1994

The granularity of concurrency control in distributed object-oriented systems supporting transactions

Michael Fazzolare

University of Wollongong

Recommended Citation

NOTE

This online version of the thesis may have different page formatting and pagination from the paper copy held in the University of Wollongong Library.

UNIVERSITY OF WOLLONGONG

COPYRIGHT WARNING

You may print or download ONE copy of this document for the purpose of your own research or study. The University does not authorise you to copy, communicate or otherwise make available electronically to any other person any copyright material contained on this site. You are reminded of the following:

Copyright owners are entitled to take legal action against persons who infringe their copyright. A reproduction of material that is protected by copyright may be a copyright infringement. A court may impose penalties and award damages in relation to offences and infringements relating to copyright material. Higher penalties may apply, and higher damages may be awarded, for offences and infringements involving the conversion of material into digital or electronic form.
The Granularity of Concurrency Control in Distributed Object-Oriented Systems Supporting Transactions

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

Michael Fazzolare, Bachelor of Mathematics, Honours Class 1 (Wollongong)

Department of Computer Science
1994
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 identical work either prior to this thesis or currently being pursued.

Michael Fazzolare
Abstract

This thesis is concerned with the granularity of concurrency control in distributed object-oriented systems that support nested transactions. Novel linguistic constructs are introduced that allow the specification of object structures that support different granularities of concurrency control. The so-called "multi-granular concurrency control" has static and dynamic variations. Static multi-granular concurrency control allows an application developer to instantiate the same object topology with different numbers of concurrency controllers. Dynamic multi-granular concurrency control allows an application developer to vary the number of concurrency controllers used by an instantiated object topology. Multi-granular concurrency control is introduced in such a way that the serialisability of potentially nested transactions is maintained.

The mechanisms presented in this thesis have a number of advantages over existing concurrency control approaches. The separation of concurrency control specification from class specification allows flexibility during system development and potentially more efficiency during system execution. Applications developers can fine-tune the performance of their applications without necessarily having to change the structure or semantics of the code. Typical features of object-orientation such as reusability, incremental development and ease of specification are supported.
Acknowledgements

I would like to thank a number of people and institutions who have been helpful throughout the time that I have worked on this thesis. I would like to thank my supervisor, associate professor Neil. A. B. Gray, for providing support and guidance during the thesis. To the management and staff of the Telecommunications Software Research Centre (TSRC) thanks for providing me with an excellent working environment. Thanks also go to the University of Wollongong and Telecom Australia for their financial support. To professor Fergus O’Brien, thanks for the scholarship and the thesis advice. To Dr Marilyn Cross, thanks for managing the centre.

Special thanks go to B. G. Humm and R. David Ranson. Without their constant support, guidance and willingness to be involved in technical discussions this thesis would not have been the same. We all learned a lot together.

Thanks also to all of the above mentioned people for reading and commenting on various drafts of this thesis.

To John and Mary Wheatley. That silly birthday card started this whole process. In the future, please forget my birthday. This thesis is dedicated to my family.
# Contents

List of Figures vi

List of Tables vii

1 Introduction 1

2 Transactions and Objects in Distributed Systems 4
   2.1 Issues in Distributed Systems 4
      2.1.1 Parallelism 5
      2.1.2 Independent Node Failures 6
      2.1.3 Independent Communication Link Failures 7
      2.1.4 Network Communication 8
   2.2 Programming Distributed Systems 8
      2.2.1 Persistent Data in Distributed Systems 9
   2.3 Transactions in Distributed Systems 9
      2.3.1 Concurrency Control 10
      2.3.2 Recovery 16
   2.4 Nested Transactions 18
      2.4.1 Concurrency Control 19
      2.4.2 State Restoration 19
      2.4.3 Persistence Handling 19
   2.5 Object-Orientation in Distributed Systems 20
      2.5.1 Object-Orientation 20
      2.5.2 Flexible Sharing 21
      2.5.3 Advantages of Object-Orientation 22
      2.5.4 Distributed Systems that Support Objects 22
   2.6 Distributed Systems Supporting Transactions and Objects 23
      2.6.1 Object-Oriented Databases 24
   2.7 The Distributed Bank Example 25
   2.8 Domains of Applicability of Distributed Programming Environments 25

3 Hermes/ST and Granularity 27
   3.1 Hermes/ST Distributed Object Model 27
      3.1.1 Ontology 27
      3.1.2 Hermes/ST Object Kinds 29
      3.1.3 Constant Objects 29
      3.1.4 Hermes/ST Object Creation 30
      3.1.5 Hermes/ST Message Passing 32
      3.1.6 Message Parameter Specification 39
      3.1.7 Extra Transaction Operations 40

iv
CONTENTS

5.3.2 The Static Navigation Protocol - Revisited ........................................ 84
5.3.3 Triggers ................................................................. 84
5.3.4 Downward Concurrency Controller Movement ................................. 85
5.3.5 Upward Concurrency Controller Movement .................................... 88
5.3.6 The Dynamic Navigation Protocol ................................................. 90
5.3.7 Deciding Whether to Statically or Dynamically Reroute ................. 91

6 Discussion ................................................................. 93
6.1 Introduction ............................................................................. 93
6.2 Evaluation of Multi-Granular Concurrency Control ......................... 93
  6.2.1 Performance .................................................................. 93
  6.2.2 Incremental Development ............................................... 102
  6.2.3 Reuse .......................................................................... 103
  6.2.4 Ease of Specification .................................................... 103
6.3 Comparison with Other Systems .................................................. 105
  6.3.1 Dynamic Multi-Granular Concurrency Control ....................... 106
  6.3.2 Static Multi-Granular Concurrency Control ............................ 111
  6.3.3 Minimal Locking .......................................................... 111
  6.3.4 Programmable Locking ................................................... 112
  6.3.5 Commercial Relational Database Systems ............................. 113
  6.3.6 Commercial Object-Oriented Databases ............................... 114
  6.3.7 Nested Encapsulation ..................................................... 115

7 Conclusions and Future Research ................................................. 117
7.1 Conclusions .......................................................................... 117
  7.1.1 Hermes/ST Concurrency Control ...................................... 117
  7.1.2 Object Orientation ......................................................... 119
  7.1.3 Limitations of Multi-Granular Concurrency Control ............... 119
7.2 Future Research ...................................................................... 120
  7.2.1 Extending Decoupling ..................................................... 120
  7.2.2 Extensions to Mechanisms of this Thesis ............................. 121

Bibliography .............................................................................. 122
List of Figures

2.1 Standard Read/Write Locking Compatibility Matrix ........................ 13

3.1 An example message tree .................................................. 33
3.2 An example message tree containing a transaction ....................... 34
3.3 An example message tree containing a nested transaction ............... 35
3.4 An example message tree containing a thread ........................... 36
3.5 An example message tree containing a nested thread ................. 37

4.1 Standard Hermes/ST Locks - Compatibility Matrix ..................... 48
4.2 Incorrect Relationship between State Restoration and Concurrency Control Granularity ................................. 57
4.3 Incorrect Relationship between Persistence and Concurrency Control Granularity ................ 60
4.4 An example instantiation of a branch and its binary account tree ... 65

5.1 An example of an encapsulation hierarchy. Circles represent Hermes/ST objects. Triangles represent encapsulation boundaries. The shaded part of a triangle represents an object’s public interface. Objects $O_1$, $O_2$ and $O_3$ are top-level .................................................. 74
5.2 An example of an asynchronous communication that violates the noCC schedulability objective. $O_{1.1}$ performs two asynchronous method invocations on $O_{1.2}$, which has no concurrency controller defined .... 76
5.3 An example encapsulation hierarchy showing the various styles of (static) multi-granular concurrency control that Hermes/ST supports by specifying that either highest and lowest markers be positioned together, or by their omission .................................................. 81
5.4 An example of dynamic multi-granular concurrency control showing the highest marker at the top-level Hermes/ST object and no lowest marker .. 82
5.5 An example encapsulation hierarchy showing the most general form of dynamic concurrency control granularity that Hermes/ST supports ............ 83
5.6 “Before” and “after” states of an encapsulation hierarchy that has moved its concurrency controllers down one level .................................................. 85
5.7 Before and after states of an encapsulation hierarchy that has moved its concurrency controllers up .................................................. 88

6.1 Throughput versus Multi-Programming Level (MPL) - No idle time .... 97
6.2 Throughput versus Multi-Programming Level (MPL) - 1 second idle time 98
6.3 Throughput versus Multi-Programming Level (MPL) - 2 second idle time 98
6.4 Throughput versus Multi-Programming Level (MPL) - 5 second idle time 98
List of Tables

2.1 An example of incorrect interleaving of deposit operations ............... 6
2.2 Cascading Abort Example. ........................................... 12
2.3 Deadlock Example. .................................................... 14
Chapter 1

Introduction

This thesis is about concurrency control in object-oriented distributed systems that support nested transactions. Within the last decade, distributed systems have become increasingly important [BST89, CC91, Mul89, Mul93]. Programming distributed systems is inherently more complex than programming single-node, sequential systems. This is due to the extended requirements that distributed applications often demand [Mul93]. One such requirement is that distributed applications maintain reliable [GR93, BHG87, Mul89, Mul93] data. The investigation of abstractions that assist in the construction of reliable distributed applications is therefore an important component of distributed systems research. One convenient abstraction for reliable distributed computing is that of transactions [GR93, BHG87]. Transactions were originally developed in the database area and have since been extended to distributed and nested transactions [GR93, Mos85]. Transactions and nested transactions have gained wide acceptance as a key technology in the development of reliable distributed systems.

Object-orientation (e.g. [Mey88]) is a programming paradigm that was originally developed in the simulation area. Object-orientation is also a form of abstraction that deals with the complexity of programming systems in general. Its has advantages in terms of rapid prototyping, reusability, extensibility, and maintainability. Object-orientation has been adopted by many computer science communities, including the distributed systems community.

The integration of both technologies, object-orientation and nested transactions, results in distributed programming environments that support transactions over objects. A prominent early example of such a system is the Argus project [Lis82] (Massachusetts Institute of Technology). The Argus project demonstrated a successful union of transactions and objects. Currently, the integration of distribution and object-orientation is being performed on large-scale commercial distributed systems. Several such systems are currently emerging. Two prominent examples are the Advanced Network Systems Architecture (ANSA) [Arc91] and the Open Software Foundation's Distributed Computing Environment (DCE) [Lib92, Shi92].

Transactions, while a convenient abstraction for programming distributed systems, can suffer from performance problems. One important efficiency related factor of transactions is the granularity at which objects are locked. The relationship between the granularity of concurrency control and system performance is a complex mix of many factors [GR93, BHG87]. Importantly, the granularity of concurrency control that gives optimum performance changes according to the object invocation patterns of the application [GR93, BHG87]. Existing systems encourage application developers to explicitly code a fixed granularity of concurrency control into their applications. Such an approach to concurrency control specification has two main drawbacks:
• An application cannot change the level of concurrency control according to changing system conditions.

• "hard coded" concurrency control specifications hinder reusability, extensibility, and maintainability and are mostly not necessary.

This thesis argues that no single granularity of concurrency control is optimum for any one application in all circumstances. The multi-granular concurrency control introduced by this thesis allows applications to vary the granularity of concurrency control employed so that applications can achieve improved performance. This thesis also argues that by changing the specification style to be more in line with object-oriented principles, reusable and extensible concurrency control specifications can be produced. Finally, this thesis argues that the explicit specification of concurrency control by applications can often be avoided, while still maintaining control over the aspects of an application's performance that are related to the granularity of concurrency control.

The novel aspects of this thesis can be summarised as follows.

• New types of concurrency control are presented. Implicit concurrency control provides the correct synchronisation for many applications. Implicit concurrency control is automatically generated from an application's definition. Explicit concurrency control provides a means for an application developer to describe more complex synchronisation relationships when and if they are required. Furthermore, explicit concurrency control can be developed in line with object-oriented principles.

• Static multi-granular concurrency control allows the development of explicitly and implicitly concurrency controlled applications. Importantly, the granularity of concurrency control of these applications is orthogonal to the specification of the application. This means that the same application can be instantiated with a variety of concurrency control granularities at different times.

• Dynamic multi-granular concurrency control allows the development of implicitly concurrency controlled applications. Importantly, the granularity of concurrency control of these applications is orthogonal to the installation of the application. This means that the same application can change the granularity of concurrency control it employs during its execution. Changing the granularity of concurrency control allows applications to improve their overall performance.

The remainder of this thesis is structured as follows. Chapter 2 gives an overview of issues in transactional object-oriented distributed systems and introduces the example distributed bank application that is used for demonstration purposes throughout this thesis. Chapter 3 introduces the relevant constructs of Hermes/ST [FHR94, Faz94, Ran95, Hum93, FHR93c, FHR93a, FHR93b]. Hermes/ST\(^1\) is an object-oriented distributed programming environment that the author, B.G. Humm and David Ranson have developed and implemented in Smalltalk/80 [GR89].

Chapter 4 introduces implicit and explicit locking. Chapter 5 extends the usefulness of implicit and explicit locking via static multi-granular concurrency control. In addition, dynamic multi-granular concurrency control further extends the utility of implicit concurrency control. Chapter 6 gives some initial experimental results for multi-granular concurrency control and compares the Hermes/ST approach to concurrency control with

\(^1\)Hermes/ST should not be confused with IBM's Hermes system [SBC+91]. The /ST, representing the implementation language Smalltalk, has been appended to avoid confusion.
other distributed systems. Chapter 7 briefly summarises the conclusions of this thesis and outlines areas of continuing research.
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 introduces distributed computer systems. Section 2.2 introduces distributed programming environments. Section 2.3 introduces transactions. Transactions are an abstraction that aid in the development of distributed systems. Nested transactions generalise aspects of transactions and are presented in Section 2.4. Section 2.5 introduces object-orientation and describes how it is applicable to distributed systems. Section 2.6 introduces systems that incorporate both transactions and objects. Section 2.7 introduces the distributed bank example that is used as the primary example for this thesis. Finally, section 2.8 places transactions in perspective for the telecommunications area.

The fields of distributed systems and transactions are both large. Therefore, only as much of both fields is introduced as is needed to set the scene for later chapters of this thesis. To this end, concepts and terminology are introduced informally. Because this thesis is primarily concerned with the granularity of concurrency control, concurrency control is introduced in more detail than other components of transactions. A working knowledge of the fields of transactions and object-orientation as not a pre-requisite to this chapter. However, such knowledge would be advantageous to a reader.

2.1 Issues in Distributed Systems

A distributed computer system can be characterised as a set of multiple autonomous processors that do not share primary memory but cooperate by sending messages over a communications network [BST89]. A node in a distributed computer system consists of one or more processors, local memory, possibly some stable storage such as one or more disk(s), and I/O ports to connect it with the environment. Nodes communicate via communication links that interconnect some of their I/O ports. Nodes and communication links form a graph topology that is referred to as a network. A sender node communicates with a receiver node in the network by sending a network message over some number of communication links in the network.

A distributed system is a collection of applications that execute over a distributed computer system. An application is a set of processes to be executed by one or more processors. A process is a sequence of instructions to be executed on a single node. Applications communicate information by sending messages over the network.

A distributed system must deal with issues that may or may not arise in a single-
node or local computer system. These issues include: parallelism; independent failure; information sharing; naming; security; and heterogeneity. Different distributed systems place different emphasis on the importance of different issues [CC91, BST89, Lim91, Mul93]. The issues relevant to this thesis are discussed below.

2.1.1 Parallelism

The existence of multiple processors or asynchronous devices in a distributed computing system introduces the possibility of parallelism. Multiple processors allow separate processes to execute at the same time. Asynchronous disks and I/O ports allow the same type of behaviour. For example, if one process is writing a character to an I/O port the processor is free to perform some other process. The existence of suitable operating systems introduces the possibility of pseudo parallelism [BST89] on a node. Pseudo parallelism is an artificial execution order imposed on processes by an operating system. For example, a pre-emptive scheduling mechanism [Tan87] executes each process as a sequence of subsets of the instructions that comprise the process. These subsets of instructions are apportioned according to a time slice. Pseudo parallelism introduces context switching which increases the execution cost of each process but improves the response time and fairness aspects of a system. More importantly, pseudo parallelism introduces the possibility that processes can be interrupted between the execution of any pair of instructions in their instruction sequences.

