Technical Report TR-02.540, IBM, December 1972.

[BHG87] Philip A. Bernstein, Vassos Hadzilacos, and Nathan Goodman. *Concurrency Control and Recovery in Database Systems.* Series in Computer Science. Addison-Wesley Publishing Company, Reading, Massachusetts, 1987.

[Bjo73] Lawrence A. Bjork. Recovery scenario for a DB/DC system. In *Proc. ACM 73 Nat. Conf.*, pages 142–146, Atlanta GA (USA), August 1973. ACM.

[BN84] A. D. Birrell and B. J. Nelson. Implementing remote procedure calls. *ACM Transactions on Computer Systems*, 2:39–59, 1984.

[Boo90] Grady Booch. *Object Oriented Design with Applications.* Benjamin-Cummings, 1990.

[Car90] Denis Caromel. Concurrency and reusability: From sequential to parallel. *Journal of Object-Oriented Programming*, pages 34–42, September/October 1990.

[CC91] Roger C. Chin and Samuel T. Chanson. Distributed object-based programming systems. *ACM Computing Surveys*, 23(1):91–124, March 1991.

[CCI91] CCITT – International Telegraph and Telephone Consultative Committee. *Draft Recommendation F.851 - Universal Personal Telecommunications Service – Principles and Operational Provisions. Version 5. Question 35/I. Study Group I (28 May - 7 June 1991 Meeting)*, 1991.

[CCM+93] Brian Church, Peter Coleman, Matthew McGregor, Jeff Saul, Ian White, and Paul Woollard. Universal personal telecommunications. User manual and technical manual, The University of Wollongong, Department of Computer Science, Wollongong, NSW, Australia, 1993.

[CY91a] Peter Coad and Edward Yourdon. *Object-Oriented Analysis.* Yourdon Press, New Jersey, 2nd edition, 1991.

[CY91b] Peter Coad and Edward Yourdon. *Object-Oriented Design.* Prentice-Hall, 1991.

[Dav73] Charles T. Davies. Recovery semantics for a DB/DC system. In *Proc. ACM 73 Nat. Conf.*, pages 136–141, Atlanta GA (USA), August 1973. ACM.

[Dix94] Graeme Dixon. Transarc corp. Private communication, 1994.

[DLAR91] Partha Dasgupta, Richard J. LeBlanc, Mustaque Ahamad, and Umakishore Ramachandran. The Clouds distributed operating system. *IEEE Computer*, 24(11):34–44, 1991.

[EGLT76] K. P. Eswaran, J. N Gray, R. A. Lorie, and I. L. Traiger. The notions of consistency and predicate locks in a database system. *Communications of the ACM*, 19(11):624–633, November 1976.

[EME91] Jeffrey L. Eppinger, Lily B. Mummert, and Alfred Z. Spector (Eds.). *Camelot and Avalon.* Morgan Kaufmann Publishers, Inc., San Mateo, CA 94403, 1991.

[Faz94] Michael Fazzolare. Flexible concurrency control through nested encapsulation in Hermes/ST. Submitted to: Seventh International Conference on Parallel and Distributed Computing Systems, Las Vegas, Nevada, USA. October 6–8, 1994.

[FHR93a] Michael Fazzolare, Bernhard G. Humm, and R. David Ranson. Advanced transaction semantics for TINA. In *Proceedings of the Fourth Telecommunications Information Networking Architecture Workshop (TINA 93), Volume 2*, pages 47–57, L'Aquila, Italy, September 27-30 1993.

[FHR93b] Michael Fazzolare, Bernhard G. Humm, and R. David Ranson. Concurrency control for distributed nested transactions in Hermes/ST. In *Proceedings of the 1993 International Conference on Parallel and Distributed Systems (ICPADS'93)*, National Taiwan University, Taipei, Taiwan, Republic of China, December 15-17 1993.

[FHR93c] Michael Fazzolare, Bernhard G. Humm, and R. David Ranson. Hermes/ST user manual and technical manual. Technical Report No. 4, Telecommunications Software Research Centre, Department of Computer Science, University of Wollongong, Wollongong NSW 2500, Australia, 1993.

[FHR94] Michael Fazzolare, Bernhard G. Humm, and R. David Ranson. Object-oriented extendibility in Hermes/ST, a transactional distributed programming environment. In Rachid Guerraoui, Oscar Nierstrasz, and Michel Riveill, editors, *Proceedings of the ECOOP '93 Workshop on Object-Based Distributed Programming, Lecture Notes in Computer Science 791*, pages 240–261. Springer-Verlag, 1994.