2.1.1.1 Interleaving Operations

Without appropriate concurrency control, parallel and pseudo parallel processes may interleave in such a way that leads to incorrect outcomes. Consider the following example from the banking domain adapted from [Wei93b, BHG87, Hum94]. A deposit of money to an account entails, amongst other activities, adding an amount of money to the current balance of an account. The following sequence of instructions performs the relevant part of the deposit operation. For an account account with a balance balance, a temporary variable tmp and an amount amount

1. Read the balance of the account into tmp
2. Compute the value of tmp plus amount in tmp
3. Update the account's balance to contain the value in tmp

Now consider the following scenarios. The initial balance of an account is $1,000. Two deposit processes are initiated. The first process adds $10,000 and the second process adds $100 to the account. If both processes are executed sequentially then 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. incorrect, outcome is possible if both deposits are executed in parallel. Without some form of concurrency control two parallel deposit operations can interleave as shown in table 2.1.

In table 2.1 the account balance after both deposit operations have finished is $1,100. This is obtained from the last value computed into tmp at event number 4. The wrong outcome is due to the update of the balance by deposit process #1 being over-written by deposit process #2. Some sort of synchronisation or concurrency control is needed to prevent such race conditions [Wei93b] from occurring. Concurrency control must ensure that the three instructions of deposit operation #1 are performed either before or after the three instructions of deposit operation #2.
2.1.2 Independent Node Failures

There are several failure models that can be assumed for nodes in a distributed computer system [Sch93]. These include:

Failstop: Failstop or failfast [GR93] assumes that a node fails by halting. Once, halted the node stays in that state until it is restarted. Furthermore the fact that the node has halted is detectable by other nodes.

Crash: Crash is the same as failstop, except that other nodes may not be able to detect that a node has failed.

Byzantine Failures: The byzantine failure model dictates that nodes can exhibit arbitrary behaviour. Nodes can fail in a manner that is outside any expected failure model. For example, if a byzantine failure model is expected, it is possible that a node may incorrectly order a sequence of instructions and such an incorrect execution is not detectable.

This thesis assumes the crash model. At any point in time a node is either running or it is crashed. Other nodes in the network can determine the state of a node by sending the node a network message. If the message is acknowledged then the node is running. If the message is not acknowledged for some reason then the sending node assumes\(^1\) that the node is crashed. In a distributed computer system, some nodes can be running while other nodes are crashed. The ability of a distributed system to have some nodes running while others are crashed is referred to as partial failure [Sch93, GR93, BHG87, BST89].

Partial failure introduces a special form of inconsistent state into distributed systems. Inconsistent state can be defined as a state introduced when an application does not satisfy domain-specific constraints regarding its data. As an example, consider the following situation that is again taken from the banking domain. One desirable constraint for banks is that accounts always reflect the correct amount of money for each account holder. No moneys should be lost. Each account balance should represent the total of all deposits and withdrawals on that account. Therefore, a transfer of moneys from one account to another account should not change the total amount of money in both accounts. Such applications are said to have a high data integrity requirement.

If accounts are stored on different nodes of a distributed computer system then the implementation of a transfer operation must take into account the possibility of partial failures. For example, consider a transfer between two accounts (account1 and account2) that exist on separate nodes (node1 and node2). To transfer moneys from account1 to account2 then both the balance of account1 must be debited by the transfer amount, and account2 must be credited by the transfer amount.

\(^1\)The node may be crashed or a communication link may be lost. Which has happened is not determinable, hence the word “assumes”. Communication link failure is discussed in section 2.1.3 below.
the balance of account2 must be credited by the transfer amount. If either part of the transfer operation cannot be accomplished because either node1 or node2 is crashed then the whole transfer must be aborted. It should be noted that inconsistent state due to partial failure is inherent in distributed computer systems2.

2.1.3 Independent Communication Link Failures

The nodes and communication links of a distributed computer system form an arbitrary network graph. Typically a network is not completely connected. That is, there is not always a single communication link from one node to every other node in the network. However, a network is connected. That is, there exists a sequence of communication links (a network path) from one node to every other node in the network. A communications link is either running or crashed. A running communication link is referred to as being up, while a crashed communications link is referred to as being down. The same terminology is applied to network paths. The communication links of a distributed computer system can also fail independently. Furthermore, the network medium is not always reliable. These factors result in several possible erroneous fates for a network message. These include the following:

Undelivered Messages: A message can be undelivered for many reasons. The receiver node may be crashed, the message may be repeatedly corrupted or all network paths to the receiver node may be down. When all network paths to a node or set of nodes are down then the network is said to be partitioned [HT93, BHG87].

Corrupt Messages: Networks are not reliable. A network message can be corrupted by external influences on the network medium. The binary digits that represent information in a message can be arbitrarily converted from a zero to a one, or missed entirely. Different mediums have differing likelihoods of corrupting network messages [GR93].

Duplicate Messages: Networks contain routers, which are responsible for directing a message from a sender to a receiver node. In order to guarantee that a message has arrived at the receiver node a router usually requires an acknowledgement for each message sent. Acknowledgements are just another form of message and therefore can be corrupted or undelivered. A duplicate message could be caused by a router acting in the following manner. A network message is sent along a network path to a receiver node. The message arrives and is acknowledged by the receiving node. This acknowledgement is corrupted by the network and not received by the router. The router assumes that the message did not arrive at the receiver node and re-transmits it.

Out of Sequence Messages: The existence of multiple network paths from a sender node to a receiver node introduces the possibility that network messages can arrive out of sequence. For example, a router may choose two different network paths over which to send two messages. Even though the messages are sent from the router in the correct order, the various propagation speeds of network mediums and the varying traffic of network paths may cause the messages to arrive out of order.

Network partitioning resulting in undelivered messages is the most difficult of the above problems to cope with. Network partitioning is another way in which inconsistent

2In fact, partial failure is also inherent in local systems. This is attested to by the ubiquity of transactional systems in single-node database systems.
state can be introduced into a network. In terms of the transfer example above, it makes no discernible difference whether a transfer is not completed because a node is crashed or because a network message was not delivered.

Corrupted, duplicate and out-of-sequence network messages are usually dealt with by a distributed system's underlying software communications package. For example, the system used in this thesis is built on top of Smalltalk, which in turns operates on a Sun Unix platform. Sun Unix implements the DARPA internet standard protocols IP/TCP/UDP [Mic90, Com88, CS91]. The Internet Protocol (IP) handles network message routing and message corruption. The Transmission Control Protocol builds on the IP to deal with corrupted, duplicate and out of sequence messages. TCP offers a reliable, connection oriented [Mic90, Com88, CS91] protocol to network applications. It guarantees that if a message can be delivered then it will be delivered only once, will not be corrupt, and that all messages in a connection are delivered in sequence. The code developed in this thesis uses the socket [Mic90, Com88, CS91] abstractions that in turn use TCP to send network messages from one node to another.

### 2.1.4 Network Communication

Various types of networks exist and can be characterised by the speed, reliability and the distance that they operate over [BST89, Com88]. For example, a Local Area Network or LAN is typically a high speed (approximate transfer rate of 10 Megabits per second for Ethernet LAN), high reliability (message corruption is rare) and low operation distance (in the order of thousands of meters). Other network types, including Wide Area Networks (WANS), have different operating characteristics. In a WAN the data transfer rate is slower, (transfer rates of 19200 bits per second or less are common over a telephone line), messages are more likely to be corrupted and the distances involved are large. The introduction of new technologies [DHR93] is starting to blur the distinction between local and wide area networks. Hermes/ST is implemented and tested over a local area network that is connected via Ethernet. Therefore, network messages are more costly than accessing local node information, but are still reasonably efficient.

### 2.2 Programming Distributed Systems

To ease the programming of distributed systems convenient abstractions can be introduced. The abstractions mask the effects of the issues described above. Each abstraction assumes some distributed computer systems model and offers the application developer tools to ease the production of systems within that model. Such abstractions are henceforth referred to as distributed programming environments. Distributed programming environments are used to build distributed systems (consisting of distributed applications) that execute on distributed computer systems.

There are many distributed programming environments. See [CC91, BST89, Mul89, Mul93] for an overview. Different distributed applications place differing requirements on distributed programming environments [BST89, Mul89, Mul93]. There exists a spectrum of systems that range from extensions to standard operating systems such as RPC for Unix [BN84, Mic90] through to extensive systems such as OSF’s Distributed Computing Environment (DCE) [Fou92].

Distributed applications have various requirements with respect to integrity, availability and performance. Integrity refers to the reliability of information in an application [KV93]. A reliable application does not allow inconsistent state. For example, a

---

3 and Gateway Protocols
CHAPTER 2. TRANSACTIONS AND OBJECTS IN DISTRIBUTED SYSTEMS

distributed banking application requires high data integrity for account balances. Availability refers to the probability of access to the information in an application. For example, a distributed application may use replication techniques to increase data availability. Performance refers to the speed at which applications can execute. Applications can be classified as soft or hard real-time applications [KV93]. A flight control system is an example of a hard real-time system, whereas an on-line banking system is an example of a soft real-time system. This thesis is mostly concerned with soft real-time systems where data integrity is vital.

2.2.1 Persistent Data in Distributed Systems

In many distributed and non-distributed applications, data needs to be persistent. Persistent data is data that can survive node crashes. For example, account data in a distributed bank needs to survive node crashes. Distributed systems adopt several approaches to persistent data. Distributed file systems such as NFS, DOMAIN and Sprite [Sat93] favor performance over high data integrity or availability. Distributed file systems such as Coda [Sat93] and the Andrew [Sat93] file system use replication to increase the availability of data at the expense of performance. Distributed databases [Sat93] sacrifice performance to ensure data integrity. In distributed applications such as banking applications the integrity of data is paramount. In such systems, data should always be reliable.

The decision to use a distributed file system or a distributed database is usually based on a combination of factors. Databases encapsulate information about the type of data that they store, whereas file systems tend to treat data as uninterpreted byte sequences [Sat93]. Databases offer associative indexing of data, whereas file systems provide a single “name to file” indexing system [Sat93]. Each approach has advantages and disadvantages that are beyond the scope of this thesis. Importantly, in applications where the integrity of persistent data is paramount, such as in a distributed bank, the transactions offered by a distributed programming environment should be applicable to persistent data. Fortunately, a long standing and well understood abstraction called transactions deals with interleaving executions, inconsistent state and persistence. Furthermore, the transaction abstraction has been extended to work in a distributed environment. Transactions are introduced the next section.

2.3 Transactions in Distributed Systems

Transactions[BHG87, GR93, Wei93b] were originally developed for databases in the early seventies [BD72, Bjo73, Dav73]. Transactions are an abstraction that ensure consistent state for correct applications in the presence of partial failures and parallelism. Transactions were originally offered as an abstraction for single node databases and later extended to distributed systems [BG81, GR93]. Many distributed systems have applied distributed transactions as a means of maintaining consistent system state [CC91, BST89, Mul89, Mul93]. The transactional model as applied to distributed computer systems is now introduced.

A transaction models an application as a group of read and write accesses to an application’s data. As presented here, no structure is imposed on the data except that individual datums can be read and written. An application can be distributed over various nodes. A node possesses volatile storage sometimes referred to as local memory.

---

4Single node systems are also subject to partial failure. A disk can fail independently of a processor. Furthermore, disk blocks can be independently corrupted.
A node also contains non-volatile storage that is sometimes referred to as stable storage or disk. Transactions have three properties [BHG87] that ensure consistent state (data integrity). These are serialisability, atomicity, and persistence. Gray [GR93] refers to these properties as the ACID properties. A transaction is Atomic (cf. atomicity). Isolated (cf. serialisability) and Durable (cf. persistent), where the combination of these three properties ensures Consistency (The C in ACID).

Serialisability: The read and write data accesses of parallel and pseudo parallel transactions appear as if they have not interleaved. This masks out the interleaving problem introduced above.

Atomicity: A transaction either happens in its entirety ("commits") or not at all ("aborts"). No partial state is allowed. In a distributed system this effectively masks out the independent failure of nodes.

Persistence: If and when a transaction commits then its effects are persistent. An application’s non-volatile data will be accessible after a non-catastrophic node crash.

Transactions assume a particular failure model for a distributed system. Transactions as used in Hermes/ST, assume the following model.

- A node can crash at any time but is only crashed for a finite amount of time. Two types of crashes are distinguished. A non-catastrophic crash (e.g. a power failure) invalidates volatile storage but not non-volatile storage. A catastrophic crash (e.g. disk explosion) invalidates volatile and non-volatile storage. Only non-catastrophic crashes are inside the model. If catastrophic crashes are expected, then some form of replication strategy such as a tape backup or disk mirroring must be instituted. These replication strategies can be implemented to make the probability of a catastrophic failure arbitrarily small.

- Network messages are either delivered correctly or are lost. There are no duplicate, corrupt or out of sequence messages.

Transactions appear to be ideally suited to distributed computer systems. Their desired failure model can be achieved by a distributed computer system and transactions offer a suitable level of abstraction for an application developer. Transactional applications do not interleave; node and communication link failure during a transaction are masked in such a way that a transaction either completes entirely or appears not to have run at all. If a transaction commits then its results survive all future (non-catastrophic) node crashes. Transactional properties are ensured by mechanisms commonly termed concurrency control (for serialisability) and recovery (for atomicity and persistence). Both mechanisms are discussed in the following sections.

2.3.1 Concurrency Control

2.3.1.1 Serialisability

As noted in [GR93] the literature on concurrency control "is vast". Only a small subset of concurrency control approaches are discussed here. The interested reader is referred to [GR93, BHG87]. Serialisability is the definition of correctness of concurrency control in transactional systems [BHG87]. The goal of concurrency control in these systems is to provide serialisability in order to avoid any errors that can potentially be caused by interleaving transactions.
Reconsider the deposit example of Section 2.1.1.1, where the first deposit operation is called transaction \( T_1 \) and the second deposit operation is called transaction \( T_2 \). The schedule of a transaction is defined to be the set of instructions that the transaction executes. Any interleaving of these two transactions can be trivially avoided by a sequential execution of \( T_1 \) and \( T_2 \), i.e. all of \( T_1 \) happens before all of \( T_2 \) or vice versa. In general, a serial schedule of two transactions \( T_1 \) and \( T_2 \) is defined to be one in which either all operations of \( T_1 \) execute before all operations of \( T_2 \), or all operations of \( T_2 \) execute before all operations of \( T_1 \). Notice that this definition does not state in which order \( T_1 \) and \( T_2 \) execute. It just requires that they execute without any interleaving. A serial schedule of a set of transactions is defined to mean that all pairs of transactions in this set are scheduled serially.

By definition, a serial scheduling of all transactions in a distributed system avoids any transactional process interleaving. No parallelism is allowed. The exclusion of parallelism from nodes is undesirable because it amounts to poor utilisation of a nodes resources. This is reflected in poor performance, bad response time and poor fairness characteristics. Poor resource utilisation due to a lack of parallelism is only exaggerated in distributed systems that consist of many nodes and that share data.

To keep the clean semantics of serial schedules while still allowing parallelism, serial schedules are extended to the concept of serialisable schedules. A serialisable schedule of two transactions \( T_1 \) and \( T_2 \) is meant to be an execution of \( T_1 \) and \( T_2 \) so that they have the same effect on the "system state" as they would have if they had been scheduled serially. The "system state" refers to the particular state that the transactional system models. For example, in the bank deposit example, the relevant system state is the account being deposited to. Analogously, a serialisable schedule for a set of transactions requires serialisable schedules for all pairs of transactions in that set. There are many more serialisable schedules for a set of transactions than there are serial schedules. This is because every serial schedule is also a serialisable schedule but not every serialisable schedule is a serial schedule. Serialisable schedules allow parallelism amongst transactions as long as this does not affect system state.

As was the case with serial schedules, no particular execution order is specified for serialisable schedules. If the application developer requires some order then this order must be artificially imposed. For example, if a transaction \( T_i \) must be performed before a transaction \( T_j \), then \( T_j \) should only be started after \( T_i \) has committed.

2.3.1.2 Optimistic versus Pessimistic Concurrency Control

Concurrency control is performed by concurrency controllers. Concurrency controllers regulate accesses to individual data items to ensure that serialisability is achieved. Data items are accessed via read and write accesses. There are many different approaches to ensuring serialisability for transactions [BHG87, Wei93b, Wei88, Wei89, SS84, GR93, BR92, BK91]. The presentation here is based on that of [BHG87]. A concurrency controller has three options when a transaction requests some form of access to a data item. It can:

1. Perform the request immediately.
2. Delay the request for a finite amount of time.
3. Reject the request.

Concurrency control strategies can be classified according to the combination of these three options employed.
Optimistic concurrency control uses Options 1 and 3. Data accesses are scheduled immediately (Option 1) when they can be. However, a system can get into situations in which there is no possibility of finishing all transactions in a serialisable way. If this situation arises then the system has to reject (Option 3) one transaction's access request. This rejection will causes the transaction issuing the access request to abort.

Pessimistic concurrency control favours Option 2. Data access requests are delayed until serialisability can be ensured. For example, if two transactions both request a write access to a data item, then one transaction is delayed (Option 2) until after the other is completed. However, the system may get into deadlock situations (see section 2.3.1.3.1. Deadlocks cause some access requests to be rejected (Option 3) and transactions to be aborted.

2.3.1.2.1 Optimistic Concurrency Control Optimistic concurrency control potentially allows higher concurrency, but the aggressive nature in which it schedules transactions may lead to cascading aborts. A cascading abort is a special type of transaction abort. It is caused when a transaction uses the uncommitted state of another transaction and the other transaction has to abort. For example, consider the following example adapted from [BHG87]. Suppose that the initial values of two data items, named $x$ and $y$, are 0. Further suppose that transactions $T_1$ and $T_2$ issue operations that are executed in the order shown in Table 2.2.

<table>
<thead>
<tr>
<th>execution order</th>
<th>$T_1$</th>
<th>$T_2$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td>write($x$, 1)</td>
<td></td>
</tr>
<tr>
<td>2.</td>
<td>read($x$)</td>
<td></td>
</tr>
<tr>
<td>3.</td>
<td>if $x = 1$ then write($y$, 2)</td>
<td></td>
</tr>
</tbody>
</table>

Table 2.2: Cascading Abort Example.

Suppose that $T_1$ aborts. Then, the system undoes $T_1$'s write($x$, 1) operation, restoring $x$ to the value 0. Since $T_2$ has been allowed to see the uncommitted state of $T_1$ it will have executed write($y$, 2). However, because $T_1$ has been aborted then all of its effects on the system should be undone. Therefore, $T_2$ should not have written any value into $y$. Because $T_2$ has seen uncommitted state of $T_1$ it must be also aborted. This is a cascading abort.

Cascading aborts are undesirable for two main reasons. Firstly, cascading aborts represent wasted work (i.e. the abortion of $T_2$). Secondly, cascading aborts can require significant bookkeeping to maintain the dependencies between transactions needed to cascade aborts [Wei93b].

2.3.1.2.2 Pessimistic Concurrency Control Pessimistic concurrency control is a more conservative scheduling mechanism that uses delays to avoid cascading aborts. By delaying requests for access to data items, a pessimistic concurrency controller ensures that one transaction never uses the uncommitted state of another transaction. However delaying accesses to data items may result in deadlocks. Deadlocks are described in section 2.3.1.3.1. Deadlocks are undesirable, but in practise tend to be rarer than cascading aborts. This is because a deadlock requires a cycle in the dependency of transactions, whereas a cascading abort requires only a dependency.
Choosing Optimistic or Pessimistic Concurrency Control

Simulation studies [ACL87, BHG87, AC92] show that neither a pessimistic nor an optimistic concurrency control strategy is better in all cases. The performance of each type of concurrency control strategy depends on a complicated mix of many factors including: the number of data items, the number and types of nodes, the number of transactions and the access patterns of these transactions over their data items.

The consensus viewpoint (at least for database type applications) appears to favour the pessimistic approach. Most existing database systems use a form of pessimistic concurrency control [GR93, ACL87]. However, [ACL87] notes that it is an open question as to whether the sorts of environments in which optimistic algorithms do better will ever be common. These are stated as being databases for which there are many interactive applications that place relatively low demands on a (distributed) system’s resources. Bernstein et al [BHG87] conclude that in domains where conflicts are common, a pessimistic scheme is preferable, whereas in a domain where conflicts are rare then an optimistic approach is better. This thesis utilises a form of pessimistic concurrency control.

Pessimistic Concurrency Control via Two-Phase Locking

Pessimistic concurrency control can be achieved via three mechanisms: two phase locking, timestamp ordering and serialisation graph testing [BHG87]. Two phase locking (and a particular variant called strict two-phase locking) is the most commonly used locking mechanism in commercial transaction systems [BHG87, GR93]. Strict two phase locking is introduced below, and is assumed throughout this thesis.

When two phase locking is employed for concurrency control, locks are associated with data items. The schedules of transactions that access a data item are serialised by delaying any accesses that might defy serialisability. Before accessing a data item, some form of lock is acquired for that data item. Database systems do not typically have semantic knowledge of data items and therefore use variants of read/write locking. Distributed systems supporting transactions allow individual data types to define their own locking modes (see section 4.2). Locks use a compatibility matrix to determine whether or not an access needs to be delayed. The compatibility of locks is usually based on the commutativity of accesses [Wei93b, Wei89]. For example, the compatibility matrix for read and write accesses to a data item is shown in figure 2.1. Read accesses do not alter the state of a data item, and therefore commute. Thus read locks are compatible with other read locks. Write accesses change the state of a data item and therefore do not commute with other read or write accesses. Thus, write locks are incompatible with

---

**Figure 2.1: Standard Read/Write Locking Compatibility Matrix**

<table>
<thead>
<tr>
<th>Granted Locking Mode</th>
<th>Requested Locking Mode</th>
<th>Read</th>
<th>Write</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Read</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td></td>
<td>Write</td>
<td>No</td>
<td>No</td>
</tr>
</tbody>
</table>
other write and read locks. Read/write locking allows multiple concurrent transactions to read a data item, but only one transaction to write it at any one time.

After acquisition, locks are held by a transaction until they are released. A transaction cannot acquire a lock as long as it is held by another transaction in a conflicting or incompatible mode. The term "two phase" refers to the manner in which locks are acquired and released. With respect to locking, a transaction consists of two phases. During the first phase locks are acquired, and during the second phase locks are released. Once a transaction has released any locks it cannot acquire any more locks. When strict two phase locking is employed, transactions do not enter their second phase until they are ready to commit or abort. The second phase is delayed until the commit or abort of a transaction for two reasons [Wei93b]. Firstly, without extra information, the only time a transaction is guaranteed to have finished its first phase is at transaction completion. Secondly, if locks are released before commit or abort, then other transactions may see the uncommitted state of the transaction. This could lead to the cascading aborts described above. The serialisability of strict two phase read write locking has been proven [EGLT76, BHG87, GR93].

2.3.1.3.1 Deadlocks

Strict two phase locking does, however, give rise to deadlocks [Tan87, BHG87, GR93]. A deadlock scenario is depicted in table 2.3. \( T_1 \) and \( T_2 \) denote two transactions. \( x \) and \( y \) denote two data items.

<table>
<thead>
<tr>
<th>Execution Order</th>
<th>( T_1 )</th>
<th>( T_2 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td>acquires read lock on ( x )</td>
<td></td>
</tr>
<tr>
<td>2.</td>
<td></td>
<td>acquires read lock on ( y )</td>
</tr>
<tr>
<td>3.</td>
<td>tries to acquire write lock on ( y )</td>
<td></td>
</tr>
<tr>
<td>4.</td>
<td></td>
<td>tries to acquire write lock on ( x )</td>
</tr>
</tbody>
</table>

Table 2.3: Deadlock Example.

After step 4 the two parallel transactions \( T_1 \) and \( T_2 \) are deadlocked. \( T_1 \) cannot finish until it acquires a write lock on \( y \) which is read locked by \( T_2 \). \( T_2 \) cannot finish until it acquires a write lock on \( x \) which is read locked by \( T_1 \). Both \( T_1 \) and \( T_2 \) are waiting on each other to finish, yet neither transaction can finish. No progress is possible unless at least one of the two transactions releases a lock. However, strict two phase locking requires that locks are not released until a transaction has committed or aborted. Neither transaction can commit so therefore, one transaction needs to be aborted.

In this example, \( T_1 \) waits for \( T_2 \) and \( T_2 \) waits for \( T_1 \). The waits-for relationship between transactions can be represented as waits-for graph [GR93]. A waits-for graph is a directed graph where nodes represent transactions and arcs represent waits-for relationships. Deadlock occurs when there is a cycle in the waits-for graph.

There are three main approaches to handling deadlocks [SPG91, Tan87]. These are prevention, avoidance and detection.

Prevention: Deadlocks can be prevented by ordering the accesses to data items in such a manner that transactions cannot form cycles in waits-for graphs. One way to achieve this is to define a system wide canonical order over data items and insist that transactions access data items according to this order [Kor82]. For the example above assume that the canonical order is \( x \) before \( y \). This would mean \( T_2 \) would have to acquire its write lock on \( x \) before acquiring a read lock on \( y \). The deadlock could not then occur. Notice that such a canonical ordering on data items restricts the manner in which transactions are expressed. For example, if \( T_2 \) is data
dependent [Wei93b] on \( x \) then such canonical orderings make transactions harder to denote. For example, if \( T_2 \) should only write to \( y \) if \( x = 0 \), then some form of convoluted logic\(^5\) has to be implemented by the transaction to ensure that \( T_2 \) locks \( x \) before \( y \). Another strategy is to have transactions predeclare the data items that they intend to access. Transactions can then be scheduled in such a manner that deadlock will not occur. Predeclaration is also problematic when transactions are data dependent.

**Avoidance:** Avoidance can be seen as an execution time version of prevention. Rather than avoiding deadlocks via some form of predeclared order, they are avoided by disallowing runtime access patterns that could potentially lead to a deadlock being formed. There are various mechanisms that disallow potential deadlocks. The simplest of these is called no-waiting. When no waiting is used a transaction that would have to wait for a lock is aborted. In the example above, when the transaction \( T_2 \) tries to acquire a write lock on \( x \), then because some other transaction \( (T_x) \) holds this lock, \( T_2 \) would be aborted and started at some other time. No-waiting forces many unnecessary transaction abortions and is prone to a phenomenon called cyclic restarts or livelock [BHG87].

More sophisticated deadlock avoidance techniques allow transactions to wait as often as possible while still avoiding deadlock. Schemes such as cautious waiting and timestamp-based approaches like wound-wait and wait-die [RSL87, BHG87] have been identified. A timestamp based approach to deadlock avoidance assigns a time stamp to each transaction and uses rules about time stamps to avoid deadlock by aborting victim transactions. For example, a wait-die rule forces the abortion of transactions that have accessed a data item but are older than a transaction that wishes to access a data item. With respect to the example above, assume that \( T_2 \) is older than \( T_x \). The execution sequence above would cause the termination of \( T_2 \) at step 4. This is because \( T_2 \) is older than \( T_x \), which has already locked \( x \).

**Detection:** Detection of deadlocks can be performed either aggressively or conservatively. Aggressive detection of deadlocks involves building a waits-for graph and actively checking for cycles. When a cycle is found, a victim transaction is identified and aborted. The decision about which transaction to abort can be made according to various factors including: how nearly complete each transaction is and how many cycles will be broken by the abortion of any one transaction. In a distributed system the cost of aggressive deadlock detection is increased by the need to communicate information about waits-for relationships. Either a centralised deadlock detector, or distributed path pushing [BHG87] can be used.

Conservative deadlock detection is performed via timeouts. Timeouts assume that if a transaction is not completed within the period of the timeout then some misfortune, possibly a deadlock, has occurred and the transaction is restarted. The primary problem with the timeout strategy is determining a suitable value for the timeout. Too large a timeout will cause long delays for transactions that are involved in a deadlock. Too short a timeout can cause the system to erroneously classify transactions as being deadlocked.

---

\(^5\)Assigning \( x \) to \( x \) would satisfy the locking order without changing the semantics.
CHAPTER 2. TRANSACTIONS AND OBJECTS IN DISTRIBUTED SYSTEMS

2.3.2 Recovery

Recovery mechanisms are used to ensure that inconsistent state is avoided by (distributed) transactional systems. Both network partitions and independent node failures can introduce inconsistent state (refer sections 2.1.2 and 2.1.3). Catastrophic failures are outside the model (refer section 2.3). Recovery must maintain both the atomic and persistent properties of transactions (refer section 2.3). Two separate issues can be identified. Transactions must be able to be aborted, and application state must survive non-catastrophic node crashes.

2.3.2.1 Recovery from Transaction Aborts

Transactions can abort due to node crashes, deadlocks, timeouts, undeliverable messages or because the application explicitly requests an abort. The atomicity property of transactions requires that a transaction appears to have completed entirely or appears not to have run at all. With respect to system state there are four operations that a transaction can perform. It can read a data item, write a data item, commit or abort. Two main mechanisms for abort recovery are distinguished, namely update-in-place and deferred-update [Wei93b]. Both mechanisms rely on transactions keeping some form of log of changes to system state. This log may or may not need to be stored in non-volatile memory. For example, if data items are pinned in main memory during a transaction then the log can also be implemented in memory (See section 2.4.2).

Update-in-place: uses an undo log [BHG87, GR93, Wei93b] to recover from transaction abortions. Write accesses to data items are performed on the data item and an undo log record is appended to the log. Read accesses do not affect the log. A commit causes the log to be discarded. However, if a transaction aborts, then the undo log records are used to restore all write accessed data items to the values they had before the transaction started.

Deferred-update: uses a redo log [BHG87, GR93, Wei93b] to recover from transaction aborts. Write accesses to data items are not performed on the data item but instead a redo log record is appended to the log. Read accesses must use a combination of the redo log and the data item to ensure that up-to-date information is read by a transaction. An abort causes the log to be discarded. However, if a transaction commits, the redo log entries of the aborting transactions must be applied to the data in a first-in first-out order.

Update-in-place implements data accesses and commits in a more cost effective manner than deferred-update. Deferred-update provides an efficient implementation of aborts. Update-in-place is more commonly used than deferred-update [Wei93b, BHG87, GR93]. This is because it is hoped that transaction commits will be more frequent than transactions aborts. It is also because the typically more common read accesses are more efficient when update-in-place is used. Undo logging outperforms redo logging in applications where read operations are common and transaction aborts are rare. However, deferred-update can allow more concurrency in certain situations than update-in-place [Wei93b].

2.3.2.2 Recovery from Node Crashes

When a node crashes the contents of volatile memory are lost but non-volatile memory stays intact. Any transactions that are active are treated as having been aborted. The

---

6Databases usually do not pin all data items involved in a transaction in memory [GR93]. Hermes/ST does.
persistence property of transactions requires that the effects of a committed transaction must survive node crashes. This is achieved by storing data items in non-volatile storage. The persistence of transactions is first described for single node transactions and then the description is extended to distributed systems.

2.3.2.2.1 Single Node Recovery from Crashes
Non-volatile memory is typically offered by disks. Disks have a particular operations model that effects the manner in which transactions are committed. A disk consists of many disk blocks. Disk blocks can be read or written. A node can determine with an arbitrarily high probability whether a read or write of a block was successful. However, an unsuccessful write of a disk block can corrupt that block. Either logging or shadowing are used to implement persistence [BHG87, GR93, Wei93b].

2.3.2.2.1.1 Logging
When logging is used to implement transaction commit, both the log and the data items are stored on disk. Data updates, commits and aborts of transactions are recorded in this log via log records. Logging implements persistence by ensuring that all changes to data items are kept in different blocks to the actual data items. In the event of a crash, the persistent state of an object can be reproduced by replaying the log. Two kinds of log records are distinguished. Update records contain the appropriate redo or undo information for a data item and status records contain commit and abort information.

Because disks are typically much slower than volatile memory it is desirable to perform as much writing to the log as possible in an asynchronous fashion. In order to allow such asynchronous writes two rules must be satisfied. Which rule to use depends on the type of logging that is being employed. If update-in-place (undo logging) is being used, the undo rule [BHG87] or write ahead protocol [Wei93b] insists that before a data item is changed on disk an undo record must have been previously written in the log. If deferred-update (redo logging) is being used, then the redo rule [BHG87] insists that before a transaction can commit, any data items it wrote must be recorded in the log.