[For] Forte Software Inc., Oakland CA, USA. *FORTE.*

[GCLR92] Rachid Guerraoui, Riccardo Capobianchi, Agnes Lanusse, and Pierre Roux. Nesting actions through asynchronous message passing: the ACS protocol. In *Proceedings, European Conference on Object-Oriented Programming (ECOOP'92)*, 1992.

[Gif79] David K. Gifford. Weighted voting for replicated data. In *Proceedings of the 7th Symposium on Operating Systems Principles*, pages 150–162. ACM, 1979.

[GR89] Adele Goldberg and Dan Robson. *Smalltalk-80: The Language.* Addison-Wesley, 1989.

[GR93] Jim Gray and Andreas Reuter. *Transaction Processing: Concepts and Techniques.* Morgan Kaufmann, USA, 1993. ISBN 1-55860-190-2.

[Hal85] R. H. Halstead. Multilisp: A language for concurrent symbolic computation. *Transactions on Programming Languages and Systems*, October 1985.

[Hew91] Carl Hewitt. Open information systems semantics for distributed artificial intelligence. *Artificial Intelligence*, 47:79–106, 1991.

[HF92a] Bernhard G. Humm and Michael Fazzolare. Object-Oriented Analysis and Design for Universal Personal Telecommunications. In *Proceedings of the Second IASTED International Conference on Computer Applications in Industry*, Alexandria, Egypt, May 1992.

[HF92b] Bernhard G. Humm and Michael Fazzolare. Object-Oriented Analysis for Telecommunications Services. In *Proceedings of the 1992 International ACM/SIGAPP Symposium on Applied Computing*, Kansas City MO, USA, March 1992.

[HKM+94] Nicholas Haines, Darrell Kindred, J. Gregory Morrisett, Scott M. Nettles, and Jeannette M. Wing. Tinkertoy transactions. Technical report, Computer Science Department, Carnegie Mellon University, Pittsburgh, PA 15213-3891, 1994. Submitted to Conference on Lisp and Functional Programming '94.

[HR93] Theo Haerder and Kurt Rothermel. Concurrency control issues in nested transactions. *VLDB Journal*, 2(1):39–74, 1993.

[HS91] Brian Henderson-Sellers. *A Book of Object-Oriented Knowledge.* Prentice Hall, 1991.

[Hum93] Bernhard G. Humm. An extended scheduling mechanism for nested transactions. In Luis-Felipe Cabrera and Norman Hutchinson, editors, *Proceedings of the Third International Workshop on Object-Orientation in Operating Systems (IWOOOS'93)*, pages 125–134, Asheville, North Carolina, USA, December 1993. IEEE, IEEE Computer Society Press, Los Alamitos, CA.

[Knu73] Donald E. Knuth. *The Art of Computer Programming*, volume 1: Fundamental Algorithms. Addison Wesley Publishing Company, Reading Massachusetts, USA, second edition, 1973.

[LCJS87] Barbara Liskov, Dorothy Curtis, Paul Johnson, and Robert Scheifler. Implementation of Argus. *ACM Operating Systems Reviews*, 21(5):111–122, 1987.

[Lie87] H. Liebermann. Concurrent object-oriented programming in Act1. In Akinori Yonezawa and Mario Tokoro, editors, *Object-Oriented Concurrent Programming*, chapter 2, pages 9–36. The MIT Press, Cambridge MA, USA and London, England, 1987.

[Lis81] Barbara Liskov. CLU reference manual. In Goos and Hartmanis, editors, *Lecture Notes in Computer Science*. Springer Verlag, Berlin, 1981.

[Lis82] Barbara Liskov. On linguistic support for distributed programs. *IEEE Transactions on Software Engineering*, SE-8(3):203–210, May 1982.

[Lis84] Barbara Liskov. Overview of the Argus language and system. Programming methodology group memo 40, MIT Laboratory for Computer Science, February 1984.

[Lis88] Barbara Liskov. Distributed programming in Argus. *Communications of the ACM*, 31(3):300–312, March 1988.

[LS83] Barbara Liskov and Robert Scheifler. Guardians and actions: Linguistic support for robust, distributed programs. *ACM Transactions on Programming Languages and Systems*, 5(3):381–404, July 1983.