After a system crash, the persistent state of data items can be produced by replaying the log against the data item. To allow for the possibility of a node crash during a restart it is important that the restart procedure be idempotent [BHG87, Wei93b]. Idempotence of restart means that a partial execution of a restart followed by a total execution of a restart produces the same result in stable storage as a single total execution of a restart. To bound the size of the log, update and status records can be purged from the log according to a garbage collection rule [BHG87]. Furthermore, checkpointing [BHG87, GR93, Wei93b] allows a prefix of the log to be discarded.

2.3.2.2.1.2 Shadowing
Shadowing eliminates the need for a log by performing all of a transaction's updates in a single atomic action. By maintaining separate versions of data items and a level of indirection between a data item and its value, it is possible to perform all of a transaction's updates as an atomic action. As a transaction proceeds each data item it write accesses is written to a separate version on the disk. A directory [BHG87, GR93] is used as a level of indirection between the data item and its disk version. At transaction commit, a new copy of the directory is written to the disk. The installation of this new copy must be performed as an atomic action. This is achieved by changing a single bit in a master record [BHG87]. Before this atomic installation any crashes of the system will cause the old (untouched) version of data items to be used.

Shadowing, although more elegant than logging, does suffer performance problems. Firstly, access to persistent state is indirect and therefore more expensive. Secondly,
persistent versions of data items tend to get scattered around the disk and thus any clustering advantages are lost. However, shadowing requires almost no work in the event of a node crash.

2.3.2.2 Distributed Recovery from Crashes No matter whether logging or shadow paging is used a transaction that has written data items at different nodes needs to make its changes persistent on all nodes atomically. During a commit, there must be no inconsistent state introduced into the system because some part of the transaction has been made persistent while some other part has not. The two phase atomic commitment protocol [BH87, GR93, Wei93b] or two phase commit is used to ensure this. The two phase commit works as follows. One node that has been involved in the committing transaction is chosen as the coordinator. The co-ordinator and all other nodes involved in the transaction are called participants. The two phases of the commit protocol are called the prepare phase and the commit phase and work as follows:

**Prepare Phase:** The coordinator sends a prepare message to all participants to log or shadow the transaction's changes to non-volatile storage. If a participant has logged or shadowed these changes, it replies positively, otherwise it replies negatively. A positive reply represents a confirmed intention by a participant to commit during the second phase.

**Commit Phase:** If all participants have replied positively, the coordinator can decide to commit. The decision must be recorded on disk (e.g. in the log) before all participants are informed about it. The reason for this is so that a node crash of the co-ordinator can be recovered from. If the transaction is to be committed then all participants are sent a commit message. When the participants receive a commit message then they must make permanent the effects of the transaction. If logging is used, this entails adding a commit status record to the log. If shadowing is used this entails atomically installing new directory information. The commit phase finishes only after the coordinator has received positive replies from all participants. If any participant have replied negatively during the first phase of the two phase commit then the co-ordinator decides to abort. Participants are informed to discard their prepared information at this time.

The two phase commit protocol is prone to blocking. A crash of the co-ordinator during the commit phase may cause all participants to block (and hence retain locks). A crash of a participant causes the co-ordinator to block. More sophisticated, but also more expensive, mechanisms like the three-phase commit protocol [Ske82, BHG87] address blocking and other problems with the two phase commit protocol.

2.4 Nested Transactions

The distributed transaction model introduced so far is sometimes called a flat transactional model [GR93]. It suffers from the following restrictions

- Flat transactions do not allow the composition of several simple transactions into more complex transactions.
- Flat transactions do not allow concurrency within a transaction.
- A flat transaction must be committed or aborted in its entirety.
Nested transactions [Ree78, Mos81, Lis88] relieve the restrictions of a flat transactional model. In the nested transaction model, transactions can create other transactions called nested transactions. 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 [Hum94] (see section 3.1.5). The root node of a transaction tree is called a top-level transaction. Nested transactions can execute synchronously or asynchronously. An asynchronous nested transaction is serialised with respect to its sibling transactions. However, a nested transaction may abort without necessarily forcing the parent transaction to abort. Furthermore, the persistence of a nested transaction is not visible until the top-level transaction has committed. These new semantics require extensions to existing concurrency control, state restoration, and persistence handling techniques. Two main mechanisms for implementing nested transactions have been proposed: Reed's mechanism, based on timestamp ordering [Ree78]; and Moss' mechanism, based on extensions to strict two phase locking [Mos81]. A brief overview of these extensions as applied to two phase locking is given below.

2.4.1 Concurrency Control

The strict two phase locking is extended to deal with nesting. In essence, a lock can only be acquired by a nested transaction if the lock is held by some ancestor transaction in the transaction tree. At nested transaction commit the locks are inherited [Lis88] by the parent transaction. If a nested transaction aborts then its locks are discarded. There are several implementations of nested transactions based on extensions to the strict two phase locking including [Mos85, Lis88, BBG89, HR93, Hum94].

2.4.2 State Restoration

The ability of a nested transaction to abort independently of its parent transaction introduces the need for a version stack [Wei93b, Lis88, SDP91, EME91] for data items. Version stacks can be implemented via an update-in-place or via a deferred-update strategy [Lis88, SDP91, EME91]. When a deferred-update strategy is used the first write access of a data item causes a push of the new data item onto its version stack. Any future accesses by this nested transaction are performed on the current top of stack. A nested transaction commit causes the top of the stack to replace the parent's version. A (nested) transaction abort causes the top of the stack to be discarded.

An update-in-place strategy cause a copy of the data item to be placed on the version stack before the data item is write accessed. Any future accesses by this nested transaction are performed on the data item. A nested transaction commit causes the version stack to be popped. A (nested) transaction abort causes the data item to be restored to the appropriate version stack level.

2.4.3 Persistence Handling

The effects of committing nested transactions are conceptually not made permanent. There are, however, early writing and checkpointing strategies that buffer the effects of subtransaction commit to non-volatile storage before top-level transaction commit. These strategies are used to reduce the amount of work to be done at top-level transaction commit or to reduce the likelihood that a node crash will cause a top level transaction to abort. They are not necessary to ensure the semantics of nested transactions. Naturally,
CHAPTER 2. TRANSACTIONS AND OBJECTS IN DISTRIBUTED SYSTEMS

2.5 Object-Orientation in Distributed Systems

2.5.1 Object-Orientation

Object-orientation [Mey88, Boo90, WBWW90, RBP+91] is a general purpose programming abstraction that attempts to make aspects of programming easier. Object-orientation has also been applied to distributed systems in an attempt to ease the complexity of distributed systems. Object-orientation introduces several notions, including the notions of objects, classes, inheritance and dynamic binding [BGHS91].

2.5.1.1 Objects

An object is an entity that encapsulates [Mey88] both state and behaviour of an applications components. State is encapsulated by variables and behaviour is encapsulated via methods. Methods consist of a body and an interface. An object’s variables are protected and hidden from other objects. An object’s state can be altered indirectly by performing a method.

Objects interact by invoking the methods of other objects. The invocation of an object’s method is achieved via message passing. Both of the terms, “method invocation” and “message passing” are used interchangeably throughout this thesis. Information can be passed from one object to another in two ways. An object can pass information as arguments to a method. An object can return information from a method invocation.

Some object oriented systems distinguish between public and private methods. Public methods can be invoked by other objects whereas private methods can only be invoked by the object itself. The implementation of an method body is not visible to other objects. Only the interfaces of public methods are visible to other objects.

Some object oriented systems insist that only objects be present in the system. Such a uniform object model is present in Smalltalk [GR89]. Other systems allow a mixture of objects and primitive data types [Atk91].

2.5.1.2 Classes

The class concept can be seen as an extension of the abstract data type [Knu73] concept. A class is a template from which objects can be created [BBL91]. A class contains the definition of the variables and methods for a set of objects. Objects are created from classes by a process known as instantiation [Mey88, BBL91]. Every object is an instance of some class. In languages that support a uniform object model, a class itself is an object, so the process of object instantiation is simply another form of sending a message to an object.

2.5.1.3 Inheritance

Classes can be described as extensions to existing classes. A mechanism called “inheritance” allows this to be done. A class that inherits the behaviour of another class is called a subclass. Conversely, a class that provides the inherited behaviour for another class is called a super-class. When a class is viewed as a template8 the inherits relationship is

8Other interpretations of class are possible. See [Atk91].
viewed as the *reuse* of the superclass [Atk91]. A subclass conceptually owns a copy of its superclass. Therefore, instances of a subclass encapsulate not only all variables and methods defined in any subclass definition but also all variables and methods defined in any superclass definitions. The inheritance mechanism is recursive. Thus one class can have several, super and sub classes. The terms *descendent class* and *ancestor class* are used for repeated subclass-superclass relationships. The structure formed due the inheritance relationship is referred to as a class hierarchy.

Variables and methods that are reused from an ancestor class are said to be *inherited* by a 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 known as *overriding* inherited methods. When *strict inheritance* [Atk91] is employed, a subclass cannot delete the variables or methods of a superclass.

Inheritance can be either *single* or *multiple* [Mey88, Atk91, Gal91]. When multiple inheritance is supported, then one class can have several immediate superclasses. While this allows a more general class hierarchy, and hence modelling power, some problems are introduced [Gal91]. For example, if a subclass inherits methods from two, ancestor classes then some way of un-ambiguously referring to an ancestor class’s method is required.

### 2.5.1.4 Dynamic Binding

*Binding* is the process of resolving names in a class hierarchy [Bla91]. Binding can either be *static* or *dynamic*. Static binding is performed prior to the execution of methods, typically during a compilation process. Dynamic or late binding defers the resolution of names to methods until method execution. Most object oriented systems support dynamic binding.

### 2.5.2 Flexible Sharing

Inheritance alone, supports *programming by difference* [Mey88]. Programming by difference allows a class to reuse the *implementation* of its ancestor classes. Inherited classes can override the definitions of methods as is necessary.

The combination of inheritance and late binding give rise to various forms of sharing of specifications and implementations. One example of this is *inclusion polymorphism* [Bla91]. By deferring the resolution of method names until method execution *abstract classes* [Atk91] can be defined. For example, a bank account class hierarchy may contain subclasses for savings accounts and cheque accounts. The bank account class can define a “calculate interest” method that uses a “get interest rate” method. The bank account class does not need to implement the “get interest rate” method. Deferring the implementation of the “get interest rate” method allows both savings and cheque accounts to define their own version of this method. This offers three main advantages:

1. The code for calculating interest is not repeated in the savings and chequing account classes.
2. The implementation of the “calculate interest” method can be modified without modifying the savings and chequing account classes.
3. Other specialised account classes, such as a high interest rate account, can be developed without affecting the bank account class. For example, if the high interest rate class defines its own “get interest rate” method, then it automatically acquires the functionality of the “calculate interest” method.
Other examples of the sharing of specifications and implementations include parametric polymorphism (generic classes), and prototyping [Bla91]. Parametric polymorphism supports the sharing of implementations in a type consistent\(^9\) manner. For example, a collection class, such as a list, may be parameterised so that it can work over integers, strings, dates or arbitrary user defined classes. Prototyping allows behaviour sharing by including the implementation of one object in another. Prototyping is useful in domains where only single inheritance is available but an object needs to display the behaviour of multiple classes. The object can uses instance variables to prototype multiple class behaviours.

### 2.5.3 Advantages of Object-Orientation

Although there are many different approaches to object-orientation, the concepts of object, class, inheritance and dynamic binding are common to all of them [BGHS91]. Object-oriented techniques are applicable to all phases of the software development cycle [Som89]. Prominent object-oriented programming languages include Smalltalk-80 [GR89], C++ [Str86] and Eiffel [Mey88]. The concepts of objects, classes and behaviour sharing have also been applied to the analysis and design phases of software development [Boo90, WBWW90, RBP+91, HS91, CY91a, CY91b]. When considered as a programming tool, object-orientation has the following advantages.

#### 2.5.3.1 Reusability:

Classes describe behaviour of abstract data types. The ability of classes to flexibly share information allows them to be reused. Reuse is desirable because it avoids repetition, aids in maintenance and improves the reliability and performance of software [Mey88]. Repetition of the programming of classes is avoided by reusing common classes from, say, a class library. Application maintenance is simplified by using consistent well tried interfaces. Software reliability is improved by the validation of classes in different application contexts. Substantial effort can be put into supplying efficient implementations for often reused classes.

#### 2.5.3.2 Incremental Development:

Object-oriented systems are modelled as groups of interacting objects. The public interface of a class describes the manner in which objects that are instantiated from a class can interact. After object interfaces have been specified, a prototype [Som89] implementation can be produced. The prototype implementations can validate a system’s design. A validated prototype can then be incrementally extended to produce the final application. This incremental extension process is aided by the flexible sharing techniques employed in object-oriented applications. Abstract classes define interfaces that can be implemented with different degrees of functionality. The interfaces need not necessarily change during the incremental development of an application.

### 2.5.4 Distributed Systems that Support Objects

As well as being a powerful software development tool, object-orientation is also well suited to the development of distributed systems. Several distributed systems support objects. See [CC91] for a survey. Some distributed systems supporting objects support inheritance,
whereas others do not. Systems supporting objects without inheritance are called object-based [BL92, CC91]. The reason for the omission of inheritance in object based systems is more to do with complexities of dynamic binding in distributed systems. Namely, the cost of searching potentially remote hierarchies at runtime can be large [BL92].

With or without inheritance, an object provides a convenient unit of abstraction for many properties of distributed systems. An object is a unit of tightly coupled state and behaviour, whereas different objects are typically loosely coupled via message passing. Such a view of a distributed application allows the object to become the unit of many distribution properties. For example, many systems use an object as the unit of remote access, parallelism, migration, replication and heterogeneity [CC91, Lim91]. Object migration, replication and heterogeneity are beyond the scope of this thesis.

2.5.4.1 Remote Access

By definition, all object based distributed systems support some form of remote object access. In many systems that do not support transactions, remote object access is based on some form of remote procedure call [BN84]. However a desirable property of remote object accesses is that they be transparent [Lim91]. However, remote procedure call has different semantics to local object invocation. Local method invocations happen zero-or-once [BN84, Wei93a] whereas remote object accesses based on remote procedure calls only supply at-most-once semantics. Thus a remote object invocation may have happened, not happened or partially happened. Distributed systems supporting transactions reintroduce zero-or-once semantics.

2.5.4.2 Parallelism

Another common use of objects is as a unit of parallelism in a distributed system. Objects can be passive or active [CC91]. Passive objects perform a method invocation without creating a separate process. Clouds [DLAR91] is an example of a system that supports passive objects. Active objects, typically create a process to perform a method invocation. The actors model [Agh86] is an example of an active object model. Active objects extract parallelism from method invocations, whereas passive objects execute at the level of parallelism specified by the application developer.

2.5.4.2.1 The Weight of Objects Most systems trade off the level of distributed functionality with performance. Therefore, not all objects supply all levels of distributed properties. It is common for one object to provide some distributed property for a group of objects [CC91]. Such distribution objects are themselves composed of other lesser capabilitied objects. Such objects are best described as heavy as opposed to light weight objects. The Emerald [BHJ+87] system is a good example of a system providing differently weighted objects. Large objects in Emerald support parallelism, remote access, migration, and concurrency control, whereas small objects like integers can be implemented in a "single word of storage" [BHJ+87].

2.6 Distributed Systems Supporting Transactions and Objects

Several distributed systems have combined the concept of objects and (nested) transactions. An important early implementation to integrate both technologies was Argus [Lis82, LS83, LCJS87, Lis88]. Argus is object based and supports special objects called
CHAPTER 2. TRANSACTIONS AND OBJECTS IN DISTRIBUTED SYSTEMS

Guardians. A guardian is completely contained by one node in a 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, state restoration and persistence handling. Concurrency control is based on extensions to strict two phase read/write locking. State restoration and persistence are based on version stacks. State restoration is performed by stacking in memory versions of atomic objects. An abort of a (nested) transaction causes the current version of an atomic object to be discarded. A commit of a top-level transaction causes the current version (top of stack) to be logged to stable storage. Non-atomic objects are volatile and do not provide concurrency control, state restoration or persistence handling.

Guardians are also 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. Transactional properties are ensured as long as transactions access atomic objects only. Non-atomic objects can be used to reduce the cost of actions that do not need transactional semantics.

Other research systems extended or improved the Argus approach to reliable distributed programming. Examples include, Camelot/Avalon [EME91], Locus [MMP83], TABS [SBD+84], Eden [PN85], Clouds [DLAR91], Arjuna [SDP91], Apertos¹⁰ [YTM+91], Venari/ML [HKM+94], Karos [GCLR92], Raven [FAC+94] and Hermes/ST [FHR94]. The idea of combining transactions and objects is starting to manifest itself in commercial systems. Two prominent examples are, ANSA [WR93], and DCE's Encina¹¹ [Tra91]. Hermes/ST, the distributed programming environment used in this thesis, has been influenced mostly by Argus. However, Hermes/ST is object-oriented rather than object based. Hermes/ST is introduced in more detail in the next chapter, chapter 3.

2.6.1 Object-Oriented Databases

The combination of objects and transactions, possibly nested, is not restricted to the types of distributed programming environments that have been introduced thus far. There is also a large body of work that deals with the so called object-oriented databases [Cat91, Hug91, GH91]. Object-oriented databases represent the next generation of database management systems and are intended to widen the applicability of database systems. Object-oriented database are intended for use in large, complex, data intensive applications such as those found in computer aided design, computer aided software engineering (CASE), and knowledge management [Cat91, Hug91, GH91]. Cattell [Cat91] identifies a mixture of thirty three tenets, twenty of which an object-oriented database "must" meet and thirteen "desirable" tenets.

Object-oriented databases are not identical to distributed programming environments. However, there is a large overlap between the functionality (Cattell's tenets) of an object-oriented database and that of a distributed programming environment. For example, both support, amongst other functionalities: transactions (although nested transactions are not supported in all object-oriented databases), objects, inheritance, encapsulation, a programming language to manipulate persistent objects, and distribution. A comparison

¹⁰Formerly called Muse.
¹¹Encina is based on the C programming language but is currently extended to provide object support [Dix94].
of the relevant aspects (i.e. concurrency control) of object-oriented databases with respect to the results of this thesis is presented in section 6.3.6.

It is worth noting that object-oriented databases have extended requirements over those of distributed programming environments. For example, some form of query language is a "must" tenet. A query language must provide a data manipulation language (DML) that allows ad hoc operations by an end user. The query language processor must provide a high degree of physical data independence and deal with any impedance mismatch between the programming language representation of object and the persistent representation of objects. Lastly, the query language should be a subset of the programming language. Distributed programming environments do not typically support query languages. Furthermore, concepts such as associative index of data and schema evolution (in the presence of instantiated objects) [Cat91, Hug91] are typically not supported in distributed programming environments.

It is also worth noting that distributed programming environments that support transactions and objects also have extended requirements over those of object-oriented databases. For example, the ANSA [WR93] distributed programming environment has the concept of a trader (a specialised name server) which is not present in object-oriented data bases. An another example, OSF's DCE [Fou92] supports: directory services (global and local name servers), distributed time services and a distributed file service which are not usually included in object-oriented databases.

2.7 The Distributed Bank Example

In order to supply a concrete example of a distributed application throughout this thesis an exemplar distributed bank application will be used. A distributed bank is often used as a test application for distributed programming environments that have high data integrity as a requirement[Lis88, EME91, Hew91].

The electronic bank modelled in this thesis is a simplified distributed bank. The bank consists of branches that exist on nodes of the distributed computer system. A branch exists on only one node, but each node can support multiple branches. A branch contains a collection of accounts. Accounts are organised at each branch in a binary tree [Knu73] sorted by account name. Each account contains the name of the account owner, the address of the account owner and the balance of the account. Accounts are persistent and must contain reliable balance information.

For the purposes of this thesis, branches support two main operations. An account can be opened, and an account can be deposited to. A real distributed bank would, of course, include many more operations, such as account deletions, funds transfers etc. Furthermore, the implementation of account open and deposit operations would address other system issues such as security. Lastly a real distributed bank would tend to use a more sophisticated data structure than that of a binary tree. A binary tree was chosen to keep the presentation of the ideas in this thesis relatively simple.

2.8 Domains of Applicability of Distributed Programming Environments

While the example used throughout this thesis and literature is that of a distributed bank, this work and distributed programming environments in general, obviously have much wider applicability. One particular domain is that of telecommunications. Telecommunication networks are by their nature distributed, and also have requirements for reliable
(high integrity) data. For example, aspects of telecommunications include billing, network configuration and advanced telecommunication services.

An area of particular interest to the Telecommunications Software Research Centre (TSRC)\textsuperscript{12} and Telecom is TINA. TINA [Con92] is the Telecommunication Information Networking Architecture. TINA is a current international research effort. Its goal is to develop an architecture that will "enable the efficient introduction, delivery and management of telecommunications services and infrastructures" [Con92]. A key component of the TINA architecture is an object-oriented distributed programming environment [FLNP92, DPK+93]. One desired service of the distributed programming environment is support for distributed nested transactions over objects. The applicability of transactions in TINA is anticipated to be much wider than in existing networks. This is because of the proliferation of advanced services that TINA will enable. For example, distributed Universal Personal Communications (UPT) [CCI91, HF92a, HF92b]

UPT is an intended advanced service that allows customers to be contacted over the telecommunications network via a logical identifier rather than a physical phone numbers. Such a level of indirection between the identity of a telephone user and the current equipment that is being utilised by that user supports communication in the presence of user mobility [O'B92]. Such a service requires a suitably flexible interface to a billing subsystem. This subsystem needs to have high integrity data and should be transactional. Other components of the UPT system have different requirements. For example, the network connection management [FLNP92] can forgo data integrity for improved response time. The TINA distributed programming environment will address both these and other aspects of applications such as UPT. As will become apparent later in the thesis the performance of the transactional components of such applications could be improved by the results of this thesis.

\textsuperscript{12}The sponsors of this research.
Chapter 3

Hermes/ST and Granularity

This chapter introduces the linguistic features of the Hermes/ST Distributed Programming Environment. Hermes/ST has several novel linguistic constructs and mechanisms [FHR94, FHR93c, FHR93a, FHR93b]. These constructs are introduced so that later chapters have a concrete distributed programming environment to reference. Section 3.1 introduces as much of the Hermes/ST distributed object model as is relevant.

Having introduced Hermes/ST, this chapter also introduces several system related factors that are important in the development of a distributed application. Distributed applications, like their traditional counterparts, are evaluated according to several measures. One such important system measure is throughput. This chapter proposes that system throughput is effected by, amongst other factors, the granularity of concurrency control that is employed by an application. Section 3.2 introduces the interdependence between throughput and concurrency control granularity.

3.1 Hermes/ST Distributed Object Model

The Hermes/ST distributed object model is implemented in Smalltalk [GR89]. As such, its design has tried to adhere as closely to the Smalltalk model as possible. Naturally, the extended requirements of a distributed programming environment have caused deviations from the Smalltalk model.

Hermes/ST has a uniform object model [GR89]. All entities in a Hermes/ST application are objects. Hermes/ST objects are described by classes using single inheritance from the base class HermesObject. This mimics the Smalltalk approach where all objects in the system are instantiations of some subclass of the base class Object. Hermes/ST objects interact via message passing, just as is done in Smalltalk.

The following sections introduce the ontology of Hermes/ST objects, the different kinds of Hermes/ST objects, the use of constant objects, how to create Hermes/ST objects, how Hermes/ST objects interact, and how to destroy Hermes/ST objects. Illustrative examples in this section are taken from relevant components of the distributed banking example that was introduced in section 2.7.

3.1.1 Ontology

Hermes/ST objects have state and behaviour. This also mimics the Smalltalk object ontology. In Smalltalk, objects have state that is modelled through various sorts of variables, and behaviour that is expressed via methods [GR89].
3.1.1.1 State

Hermes/ST state is modelled through the use of Hermes/ST instance variables. Variables always refer to the values that they contain. This property is sometimes referred to as uniform reference semantics [Mey88]. There are no containment constructs equivalent to C's struct definition. The declaration of an object's state is made in a class. Such a declaration for an account of the distributed bank could be as follows.

```
class HermesSTAccount
superclass HermesObject
instance variables name
              address
              balance
```

The class HermesSTAccount declares that the Hermes/ST object instantiated (refer section 3.1.4) by this class will have three instance variables. These are name, address and balance. As is the case with Smalltalk variables, Hermes/ST variables are not statically typed.

3.1.1.1.1 Accessing State  Hermes/ST instance variables are always accessed via special Hermes/ST access methods. Each variable has a read access method and a write access method. By convention, the read access method has the same name as the variable. Thus, for example, the instance variable balance is always read via the balance method. By a similar convention, write accesses to variables are always made through a method with the same name as the variable concatenated with a :. Hence, to write a value to the instance variable balance one would use the balance: <value> method.

Hermes/ST uses the extra level of indirection supplied by variable accesses to ensure that any necessary concurrency control, state restoration and persistence handling is performed on Hermes/ST objects. The decision to introduce access methods conventions was taken in order to avoid having to modify the Smalltalk compiler.

3.1.1.2 Behaviour

Hermes/ST object behaviour is expressed via Smalltalk methods. Hermes/ST uses the same method declaration syntax as the host Smalltalk system. Methods consist of statements. Statements follow the usual Smalltalk syntax. Exceptions to this convention are introduced as they are needed. As an example, consider the following method that is defined in the class HermesSTAccount.

```
toTranscript
"I am a method of the HermesSTAccount class
When called, I write the value of each of my variables to the system transcript"

    Transcript show: 'name -> ', self name printString, ' address -> ', self address printString, ' balance -> ', self balance printString; cr
```

1. For efficiency reasons, the reference to some values can encode the value. For example, an integer reference usually contains the value of the integer.

2. Other, more systematic approaches are possible. For example, the semantics of read and write accesses could have been built into the Smalltalk language. This would have necessitated alterations to the Smalltalk compiler. Alterations to the Smalltalk compilation process are outside the scope of Hermes/ST.
HermesST Account to Transcript displays the state of an account to a system window (the Transcript). The method to Transcript uses the read access methods of section 3.1.1.1 to read variables. Text between "" form comments. Hermes/ST extends the use of comments for various purposes. These extensions are also introduced as needed.

### 3.1.2 Hermes/ST Object Kinds

Unlike Smalltalk, Hermes/ST supports different kinds of objects. There are transactional and volatile objects.

**Transactional**: These objects are so named because they allow the Hermes/ST user to develop applications that exhibit transactional behaviour. Transactional objects perform concurrency control, state restoration and persistence handling. Concurrency control is based on strict two phase locking (refer section 2.3.1.3) as extended to deal with nested transactions (refer section 2.4.1). More detail is presented in section 3.1.5. State restoration is performed is based on an update-in-place strategy (refer section 2.3.2.1). Before changing object state, a version (memory copy) of the previous state is created. Version stacks are used (see section 2.4.2) to deal with nesting. Section 4.4.2 discusses the granularity at which this occurs. Persistence handling is achieved via the two phase commit protocol (refer section 2.3.2.2). Committed object versions are made persistent via a logging mechanism (see section 2.3.2.2). Section 4.4.3 discusses the granularity at which persistence handling occurs.

**Volatile**: These objects do not perform any concurrency control, state restoration or persistence handling. These objects are therefore much less costly to use than a transactional object. A transactional structure may wish, for example, to maintain a volatile pictorial representation of itself to display to an application user. This representation, does not need transactional semantics and can be more efficiently implemented using volatile objects.

### 3.1.3 Constant Objects

All other objects in the Hermes/ST system are considered to be constant objects. Constant objects are immutable. They can be used to represent entities such as numbers, strings and other non changing parts of the application model. Constant objects are sometimes referred to as values [Atk91]. Constant objects, like volatile objects, have no concurrency controllers, no state restoration handlers and no persistence handlers. One important distinction between a constant and a volatile object is that a volatile object is remotely accessible while a constant object is not. (See section 3.1.5.2).

Constant objects are created, interacted with and garbage collected according to the syntax and semantics of the host Smalltalk system. Hermes/ST automatically makes the distinction between Hermes/ST objects and constant objects. For the remainder of this thesis, unless otherwise stated, the collective term Hermes/ST objects does not include constant objects. The following code fragment shows the creation of a constant string in Hermes/ST

```
' This is a constant object. In this case a string'.
```

whereas
creates a constant array consisting of a reference to the constant string 'Fazzolare', a reference to the constant string 'University of Wollongong' and a reference to the constant integer 100.

3.1.4 Hermes/ST Object Creation

Hermes/ST object creation, like object instantiation in Smalltalk, is performed by sending a special instance creation message to an appropriate object. In Smalltalk, the instance creation method is some variant of the new or new: message and is sent to a class object. In Hermes/ST, object creation is performed by sending an instance creation message to a Hermes/ST object. This message is the containNew message. Hermes/ST deviates from the Smalltalk object model. Hermes/ST objects are structured into tree topologies via nested encapsulation (see section 5.2.7). Smalltalk has a flat [GR89] object structure.

To express the extra features that a Hermes/ST object may require the containNew message can be qualified with various instance creation parameters. Multiple instance creation parameters can be cascaded [GR89] with the instance creation message via the Smalltalk cascade operator ;. Hermes/ST object creation returns a Hermes/ST pointer which is used to interact with the created Hermes/ST object.

The following code fragment shows the minimum specification for the creation of a Hermes/ST object.

```
HermesSystem containNew; hermesClass: HermesSTAccount.
```

The containNew message tells Hermes/ST to create a new object that is contained by the system. All Hermes/ST objects are contained by one other Hermes/ST object. The reasons for this containment will be made clear in chapter 5. By default, Hermes/ST creates volatile objects. The cascaded hermesClass: instance creation parameter names the class from which the Hermes/ST object is to be instantiated. In this example, Hermes/ST creates a volatile instance of HermesSTAccount with all instance variables set to nil.

To create a transactional object, the following code could be used.

```
HermesSystem containNew; kind: ^transactional; hermesClass: HermesSTAccount.
```

Specifying the kind: instance creation parameter to be #transactional and cascading it with the containNew message will cause Hermes/ST to create a transactional object. This object is recorded in an object store [FHR93c] and has concurrency controllers, state restoration handlers and persistence handlers attached. All instance variables are initialised to nil. Because the creation of transactional objects is frequent in Hermes/ST there is a shorthand notation for kind:#transactional. Thus, the above example has the same effect as

```
HermesSystem containNew; transactional; hermesClass: HermesSTAccount.
```
CHAPTER 3. HERMES/ST AND GRANULARITY

It is often desirable to instantiate an object with its instance variables initialised to values other than nil. Hermes/ST performs object initialisation via the `args:` instance creation parameter. The following code fragment demonstrates this.

```plaintext
HermesSystem
  containNew;
  args: (Array new
    with: 'Fazzolare'
    with: 'University of Wollongong'
    with: 100);
  hermesClass: HermesSTAccount
```

The argument of the `args:` parameter is passed to the `initializeWithArguments:` of the newly created object. `initializeWithArguments:` is defined by the HermesObject class, and can be overridden by subclasses. For example, `HermesSTAccount` could override `initializeWithArguments:` as follows.

```plaintext
initializeWithArguments: args

  args size > 0
  ifTrue:
    [self name: args first.
     self address: args second.
     self balance: args third]
```

`initializeWithArguments:` assigns references for each element of its argument array `args` to the variable that should contain them. Thus the instantiated `HermesSTAccount` object has its `name` variable referring to `‘Fazzolare’`, its `address` variable referring to `‘University of Wollongong’` and its `balance` variable referring to `100`.

Hermes/ST is a distributed programming environment. Objects can be created on any node\(^3\) in a Hermes/ST network. The following code demonstrates how to specify the node on which an object is to be created.

```plaintext
HermesSystem
  containNew;
  location: #harpo;
  hermesClass: HermesSTAccount.
```

The `location:` instance creation parameter specifies the name\(^4\) of the node on which an object is to be created. In this case the object will be created on the node named `harpo`. If the `location:` instance creation parameter is omitted then the Hermes/ST object creation defaults to the current node.

Hermes/ST also allows objects to be referenced by system-wide aliases. Aliases are stored in the Hermes/ST name server. The following code fragment demonstrates the creation of a Hermes/ST object with an alias.