[LSP82] L. Lamport, R. Shostak, and M. Pease. The byzantine generals problem. *ACM Transactions on Programming Languages and Systems*, 4(3):382–401, 1982.

[Mey88] Bertrand Meyer. *Object-Oriented Software Construction*. Prentice Hall, Cambridge, Great Britain, 1988.

[MMP83] Erik. T. Mueller, Johanna D. Moore, and Gerald J. Popek. A nested transaction mechanism for LOCUS. In *Proceedings of the 9th Symposium on Operating Systems Principles*, pages 71–83. ACM/SIGOPS, ACM Press, 1983.

[Mos81] J. Eliot B. Moss. *Nested Transactions: An Approach to Reliable Distributed Computing*. PhD thesis, M.I.T. Department of Electrical Engineering and Computer Science, April 1981. Available as M.I.T. Laboratory for Computer Science Technical Report 260. Out of print.

[Mos85] J. Eliot B. Moss. *Nested Transactions – An Approach to Reliable Distributed Computing*. MIT Press Series in Information Systems. The MIT Press, Cambridge, Massachusetts and London, England, 1985.

[MTH90] R. Milner, M. Tofte, and R. Harper. *The Definition of Standard ML*. The MIT Press, 1990.

[NW91] Scott M. Nettles and Jeannette M. Wing. Persistence + undoability = transactions. Technical Report CMU-CS-91-173, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA 15213, 1991.

[Obj] Object Design, Burlington MA, USA. *ObjectStore*.

[Par92] ParcPlace Systems, Sunnyvale CA, USA. *VisualWorks Release 1.0*, 1992.

[Pen] Penobscot Development Corp., Cambridge Mass. USA. *KALA*.

[PN85] Calton Pu and Jerre D. Noe. Nested transactions for general objects: The Eden implementation. Technical Report TR-85-12-03, Department of Computer Science, University of Washington, Seattle, WA 98195, December 1985.

[Ran94] R. David Ranson. A framework for transactional semantics. Submitted to: Distribution and Concurrency in Persistent Systems, HICSS Minitrack, Maui, Hawaii, January 1994.

[RBP+91] James Rumbaugh, Michael Blaha, William Premerlani, Frederick Eddy, and William Lorensen. *Object-Oriented Modelling and Design.* International Editions. Prentice Hall, New York, 1991.

[Ree78] David P. Reed. *Naming and Synchronization in a Decentralized Computer System.* PhD thesis, M.I.T. Department of Electrical Engineering and Computer Science, September 1978. Available as M.I.T. Laboratory for Computer Science Technical Report 205.

[RHR+93] Matthew Robinson, Linda Hennessey, Shane Richards, Adam Barclay, and Fiona Soper. A reliable distributed name server. User manual and technical manual, The University of Wollongong, Department of Computer Science, Wollongong, NSW, Australia, 1993.

[RSL87] D. J. Rosenkrantz, R. E. Stearns, and P. M. Lewis. System level concurrency control for distributed database systems. *ACM Transactions on Database Systems*, 3(2), June 1987.

[SBD+84] Alfred Z. Spector, Jacob Butcher, Dean S. Daniels, Daniel J. Duchamp, Jeffrey L. Eppinger, Charles E. Fineman, Abdelsalam Heddaya, and Peter M. Schwarz. Support for distributed transactions in the TABS prototype. In *Proceedings of the 4th Symposium in Distributed Software and Database Systems*, pages 186–206, Bethseda, Maryland (USA), 1984. IEEE Computer Society, IEEE Computer Society Press.

[Sch93] Fred B. Schneider. What good are models and what models are good? In Sape Mullender, editor, *Distributed Systems, Second Edition*, chapter 2, pages 17–26. ACM Press, New York, 1993.

[SDP91] S. K. Shrivastava, G. N. Dixon, and G. D. Parrington. An overview of the Arjuna distributed programming system. *IEEE Software*, pages 66–73, 1991.

[Ske82] D. Skeen. *Crash Recovery in a Distributed Database System.* PhD thesis, University of California, Berkley, 1982.

[Str86] Brian Stroustrup. *The C++ Programming Language.* Addison-Wesley, 1986.

[Tra91] Transarc Corporation, Pittsburgh, PA, USA. *Encina Transaction Processing System, TP Monitor, TP-00-D078*, 1991.