\(^3\)See section 2.1.

\(^4\)The departmental computers use Internet Domain Name Serving. Amongst many other nodes, there are the “harpo”, “chico”, “groucho”, “zeppo”, and “karl”
HermesSystem
   containNew;
   alias: #demonstrationAccount;
   hermesClass: HermesSTAccount.

The alias: instance creation parameter specifies a symbol that can be used to gain access to an object in a system-wide, location-independent manner. In this case the Hermes/ST object is known by the alias #demonstrationAccount.

Besides containNew, volatile:, args:, location: and hermesClass: there are many other instance creation parameters in Hermes/ST. These will be introduced as necessary throughout this thesis. Most instance creation parameters are orthogonal and can be combined arbitrarily, as the following code fragment demonstrates

HermesSystem
   containNew;
   transactional;
   location: #harpo;
   alias: #remoteDemonstrationAccount;
   args: (Array with: 'Demo' with: 'Demo Address' with: 0);
   hermesClass: HermesSTAccount

The transactional object known by the alias #remoteDemonstrationAccount is created on the node harpo, is initialised to have the name 'Demo', the address 'DemoAddress' and balance 0.

3.1.4.1 Cloning Instance Creation Parameters

The structuring of Hermes/ST objects as trees implies that each Hermes/ST object has a parent object. In the absence of explicit instance creation parameters a newly created Hermes/ST object clones the values of instance creation parameters from its parent object. For example, if the kind: instance creation parameter is omitted then a Hermes/ST object is created according to its parent’s kind. Thus, if the parent object is transactional then the object being created will also be transactional.

3.1.5 Hermes/ST Message Passing

In Smalltalk, objects invoke the methods of other objects via message passing. Hermes/ST objects also invoke the methods of other objects via message passing. Constant objects are invoked via the underlying Smalltalk implementation whereas Hermes/ST objects are invoked via the Hermes/ST system.

Hermes/ST message passing consists of a sender object and a receiver object. A sender object is said to “send a message to” or “invoke a method of” a receiver object. The receiver object executes the sent message. Hermes/ST messages can have arguments and return results thus allowing information to be shared by objects in a Hermes/ST system. Hermes/ST messages can also have various message parameters. Message parameters are needed to express the richer interaction model that is needed by a distributed programming environment. Message parameters are distinct from arguments. They are used to

A conceptual, Hermes/ST system object is the ancestor of every object

See section 3.1.5.2 for the semantics of information sharing
supply meta information about Hermes/ST messages. For example, one message parameter is the **transactional:** parameter. A message that is qualified by a **transactional:** true parameter is executed in a transactional fashion. As was the case with the Hermes/ST object creation of section 3.1.4 these message parameters are orthogonal and can be cascaded. Other message parameters will be introduced as they are needed throughout this thesis.

3.1.5.1 The Semantics of Hermes/ST Messages

Because the Hermes/ST message passing model is novel an overview of its semantics is presented here. The reader interested in greater detail is referred to [Hum94].

A *message* is specified by a *receiver* object, *message parameters*, a *method name* and *arguments*. Message parameters describe the *kind* of a message. A message can be synchronous or asynchronous), (transaction creating or non-transaction creating) and has a *lock type*. Every message can access (read and write) the receiver object’s variables (refer section 3.1.1.1.1) and send other messages (sub-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*.

Figure 3.1 is an example of a message tree. 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 3 and 5 are submessages of the message labeled with 2. To refer to a message such a number is prefixed by an upper case M. For example, the root node is referred to as message M1. 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. Therefore, there is conceptually an arc leading to the root node (not shown in figure 3.1).

![Figure 3.1: An example message tree.](image)

The 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.

7Italics signify the definition of a Hermes/ST meaning
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 $M_5 \leq M_5$ but $M_5 \ngeq M_3$. Conversely, $M_4 \geq M_2$ and $M_4 \geq M_4$ but $M_2 \ngeq M_5$.

A message path is a data structure that describes the message parameters and the position of a particular message in a message tree. A message path is a non-empty sequence of message path elements. A message path element is denoted by prefixing the message number with a lower case m. For example, the message path element for $M_3$ is $m_3$ and the message path for $M_3$ is $m_1, m_2, m_3$.

### 3.1.5.1.1 Transactions

Transactions are specified by cascading the transactional: true (or, in shorthand, trans) message parameter with a Hermes/ST message. Figure 3.2 represents the example message tree when $M_2$ has been specified as being transactional. The trans message parameter causes a transaction to be created before the first line of the method named by $M_2$ is executed. This transaction is terminated after the execution of the last line of the method named by $M_2$ is completed. The semantics of transactions were introduced in section 2.3. $M_2$ is called a transaction creating message. $M_2$, $M_3$, $M_4$, and $M_5$ are all descendants of $M_2$ and are referred to as transactional messages. Thus a transaction in Hermes/ST is a specific subtree of a message tree and refers to a set of messages.

Hermes/ST transactions that invoke methods of constant or volatile objects do not perform concurrency control, state restoration or persistence handling on these objects. Hermes/ST transactions that invoke methods on transactional objects perform concurrency control, state restoration and persistence handling for the life of the transaction. At the completion (commit or abort) of the transaction, all locks are released, all state
restoration information is discarded and any committed changes to the transactional object are made persistent.

3.1.5.1.2 Nesting Transactions A transactional message can send another transaction creating message. The resultant transaction is called a nested transaction. In figure 3.3, messages $M_3$ and $M_4$ form a nested transaction. The transaction created by $M_2$ is referred to as a top-level transaction. The semantics of nested transactions were introduced in section 2.4. In Hermes/ST, nested transactions do not perform early writing.

![Figure 3.3: An example message tree containing a nested transaction.](image)

3.1.5.1.3 Threads Analogously to transactions, threads are specified by cascading the `isSynchronous:false` (or, in shorthand, `async`) message parameter with a Hermes/ST message. Unless the async message parameter is specified with a message then the message is executed synchronously. A synchronous message execution is one in which the execution of the parent message is suspended until the message has completed and returned any results. Figure 3.4 represents the example message tree when $M_2$ has been specified as being asynchronous. The async message parameter causes an asynchronous message execution. An asynchronous message execution is one in which the execution of the parent message is not suspended during the execution of the child. Asynchronous messages cause a thread to be created before the first line of the method named by $M_2$ is executed. This thread is terminated after the execution of the last line of the method named by $M_2$ is completed. In figure 3.4 $M_2$ is called a thread creating message. $M_2$, $M_3$, $M_4$ and $M_5$ are all descendants of $M_2$ and belong to this same thread. A Hermes/ST thread is also a specific subtree of a message tree and refers to a set of messages.

Hermes/ST threads that invoke methods of constant and volatile objects do not perform concurrency control, state restoration of persistence handling. Hermes/ST threads that invoke methods on transactional objects perform concurrency control, state restoration and persistence handling for the life of the message only. Thus at the completion of the method, all locks are released, all state restoration information is discarded and any
changes to the transactional object are made persistent. Threads perform concurrency control, state restoration and persistence handling so that they do not interfere with the semantics of transactions.

3.1.5.1.4 Nesting Threads A message that is part of a thread can send a thread creating message. The resultant thread is called a nested thread. In figure 3.5 messages $M_3$ and $M_4$ form a nested thread. The thread created by $M_2$ is referred to as a top-level thread. Nested threads do not extend the semantics of threads. A nested thread performs the same concurrency control, state restoration and persistence handling as a top-level thread.

3.1.5.1.5 The Semantics of Combined Message Parameters Hermes/ST message parameters can be combined arbitrarily when invoking a method. The same cascade mechanism that was used for the instance creation parameters applies. Therefore, disregarding lock message parameters, each message can be transaction creating or non-transaction creating, synchronous or asynchronous and nested or non-nested. The manner in which Hermes/ST treats these eight kinds of messages follows:

1. A synchronous non-transaction creating message with no ancestor transaction is performed as part of the ancestor thread.

2. A synchronous non-transaction creating message with some ancestor transaction is performed as part of the ancestor transaction.

3. A synchronous transaction creating message with no ancestor transaction creates a synchronous top-level transaction.

4. A synchronous transaction creating message with some ancestor transaction creates a synchronous nested transaction. A synchronous nested transaction causes

---

*In contrast, nested transactions extend the semantics of transactions. I.e. They allow partial failure and more concurrency*
the suspension of the ancestor transaction until the synchronous transaction has completed.

5. An asynchronous non-transaction creating message with no ancestor transaction creates a nested thread. A nested thread executes in parallel with other threads.

6. An asynchronous non-transaction creating message with some ancestor transaction creates an asynchronous thread of the ancestor transaction. This kind of message is interesting in two ways. Firstly, it allows the creation of parallelism without the expense of a nested transaction. Secondly, the ancestor transaction is not suspended during the execution of the child.

7. An asynchronous transaction creating message with no ancestor transaction creates a top-level transaction that executes in parallel with the invoking thread.

8. An asynchronous transaction creating message with some ancestor transaction creates a nested transaction that is executed in parallel with the ancestor transaction.

3.1.5.1.6 Nomenclature of Threads and Transactions As defined above, a thread is a set of synchronous messages that have a common ancestor thread creating message and no other thread creating message. A thread may or may not contain transaction creating messages. The term thread is used when such a collective term is useful. If the transactional properties of a thread are important then the terms transaction or transactional thread are used. The two terms refer to different sets of messages. For example, a transaction may span several threads. The semantics of combined message parameters (see section 3.1.5.1.5) allow this to happen.

3.1.5.1.7 Scheduling Mechanism The Hermes/ST scheduling mechanism is detailed in [Hum94]. It correctly schedules any nested or top-level, asynchronous or synchronous transactional and non-transactional messages. The scheduling mechanism is based on extensions to Moss' [Mos85] two phase locking in nested systems.
Some key points to note are:

- Asynchronous nested transactions are permitted and serialised. This differs from other systems such as Argus [Lis82, LS83, LCJS87, Lis88]. In Argus, nested transactions, although asynchronous with respect to each other, are synchronous with respect to the parent transaction.

- Hermes/ST, unlike systems such as Argus, allows the introduction of parallelism within a transaction without the need to create a nested transaction. Such asynchronous transactional threads do not require a convolution of message invocations in order to express parallelism [Hum94].

- Non-transactional threads are not two phase, only individual non-transactional messages are serialised (see section 3.1.5.1.3). Non-transactional messages are said to be synchronised [Hum94]

3.1.5.1.8 Message Exceptions Threads and transactions, which are sets of messages in Hermes/ST either complete successfully or cause exceptions to be raised. Transactions can trap these exceptions and cause alternative threads or transactions to be executed (see section 3.1.7 for an example). Thread exceptions, as might be caused by a communications failure, cannot currently be trapped and cause a Smalltalk debugger window to be displayed. The successful completion or termination via exception of either a thread or a transaction is henceforth called thread completion.

3.1.5.2 The Semantics of Object Sharing

In Smalltalk, objects share information by invoking the methods of objects with other objects as arguments. Methods can also return objects to the invoking method. As Smalltalk employs uniform object reference, such shared information is passed by reference [Mey88, KR88]. This means that rather than copying objects in and out of method invocations, objects are passed by passing a reference to the object. Any changes made by a method to a passed object are effective after the method completes. This is not the case when information is copied in and out (passed by value).

Hermes/ST objects also share information by invoking methods and returning values from these methods. Hermes/ST objects can share information in the form of other Hermes/ST objects or constant objects. As was the case in Smalltalk, shared objects are also passed by reference in Hermes/ST. The Hermes/ST pointer implements the Hermes/ST object reference, including remote Hermes/ST objects. Therefore, in a Hermes/ST system there may exist a many to one relationship between Hermes/ST pointers and Hermes/ST objects.

Since constant objects are immutable the system is free to pass such objects by value. For performance reasons Hermes/ST uses pass by value semantics for constant objects involved in a remote object invocation. The constant object is deep copied [Mey88] from the calling node to the remote node. The Hermes/ST system uses built in features of the Smalltalk system to deep copy constant objects. If the programmer wishes to override the default system copying behaviour, this can be achieved by redefining the packOn: method of the constant object's Smalltalk class. It is safe to have multiple copies of a constant object because a constant object, by definition cannot be altered.

For example, the following code fragment makes the balance variable of #remote-DemonstationAccount refer to the constant object 500.
remoteObjectInteraction

"Assigns a constant object to a remote Hermes/ST object"

| account |
account := NameServer hermesPointerOf: #remoteDemonstrationAccount.
account balance: 500

NameServer hermesPointerOf: #remoteDemonstrationAccount returns a Hermes/ST pointer to the remote demonstration account. In the statement account balance: 500, the constant object 500, is deep copied and transferred to the remote demonstration account node. At this node the constant object is automatically reconstructed by Hermes/ST. This reconstructed 500 is stored into the balance variable.

3.1.5.3 The hermesSelf Hermes/ST Pointer

Every Hermes/ST object contains a Hermes/ST pointer to itself. This pointer can be accessed via the hermesSelf variable. This is meant to be analogous to the Smalltalk self pseudo variable [GR89]. To demonstrate a use of the hermesSelf variable consider the following method.

factorial: number

"Illustrates a use of the hermesSelf pointer."

number = 1
ifTrue: [1]
ifFalse: [number * (hermesSelf factorial: number — 1)]

Such a method can be defined by any Hermes/ST object to compute the factorial of a number.

3.1.6 Message Parameter Specification

Hermes/ST provides two ways of specifying message parameters.

1. Message parameters can be specified at method invocation. For example, assuming that account has been assigned thus

account := NameServer hermesPointerOf: #remoteDemonstrationAccount.

Then the following piece of code will create an asynchronous transaction that updates the remote demonstration account to contain $500.

account trans; async; balance: 500.

^account balance:500 is an example of a Hermes/ST write access method as discussed in section 3.1.1.1.1
2. Message parameters can be specified at method declaration. Hermes/ST currently allows the definition of message parameters in the comment of the method they apply to. This decision was made in order to avoid changing the Smalltalk compiler. As an example consider the following piece of code

```
messageParameterSpecificationAtMethodDeclaration
```

```
  InvocationScheme
  async
trans
```

↑ account balance: 500.

Section 4.2.4 contains the syntax of method-lock associations and chapter 5 contains several examples of these declarations.

To determine the correct message parameter value to use, Hermes/ST uses the following precedence rules. If a message parameter is specified as part of the method invocation then that value is used. If not, then if the parameter is specified as part of the method definition then that value is used. If the message parameter still has no value then the receiver object’s superclasses are searched for the same method with the parameter declared. If there is still no value for a parameter then system defaults are used. The system defaults are that a message be synchronous, non-transactional and not acquire any locks.

3.1.7 Extra Transaction Operations

Due to the possibility of deadlock and independent failure of individual nodes in the Hermes/ST system, extra linguistic constructs are made available to the programmer. Transaction abortion is based on an user-specified timeout. A timeout is another message parameter and can be specified at method declaration or method invocation. The syntax is `timeOut: <value>`, where `value` is the number of seconds to wait before timing the message out.

Transactions can also be explicitly aborted by the applications developer using `abortToplevel` and `abortCurrentTransaction` messages. `abortToplevel` causes the transaction creating message that is closest to the root of the message tree (if any) to be aborted. `abortCurrentTransaction` causes the transaction creating method that is the closest ancestor of the current message (if any) to be aborted.

As introduced in chapter 2, subtransactions are allowed to abort without necessarily causing the abort of the invoking parent transaction. Hermes/ST supplies a linguistic mechanism to allow the user to specify an alternative execution path if a transaction aborts. The following code demonstrates

```
failureAlternatives
```

```
  Illustrates the use of performIfFail: .
```

```
| primary backUp |
primary := NameServer hermesPointerOf: #primaryCopy.
backUp := NameServer hermesPointerOf: #backUpCopy.
primary
```
trans;
performIfFail: [backUp trans; writeBackUp];
writePrimary

If, for any reason (communication failure, node crash, deadlock), the transaction to write the primary object fails then a back-up object is updated instead.

Finally, transactions need to deal with the creation and destruction of objects by transactions that commit or abort. Objects that are created during a transaction must be deleted if the transaction aborts, and objects that are deleted during a transaction must be remade if the transaction aborts. The Hermes/ST system is responsible for ensuring these semantics.