[Ver] Versant Object Technology, 4500 Bohannon Drive, Menlo Park, CA 94025, USA. *Versant.*

[WBWW90] Rebecca Wirfs-Brock, B. Wilkerson, and L. Wiener. *Designing Object-Oriented Software.* Prentice Hall, 1990.

[WFMN92] Jeannette M. Wing, Manuel Faehndrich, J. Gregory Morrisett, and Scott Nettles. Extensions to standard ml to support transactions. Technical Report CMU-CS-92-132, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA 15213, 1992.

[WL85] William Weihl and Barbara Liskov. Implementation of resilient, atomic data types. *ACM Transactions on Programming Languages and Systems*, 7(2):244–269, April 1985.

[WN93] Xuequn Wu and Jan Neuhaus. Extending PCTE with object-oriented capabilities. In *Proceedings of DEXA93*, pages 681–684, 1993.

[YSTH87] A. Yonezawa, E. Shibayama, T. Takada, and Y. Honda. Modelling and programming in an object-oriented concurrent language ABCL/1. In Akinori Yonezawa and Mario Tokoro, editors, *Object-Oriented Concurrent Programming*, chapter 4, pages 55–90. The MIT Press, Cambridge MA, USA and London, England, 1987.

[YT87] Yasuhiko Yokote and Mario Tokoro. Concurrent programming in ConcurrentSmalltalk. In Akinori Yonezawa and Mario Tokoro, editors, *Object-Oriented Concurrent Programming*, chapter 6, pages 129–158. The MIT Press, Cambridge MA, USA and London, England, 1987.

[YTM+91] Yashuiko Yokote, Fumio Teraoka, Atsushi Mitsuzawa, Nobuhisa Fjinami, and Mario Tokoro. The Muse object architecture: A new operating system structuring concept. *Operating Systems Review*, 25(2), 1991.

Index

2PC, 10
    coordinator, 10
    participant, 10
    commit phase, 11
    prepare phase, 10
2PL, 8

abort recovery, 9, 10
abstract class, 15
abstract data type, 14
ancestor, 39
ancestor class, 14
ancestor transaction, 12
any, 40
Argus, 18, 34, 107
    atomic object, 18, 34
    guardian, 18, 34
    handler, 18
    non-atomic object, 18, 34
asynchronous message, 24, 38
asynchronous with respect to, 43
atomic object in Argus, 18, 34
atomicity, 5
attempted voucher redeem, 93
Avalon/C++, 35
    server, 35
awaited object, 24

behaviour sharing, 16
belongs to top-level transaction, 41
belongs to transaction, 41
binary search tree, 22
broadcast, 82
Byzantine failure, 5

C++, 16
caching, 74
Camelot, 35
cascading abort, 7, 47
catastrophe, 9, 11
CBox, 24
checkpointing, 12
child transaction, 12
class, 14
code sharing, 16
commit phase of 2PC, 11
common ancestor, 39
composability of concurrency control, 31
concurrency control, 6, 28, 79
    multiple-version, 8
    optimistic, 7
    pessimistic, 7
    single-version, 8
concurrency controller, 7
conflicting, 39
constant object in Hermes/ST, 21
crash recovery, 9, 10

deadlock, 8
    avoidance, 9
    detection, 9
    prevention, 9
descendant, 39
descendant transaction, 12
descendent class, 14
distributed bank, 20
distributed system, 3
downgrading lock mode, 113
downward lock inheritance, 113
dynamic return dependency, 93

early writing, 12
Eden, 111
Eiffel, 16
encapsulation, 14
explicit concurrency control, 29
extendibility, 16

failure model, 5
fine-grained object model, 21
finish of execution of message, 44
future variable, 24

generalized message scheme, 23
guard method, 30
guardian in Argus, 18, 34

handler in Argus, 18
Hermes/ST, 20
  class, 21
  concurrency control, 28
    explicit, 29
    implicit, 28
    minimal locking, 28
    programmable lock approach, 29
  instance creation parameter, 21
  message
    asynchronous, 24
    generalized message scheme, 23
    kind, 24
    parameter, 25
    synchronous, 23
    wait-by-necessity, 24
  method
    access method, 21
    guard method, 30
    arguments, 24
    name, 24
  object, 21
    constant, 21
    HermesObject, 21
    kind, 21
    persistent, 21
    receiver, 24
    volatile, 21
  retries, 26
  timeout, 26
  transaction mode, 26
  variable
    indexed, 21
    named, 21
holding a lock, 8, 100, 112