3.1.8 Hermes/ST Object Deletion

Unlike Smalltalk, Hermes/ST does not perform automatic garbage collection for Hermes/ST objects. Constant objects are garbage collected in the usual manner. Hermes/ST objects are deleted by sending a delete message to an object. The following code fragment shows the creation, followed by the destruction of a transactional account.

```smalltalk
| account |
account := HermesSystem
    containNew;
transactional;
hermesClass: HermesSTAccount.
account delete
```

Object deletion removes the object from the object store and if the object has an alias, the alias is removed from the Hermes/ST name server.

3.2 Hermes/ST Objectives

Hermes/ST is both a distributed and object-oriented programming environment. As such Hermes/ST has adopted objectives from both object-orientation and distributed systems. Amongst these objectives are system performance, reuse, incremental development and ease of specification.

3.2.1 Performance

The “performance” of a system can be gauged by several, possibly conflicting, measures. For example, throughput, response time and fairness are all criteria used in traditional operating systems literature [Tan87, SPG91]. Throughput is a measure of the number of activities that can be performed by a system in a given unit of time. Response time is a measure of the amount of time taken to execute a single task. Fairness is a measure of the allocation of system resources amongst various competing activities.

Various systems place differing degrees of importance on these criteria and indeed, these criteria can be contradictory. For example, fairness can be improved by some form of pre-emptive scheduling [Tan87, SPG91] but pre-emptive scheduling introduces context...
switching [Tan87, SPG91] which in turn degrades throughput. For the purposes of this thesis the performance of the Hermes/ST system will be measured by throughput. Other performance measures are important, but are outside the scope of this work.

Throughput in Hermes/ST can be defined as the number of Hermes/ST threads that are executed per unit of time. It can be measured as

\[ T = \frac{\text{number of threads completed}}{\text{Time taken to complete those threads}} = \text{threads/sec} \]

where the larger the value of \( T \) the better. There are many factors that contribute to the throughput of a distributed system such as Hermes/ST [Mul93, BHG87]. Amongst these are: the number of processors available; the number of instructions to be executed by each thread; the multi-programming level; the nature of the processor utilisation by each thread; the type and granularity of concurrency control, state restoration and persistence handling mechanisms used for each thread; and the local and remote object invocation mechanisms used by threads.

In distributed programming environments that support transactions, system performance can be severely impacted by the granularity of concurrency control [GR93, BHG87, Mul93]. This thesis is mostly concerned with issues related to the granularity of concurrency control in such systems. Some of the aforementioned throughput factors are more or less related to concurrency control granularity. For example, the number of processors is unrelated to the granularity of concurrency control employed. As another example, the multi-programming level (MPL) is slightly more related to concurrency control. The relationship being that a poorly chosen granularity of concurrency control can severely restrict the MPL. The rest of this section discusses aspects of those throughput factors which have some relationship to the type of concurrency control in a distributed system.

3.2.1.1 The Relationship between Concurrency Control Granularity and Throughput

The granularity of concurrency control for an application refers to the number of concurrency controllers used by that application. An application that uses many concurrency controllers is said to have a fine granularity of concurrency control whereas an application that uses only a few concurrency controllers is said to have coarse granularity of concurrency control [BHG87].

3.2.1.1.1 Lock Management Overheads

The granularity of concurrency control affects throughput via the lock management overheads introduced by concurrency controllers. A fine granularity of concurrency control will entail the use of many locks. A coarse granularity of concurrency control will entail the use of fewer locks.

Each lock in a system introduces lock management overheads into the system. Lock management overheads include the time taken for the creation, and deletion of each lock as well as the time taken to test for conflicts between this and other locks. The more lock management overheads there are, the longer each thread in the system takes to execute. This, in turn, affects the throughput of the system.

3.2.1.1.2 Lock Contention

Hermes/ST objects have variables that represent state and methods that implement behaviour. Methods are invoked by sending messages that are grouped into threads. Threads can be transactional or non-transactional. Threads concurrency control is based on locking. Transactions are serialised via extensions to the well known, strict two phase locking mechanism. Two phase locking is a form of concurrency control that implements serialisability by delaying threads that request conflicting

\[^{10}\text{See section 3.1.5.1.6}\]
locks. Lock conflict is expressed via a compatibility matrix. For example, the lock compatibility matrix for standard read/write locking was shown in figure 2.1. To recap, read locks conflict with write locks, and write locks conflict with both read and other write locks. The delays caused by conflicting locks can be long when transactional threads are involved. This is because strict two phase locking requires locks to be held until the completion of a transactional thread. In Hermes/ST, the delays caused by locking conflicts are referred to as *lock contention*. Lock contention affects throughput by reducing the level of allowable parallelism in a system.

For the purposes of this thesis, two types of lock contention will be distinguished.

1. **Necessary contention** occurs when two threads must access the same state in a conflicting manner. Parallel writes to the same variable are an example of necessary contention. Necessary contention is unrelated to the granularity of concurrency control.

2. **Unnecessary contention** occurs when the granularity of concurrency control is too coarse. For example, consider a Hermes/ST object that contains many variables. If each variable is individually locked, then threads that write access different variables can be executed in parallel. Conversely, if the entire object is locked, then even though each thread write accesses a different variable, no concurrent thread execution is allowed. Each thread must wait until the object lock is released by any thread holding the object lock.

Unnecessary lock contention is related to the granularity of concurrency control. If a fine granularity of concurrency control is employed then threads exhibit less unnecessary contention. Conversely, if a coarser granularity of concurrency control is employed then threads exhibit more unnecessary contention.

### 3.2.1.1.3 Deadlock

Deadlock occurs when two or more threads have a cycle in their waits-for-graph (see section 2.3.1.3.1). Deadlocks in the Hermes/ST system cause transactions to be aborted. These abortions represent wasted work and affect the throughput of a system.

Deadlock is related to the granularity of concurrency control. Changing the granularity of concurrency could introduce some forms of deadlock and relieve other forms of deadlock. For example, consider a Hermes/ST object that contains variables *a* and *b* which can be locked individually. Furthermore, consider the existence of two threads *t1* and *t2* that wish to write to both *a* and *b*. If *t1* acquires a write lock on *a*, and *t2* acquires a write lock on *b*, then there will be a cycle in the waits-for-graph of both threads and deadlock will occur. No such deadlock could occur if the granularity of concurrency control was at the level of the object.

As a counter example, consider two Hermes/ST objects that are object level locked. As in the previous example, both objects contain variables *a* and *b* and are to be visited by two parallel threads. This time thread *t1* wishes to write to variable *a* of both objects and thread *t2* wishes to write to variable *b* of both objects. In this case a deadlock can occur if thread *t1* writes to the first object and *t2* writes to the second object. No such deadlock would have occurred if the granularity of concurrency control was at the variable level.

The amount of deadlock exhibited by an application and its impact on the distributed system depends on the application and its interaction with the distributed system [GR93]. Typically, applications are designed so that the probability of deadlock is low [GR93, BHG87]. Section 6.2.1.6 gives an example of how deadlock related to the granularity of concurrency control can be dealt with by Hermes/ST.
3.2.2 Incremental Development

In Hermes/ST, incremental development is the ability to develop a system with ever increasing layers of complexity. Its aim is to facilitate the development of complex systems\textsuperscript{11}. For example, the distributed bank application has several layers of complexity. It is desirable to structure the accounts of a particular branch in the bank so that they can be accessed reasonably efficiently. This is achieved by structuring accounts as a binary tree. It is also desirable to implement accounts in such a manner that the information contained in them will be able to survive power outages. This is achieved by making accounts transactional objects. It is further desirable to perform operations that change the value of accounts in such a manner that the failure of a branch will not lead to an incorrect balance being recorded in an account. This is achieved by performing such operations in a transactional manner. Naturally, an incremental development strategy should entail as few specificational changes at each step as possible.

An incremental development strategy is particularly advantageous in complex systems such as distributed programming environments. Incremental development allows the application developer to divide and conquer individual system components. For example, the distributed bank's branch account tree structure can first be prototyped as a traditional sequential binary tree class. When this prototype is debugged, it can be extended to have persistent data and perform its methods in a transactional manner.

3.2.3 Reuse

In Hermes/ST, reuse refers to the ability to re-instantiate one class in various applications. For example, the binary tree class of a distributed bank should be able to be reused by another application that needs to structure its own types of objects. Furthermore, in a distributed system, it should be possible to re-instantiate a class with various kinds of behaviour. For example, the same binary tree class should be able to be instantiated with transactional or volatile constituent objects. Naturally, the reuse of a class should entail as few specificational changes to the class as possible.

Reuse is particularly advantageous in complex systems such as distributed programming environments. Reuse allows the application developer to build on the efforts of other developers. These efforts may have entailed many hours of development and testing. A binary tree class may contain many optimisations such as balancing. It is undesirable to repeat this work for each application.

3.2.4 Ease of Specification

In Hermes/ST, ease of specification refers to the ability of an application developer to describe an application at a high level of abstraction in an error free manner. A message parameter is a good example of such a specification. The existence of the \texttt{trans;} message parameter allows method invocations or threads to be specified to be transactional. Error free specifications can be achieved to various degrees. For example, the minimal lock approach of chapter 4 ensures error free lock specifications by inferring the lock specification from the method definition. As another example a lesser degree of error free specification is ensured by the \texttt{trans;} message parameter. It is possible for the application developer to accidentally omit a \texttt{trans;} message parameter but it is not possible for the application developer to have an unterminated transactional thread. An unterminated transactional thread would be possible if some form of "\texttt{beginTransaction}, \ "\texttt{endTransaction}" construct were used.

\textsuperscript{11}Including distributed systems.
Chapter 4

Implicit and Explicit Concurrency Control

4.1 Introduction

This chapter introduces Hermes/ST's implicit and explicit concurrency control. Explicit concurrency control is achieved via programmable locks. Implicit concurrency control is achieved via minimal locking.

Section 4.2 introduces the Hermes/ST programmable lock approach. Programmable locking allows the application developer to specify efficient lock specifications that support reuse and incremental development. Such specifications are explicitly denoted by the applications developer and allow individual methods to trade-off performance versus the number and type of programmable locks used. Section 4.3 evaluates the Hermes/ST programmable lock approach.

Section 4.4 introduces minimal locking. Minimal locking allows the development of applications that exhibit high parallelism. This parallelism is obtained implicitly from the method specification. No programmable lock like concurrency control specifications are required. However, minimally locked structures use many concurrency controllers which can adversely affect the performance of applications. Chapter 5 presents an approach to remedying this and other problems. Minimal locking is evaluated in section 4.4.5.

Another problem with minimal locking is that not all applications can be specified via minimal locking. To address this problem minimal and programmable locking are combined in section 4.5 to produce dual locking. Dual locking, whilst using even more locks than minimal locking, allows non-trivially synchronised applications such as a cheque spooler to be specified in an elegant manner. Dual Locking is evaluated in section 4.5.1.

4.2 The Programmable Lock Approach

In the programmable lock approach, locks are used to protect a group of objects. This group of objects is henceforth referred to as a thread's resource set. If the resource sets of two threads are disjoint then both threads may proceed in parallel. If resource sets intersect then a thread can proceed only if the message it is attempting to execute "is compatible with" all other messages being executed on the intersecting resource set.

For example, consider depositing money into an account of the distributed bank application. Recall that accounts are stored at each branch in a transactional binary tree. A branch object is implemented by the class HermesSTBranch. The binary account tree

\footnote{As defined by a compatibility matrix. See, for example, section 3.2.1.1.2.}
is implemented by the HermesSTBinaryTree and HermesSTBinaryTreeNode classes. Accounts are implemented by the HermesSTAccount class.

Deposit messages that are sent to different accounts write to different balance variables but read common nodes of the accounts tree. As read accesses are compatible with each other, and the write accesses are to disjoint accounts, deposit messages to different accounts can be executed concurrently. One way that Hermes/ST allows the specification of such behaviour is through programmable locks. Programmable locks deal with thread synchronisation and as such are only applicable to transactional objects in Hermes/ST.

### 4.2.1 The Programmable Lock Hierarchy

Programmable locks form a hierarchy with the abstract class ProgrammableLock as the root. ProgrammableLock supplies two methods isSchedulable and isCompatibleWith:. ProgrammableLock isSchedulable and ProgrammableLock isCompatibleWith: both return true. Subclasses can override these methods to provide customised behaviour.

The method isSchedulable allows a programmable lock to make scheduling decisions that depend on the state of a transactional object. The isSchedulable method is useful for expressing behaviour such as that required by a bounded buffer (see section 4.2.4.2). The method isCompatibleWith: defines a programmable lock's compatibility predicate (refer chapter 3.2.1.1.2) with other programmable locks.

Hermes/ST pre-defines a small hierarchy that encompasses some commonly used lock behaviours. The predefined lock hierarchy includes definitions for standard read/write locking behaviour, mutual exclusion and non-conflicting “no locks". The definitions for these lock classes follow.

---

**class** ReadLock  
**superclass** ProgrammableLock  
**instance variables** none

"A ReadLock is compatible with other ReadLocks and NoLocks."

**locking**

isCompatibleWith: anotherLock

↑(anotherLock isKindOf: NoLock)  
| (anotherLock isKindOf: ReadLock)

---

**class** WriteLock  
**superclass** ProgrammableLock  
**instance variables** none

"A WriteLock is only compatible with NoLocks."

**locking**

isCompatibleWith: anotherLock

↑anotherLock isKindOf: NoLock
class MutualExclusionLock
superclass ProgrammableLock
instance variables none

" A MutualExclusionLock is incompatible with all other ProgrammableLocks. "

locking

isCompatibleWith: anotherLock
↑false

class NoLock
superclass ProgrammableLock
instance variables none

" a NoLock can be forced to wait by a MutualExclusionLock. "

locking

isCompatibleWith: anotherLock

↑(anotherLock isKindOf: MutualExclusionLock) not

All classes express their compatibility with other sorts of programmable locks by over­
riding the default ProgrammableLock isCompatibleWith: method. MutualExclusion­
Lock isCompatibleWith: returns false to express the fact that MutualExclusionLock
locks are incompatible with every other sort of pre-defined lock. NoLock locks can be
used by methods that only read constant variables. Because a constant variable can­
not subsequently be written a NoLock lock can be less restrictive than a ReadLock lock.
NoLock locks must still interact correctly with MutualExclusionLock locks. NoLock is­
CompatibleWith: expresses this by returning false if the other lock is mutual exclusion
lock. ReadLock isCompatibleWith: and WriteLock isCompatibleWith: implement
the standard read/write locking compatibility matrix introduced in section 3.2.1.1.2. A
ReadLock lock is compatible with other read locks and NoLock locks. A WriteLock lock
is only compatible with other NoLock locks.

The resultant lock compatibility matrix for the pre-defined Hermes/ST locks is shown
in Figure 4.1.

4.2.2 Extending the Programmable Lock Hierarchy

The following code fragment is the overridden definition of the isCompatibleWith:
method for the AccountWriteLock that is to be used by the deposit method.

isCompatibleWith: anotherLock
"
  I am compatible with noLocks and most other AccountWriteLocks
"
CHAPTER 4. IMPLICIT AND EXPLICIT CONCURRENCY CONTROL

Figure 4.1: Standard Hermes/ST Locks - Compatibility Matrix

<table>
<thead>
<tr>
<th>Requested Locking Mode</th>
<th>M</th>
<th>W</th>
<th>R</th>
<th>N</th>
</tr>
</thead>
<tbody>
<tr>
<td>M</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>W</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>R</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>N</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
</tbody>
</table>

The code can be interpreted as follows. An AccountWriteLock is compatible with another lock if the two locks are compatible according to the pre-defined system table, or if the locks refer to separate accounts. super isCompatibleWith:anotherLock tests for compatibility with other pre-defined lock types. AccountWriteLock is derived from WriteLock. Therefore, its compatibility with the pre-defined hierarchy is the same as that of a write lock. The expression super isCompatibleWith:anotherLock tests this. The expression lockedAccountName ~= anotherLock lockedAccountName tests that the locks refer to separate accounts.