implicit concurrency control, 28
implicit future, 24, 117
incomparable, 39
inconsistent state, 4
information hiding, 14
inheritance, 14
  multi-rooted, 14
  multiple, 14
  non-strict, 14
  single, 14
  single-rooted, 14
  strict, 14
instance, 14
interleaving operations, 3

is-a relationship, 15

KAROS, 117

lazy information propagation, 74
least common ancestor message, 39
least common ancestor transaction, 41
location transparency, 18
lock
  anti-inheritance, 13
  compatibility matrix, 8
  controlled downward inheritance, 113
  downward inheritance, 113
  holding, 8
  inheritance, 13
  type, 38
  releasing, 8
  upward inheritance 13
locking
  mutex, 8
  read/write, 8
  two-phase, 8
  type-specific, 8
logging
  redo, 10
  undo, 10
long message path, 84

maintainability, 16
message, 14, 38
  argument, 38
  belongs to thread, 42
  finish execution, 44
  identifier, 38
  kind, 38
  parameter, 38
  passing, 14
  path, 40, 84
    long, 84
    short, 84
  path element, 40
  return, 44
  start execution, 44
  tree, 38
message-submessage relationship, 38
method, 14
migration, 18
minimal locking, 28
Moss, J. Eliot B., 13, 51, 100
multi-rooted inheritance, 14
multiple inheritance, 14

multiple-version concurrency control, 8
mutex locking, 8
mutual exclusion locking, 8

nested encapsulation, 21
nested transaction, 12
network, 3
network failure, 4
node, 3
node crash, 4, 5
non-atomic object in Argus, 18, 34
non-strict inheritance, 14
non-transaction creating, 38
non-transactional, 41

object, 14
object-based, 16
object-orientation, 13
offering a lock, 113
one level below least common ancestor, 41
optimistic concurrency control, 7
optional element, 40
overriding, 14

parameterization, 20
parent transaction, 12
parent-child relationship, 39
partial thread, 43
permanence, 5
permanent memory, 5
pessimistic concurrency control, 7
polymorphism, 15
prepare phase of 2PC, 10
private method, 14
programmable lock, 29
programmable lock approach, 29
public method, 14

read/write locking, 8
real ancestor, 39
real ancestor transaction, 12
real descendant, 39
real descendant transaction, 12
receiver object, 24, 38
recovery, 6, 9
    abort, 9, 10
    crash, 9, 10
redo logging, 10
releasing a lock, 8
remote procedure call, 17
replication, 18
retaining a lock, 112
retDep, 53
return dependency, 47
    dynamic, 93
reusability, 16

schedulability predicate, 53
schedulable, 53
schedulable with respect to, 54
schedule
    serial, 6
    serializable, 6
scheduled serially, 44
scheduling, 80
serial schedule, 6, 44
serializability, 5
serializable schedule, 6, 45
serialization graph testing, 8
serialized, 45
server in Avalon/C++, 35
short message path, 84
single inheritance, 14
single-rooted inheritance, 14
single-version concurrency control, 8
Smalltalk-80, 16
start of execution of message, 44
status record, 10
strict inheritance, 14
subclass-superclass relationship, 14
subtransaction, 12
    abort, 12
    commit, 12
successful voucher redeem, 93
synch, 40
synchronized, 46
synchronized schedule, 46
synchronous message, 23, 38
synchronous with respect to, 42

thread, 42
    identifier, 38
    of a message, 42
    under transaction, 43
three-phase commit, 11
timestamp ordering, 8
top-level abort, 12
top-level commit, 12
top-level message, 39
top-level transaction, 12
top-level transaction of a message, 41

trans, 40
transaction, 5
    abort, 5
    characteristics, 38
    commit, 5
    creating, 38
    identifier, 38
    of a message, 41
    one level below least common ancestor, 41
    tree, 12, 41
transactional, 41
two-phase commit, 10
two-phase locking, 8
type-specific locking, 8

undo logging, 10
uniform reference semantics, 21
Universal Personal Telecommunications, 33
unsuccessful voucher redeem, 93
update record, 10
upgrading lock mode, 113
upward lock inheritance, 13, 101

variable, 14
Venari/ML, 36, 115
volatile memory, 5
voucher, 24
    redeem, 93
      successful, 93
      unsuccessful, 93

wait-by-necessity, 24, 92
waits-for graph, 9