Methods of transactional objects can be associated with a programmable lock class. The invocation of such methods causes the instantiation of the programmable lock class as a Smalltalk object. For example, the lock instantiation method of the AccountWriteLock class is as follows.

```
| res |
res := (super isCompatibleWith: anotherLock)
or: [lockedAccountName ~= anotherLock lockedAccountName].

| res |
```

This code instantiates the AccountWriteLock class and sets the instance variable lockedAccountName to the argument of the method. This instance creation argument is the value that is later used by AccountWriteLock isCompatibleWith:.

Locks are volatile and are lost in the event of node crashes. A uniform concurrency controller associated with each transactional object invokes the instantiated programmable lock's schedulability and compatibility method to determine when the invocation can be scheduled. Like programmable lock objects, concurrency controllers are volatile and are lost in the event of node crashes.
4.2.3 The Programmable Lock Concurrency Controller

The concurrency controller associated with each transactional object owns a queue of pending requests and a set of granted requests. A request represents a method invocation by a message and contains the method name, its arguments, the thread identifier and the associated programmable lock. At any point in time, all programmable locks associated with requests in the set of granted requests are mutually compatible and their isSchedulable: methods have evaluated to true. Furthermore, at any point in time, all locks associated with requests in the queue of pending requests are incompatible with at least one granted request, or their isSchedulable method has evaluated to false.

In order for an invocation request to proceed one of two independent conditions must be satisfied.

1. Either the incoming request is part of a thread that can be dispatched according to the Hermes/ST advanced scheduling mechanism (refer chapter 3.1.5.1.7)

2. Or both isSchedulable evaluates to true and the incoming request’s lock is compatible with each lock in the set of granted requests. This compatibility is tested by calling the incoming request lock’s isCompatibleWith: method with a granted lock as an argument.

If both of these conditions are false with respect to any granted request then the incoming request is placed at the end of the queue of pending requests. The completion of a thread or transaction causes some locks to be released. Whenever a lock is released each pending request is processed in the same manner as an incoming request. Pending requests are processed from the head to the tail of the pending queue.

4.2.4 Applying the Programmable Lock Hierarchy: The Method-Lock Association

An object’s concurrency controller instantiates programmable locks that are associated with a transactional object’s method as part of the method invocation process. Parameters can be passed to the instance creation method of the programmable lock. These parameters are specified when the programmable lock is associated with a transactional object’s method. The following tuple represents the general structure of a method-lock association.

<method name> <lock name> <arg1> ... <argn>

The parameters <arg1> ... <argn> are by convention stored as variables of the lock. These variables can be accessed by the lock’s methods isSchedulable and isCompatibleWith: whenever these methods are invoked by the concurrency controller.

Parameters to the programmable locks can be the actual parameters of the transactional object’s method invocation, the names of guard methods of the transactional object or any other relevant expression.

As with other message parameters (refer chapter 3.1.6) method-lock associations can be specified during method declaration as demonstrated by

---

2 The queue of pending requests differs from the queue abstract data type in that elements can be de-queued from any position in the queue.
depositTo: accountName amount: amount

InvocationScheme
lock: [AccountWriteLock account: accountName]

Furthermore, a lock-method association can be overridden by a method invocation as demonstrated by

lockSpecificationAtMethodInvocation

An example of specifying a lock as a message parameter at method invocation.

| hp |

hp := NameServer hermesPointerOf: #branch1.

hp lock: [AccountWriteLock account: #demonstrationAccount]; depositTo: #demonstrationAccount amount: 300

4.2.4.1 Accessing Method Arguments

Consider the example of the programmable lock class AccountWriteLock which is associated with method HermesSTBranch depositTo: amount:. The method-lock association in depositTo: amount: specifies that the argument accountName is passed to AccountWriteLock instance creation method account:. As demonstrated in section 4.2.2 the argument (accountName) is first stored by AccountWriteLock account: in the instance variable lockedAccountName. This is later tested by AccountWriteLock isCompatibleWith: to determine whether anotherLock refers to the same account as the lock itself. This re-defined behaviour of isCompatibleWith: allows more than one AccountWriteLock to be granted in each branch of the distributed bank.

4.2.4.2 Accessing Object State with Guard Methods

A read-only method of a transactional object that inspects object state is called a guard method [Atk91]. Guard methods are used to access the state of a transactional object at the time a programmable lock is tested by the concurrency controller. This type of behaviour is useful when operations must be delayed until a condition is satisfied. A bounded buffer is a good example of this sort of behaviour. A bounded buffer is a fixed size, first-in first-out (FIFO) queue. Because of its fixed size, a bounded buffer needs to delay a put operation if the buffer is full. It also needs to delay a get operation if the buffer is empty. In order to achieve this, locks must be able be ascertain the current state of the buffer. Guard methods are the Hermes/ST mechanism that permit this.

Although guard methods are read-only, they must be synchronised with currently active method invocations to ensure that guard methods do not see an inconsistent object state. For example, the check whether a bounded buffer is empty or not should not be performed during the invocation of a put method.

Guard methods can acquire programmable locks in the same manner as other transactional object method invocations. This ensures that guard methods are synchronised with any active method invocations. Guard methods must not hold locks until transactions commit or abort. If a programmable lock has been associated with a guard method then it is released when the guard invocation is complete. The early release of guard locks is
necessary to ensure that guard methods do not stop the execution of other transactional object methods. For example, consider an implementation of a circular bounded buffer with a head and a tail pointer, that exports get and put methods. Further consider that the get method utilises a guard method isEmpty to determine if there are any elements in the bounded buffer. An isEmpty method has to read both the head and the tail pointer and hence needs to acquire a read lock on the bounded buffer structure. If the read lock acquired by the isEmpty guard method is not released before the completion of the get message's encompassing thread then a put message which requires write access to the head and tail pointers could not execute. This would result in deadlock. It is the programmers responsibility to ensure that the early release of the guard locks does not affect the serialisability of transactional messages that visit the programmably locked object. The PutLock and GetLock of section 4.2.6 do this.

Recall that a guard method is invoked from within the isSchedulable or the isCompatibleWith: method of a programmable lock. If a guard method fails to acquire one of its locks then isSchedulable or isCompatibleWith: returns false. This means that the transactional object method invocation that the guard was protecting cannot be scheduled at this time. The concurrency controller will re-try scheduling the method later, as described in Subsection 4.2.3.

Guard methods can be invoked from either the isSchedulable or isCompatibleWith: methods. However, since isCompatibleWith: is invoked potentially many times per lock test, it is more efficient to invoke guard methods from within isSchedulable, which is invoked only once per lock test.

4.2.5 State Restoration and Persistence Handling

Transactional objects perform state restoration and persistence handling. The granularity of these operations for programmably locked objects is the same as for minimally locked objects. The granularity of state restoration for minimally locked objects is discussed in section 4.4.2. The granularity of persistence handling for minimally locked objects is discussed in section 4.4.3

4.2.6 The Cheque Printing Example

The banking application uses a bounded buffer to implement a check printing facility. Each branch object contains a variable called printSpool which refers to an instance of the HermesSTProgBB3 class. Branch objects contain two methods related to cheque printing. These are the HermesSTBranch enqueue: and HermesSTBranch dequeue: methods. The code for these methods follows.

enqueue: anObject
\nPlaces the object <anObject> into the bounded buffer of <printSpooler>
\nself printSpooler put: anObject

dequeue
\nInvocationScheme

3The class name stands for Hermes/ST programmably locked bounded buffer class.
async

<table>
<thead>
<tr>
<th>anObject</th>
</tr>
</thead>
<tbody>
<tr>
<td>[true]</td>
</tr>
</tbody>
</table>
whileTrue:
  [anObject := self printSpooler get.
   ChequePrinter print: anObject]

enqueue: places an object (cheque) in the buffer. It uses the HermesSTProgBB put:
method to do so.

dequeue is an asynchronous non-terminating loop that retrieves objects from print-
Spool via HermesSTProgBB put: and sends them to the check printing device (Cheque-
Printer). Care must be taken when printing cheques, even if the dequeue operation is
performed transactionally. This is because physical actions such as typing on a cheque
are not undoable.

In order to facilitate reuse, the code that implements the semantics of the bounded
buffer has been separated from the code that synchronises the bounded buffer. Section
4.5 demonstrates this reuse. The semantic part of the bounded buffer is implemented in
the class HermesSTBoundedBuffer. The code for the relevant parts of the class Hermes-
STBoundedBuffer follows

put: anObject

  | res |
  | res := self at: self head.
  self head: self head \ self size + 1.
  \res

get

isFull

  ↑self head = (self tail \ self size + 1)

isEmpty

  ↑self head = self tail

The HermesSTBoundedBuffer class uses an array to implement the bounded buffer. Ele­
ments are inserted (via put:) at the tail of the buffer and are removed (via get) from the
head of the buffer. The current head and tail positions are recorded by head and tail
indexes which navigate the array in a circular manner. The at: and at:put: methods
mimic the behaviour of the Smalltalk at: and at:put: methods for indexed instance
variables. The at: and at:put: behaviour is defined in the Hermes/STArray class, from
which the HermesSTBoundedBuffer class is subclassed.

Applying synchronisation to the HermesSTBoundedBuffer class is achieved subclassing
from it and adding in synchronisation constraints. This is demonstrated in the following
methods of the HermesSTProgBB class.
put: anObject
"
InvocationScheme lock: [PutLock isFullMethod: #isFull]
"

↑super put: anObject

get
"
InvocationScheme lock: [GetLock isEmptyMethod: #isEmpty]
"

↑super get

isFull
"
InvocationScheme
  lock: ReadLock
"

↑super isFull

isEmpty
"
InvocationScheme
  lock: ReadLock
"

↑super isEmpty

All methods define their bounded buffer semantics by calling the appropriate method of the parent HermesSTBoundedBuffer class. This is expressed by the Smalltalk super construct. For example, HermesSTProgBB put: performs the HermesSTBoundedBuffer put: method by denoting super put:. Synchronisation is added via Hermes/ST lock declarations.

Methods put: and get both change object state, so their associated programmable lock classes PutLock and GetLock are defined as subclasses of WriteLock. isFull and isEmpty are used as guard methods, so they are passed as arguments to the programmable locks’ instance creation method. For example, the HermesSTProgBB method get is associated with the programmable lock class GetLock. The method passed to the programmable lock instance creation is isEmpty. The syntax for this association is as follows.
"

InvocationScheme lock: [GetLock isEmptyMethod: #isEmpty]
"

isFull and isEmpty both instantiate ReadLocks to synchronise themselves with other method invocations.

PutLock and GetLock inherit the isCompatibleWith: method from WriteLock and ReadLock respectively without modification. They only override their respective isSchedulable methods. For GetLock, the isSchedulable method is defined as follows.
CHAPTER 4. IMPLICIT AND EXPLICIT CONCURRENCY CONTROL

isSchedulable

\[ \text{empty} \]
\[
\text{empty} := (\text{self receiver}) \text{guard; hermesPerform: isEmptyMethod.}
\]
\[ \text{full} \]
\[
\text{full} := (\text{self receiver}) \text{guard; hermesPerform: isFullMethod.}
\]

whereas for PutLock the isSchedulable method is defined as follows.

isSchedulable

\[ \text{full} \]
\[
\text{full} := (\text{self receiver}) \text{guard; hermesPerform: isFullMethod.}
\]
\[ \text{get} \]
\[
\text{GetLock isSchedulable performs the guard method isEmptyMethod which has been}
\]
\[ \text{associated with HermesSTProgBB isEmpty as specified by the association above. If a}
\]
\[ \text{guard lock of isEmpty cannot be granted or the buffer is actually empty then false is}
\]
\[ \text{returned and the ReadLock of isEmpty is released (see section 4.2.4.2). When GetLock's}
\]
\[ \text{isSchedulable becomes true and the isCompatibleWith: method returns true against}
\]
\[ \text{all currently granted requests, then the get request can be scheduled. A similar argument}
\]
\[ \text{holds for the PutLock isSchedulable method.}
\]

4.3 Evaluation of Hermes/ST Programmable Locking

As was introduced in chapter 2, Hermes/ST mechanisms are evaluated against four criteria:

1. **Performance**: Programmable locking allows the application developer to develop classes that trade off the level of parallelism and deadlock versus the number of locks that they employ. Programmable locks allow the application developer to affect the level of parallelism by avoiding "unnecessary contention". For example, an AccountWriteLock allows a high level of parallelism when there are many deposits to different accounts, while only instantiating as many AccountWriteLocks as there are outstanding deposits.

   Individual programmable locks are expensive to test and set. For example, the AccountWriteLock isCompatibleWith: method has to compare one account name with another account name. In general, isCompatibleWith: methods can be of arbitrary complexity. The magnitude of the lock management costs from schedulability and compatibility testing for the programmable lock approach and the resultant impact on performance will vary from application to application.

2. **Incremental Development**: First, simple system-defined programmable locks like mutual exclusion locks or read/write locks can be employed. Performance analysis of this system may detect bottlenecks. These bottlenecks may then be alleviated by the introduction of more sophisticated application-specific programmable locks such as the AccountWriteLock.

3. **Reuse**: The Hermes/ST programmable lock approach supports 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 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 to different classes as appropriate.

• Since programmable locks are defined in an inheritance hierarchy, concurrency control behaviour can be reused through “programming by difference”. An example is the implementation the AccountWriteLock class, which utilised the locking behaviour of its superclass WriteLock and weakened the isCompatibleWith: using a logical “or” operator.

4. Ease of Specification:

A drawback of the programmable lock approach is the amount of thought required in order to develop programmably locked methods. The application developer has to use some existing suitable programmable lock, or if one does not exist, develop a new programmable lock. The development of the AccountWriteLock demonstrated how this can be done for the depositTo:account: method.

However, not all methods can specify their resource sets so precisely. Consider, for example, the HermesSTBranch openAccount: method that creates a new account at an existing branch. HermesSTBranch openAccount: achieves this by creating a new node in the branch's binary tree and making that node refer to this new account. Therefore, the position of the new account in account tree is dependent on the existing structure of the tree. This makes the development of an “AccountOpenLock” that allows parallel account openings harder than it was for the AccountWriteLock. The problem is that the separation of the nodes of the binary tree that HermesSTBranch openAccount: reads and writes is no longer easily specifiable. The application developer can easily “give up” and use a WriteLock for the openAccount: method. The use of a WriteLock by HermesSTBranch openAccount: would mean that only one account could be opened at a branch at any given time. Furthermore, because WriteLocks are incompatible with most other Hermes/ST programmable locks, there would be no other account activity possible during an account open. This type of over-specification problem is not restricted to binary trees. Any object structure that is governed by one lock, and is partly read or partly written, will be susceptible to such over-specifications.

4.4 Minimal Locking

Hermes/ST supports uniform reference semantics over transactional, volatile and constant objects. Therefore, the read and write accesses of variables precisely determines the resource set of a thread. Precise knowledge of a thread’s resource set allows the Hermes/ST system to avoid unnecessary lock contention. Furthermore, no explicit concurrency control specifications like the AccountWriteLock of section 4.2.4 are needed to avoid this unnecessary lock contention.
Recall that all variables in Hermes/ST objects are accessed via Hermes/ST access methods (refer chapter 3.1.1.1.1). When *minimal locking* is being used with transactional objects, the Hermes/ST system associates a read lock with a read access method and a write lock with a write access method. This association is performed automatically by the Hermes/ST system. The resultant locks are managed by minimal locking concurrency controllers and their use avoids unnecessary lock contention.

### 4.4.1 The Minimal Locking Concurrency Controller

To implement minimal locking, Hermes/ST associates a concurrency controller with each variable of each minimally locked transactional object. This concurrency controller owns a queue of *pending variable access requests* and a set of *granted variable access requests*. Variable access requests contain, amongst other things, minimal locks. At any point in time, all minimal locks associated with variable access requests in the set of granted requests are mutually compatible. Furthermore, at any point in time, all locks associated with requests in the queue of pending requests are incompatible with at least one granted request.

In order for an incoming invocation request to proceed, one of two independent conditions must be true with respect to all requests in the granted set.

1. Either the incoming variable access request is part of a thread that can be dispatched according to the Hermes/ST advanced scheduling mechanism (see chapter 3.1.5.1.7).
2. Or the incoming variable access request is compatible according to the standard read write locking rules (refer section 3.2.1.1.2). Namely, a variable read request is compatible with other variable read requests but incompatible with variable write requests. A variable write request is incompatible with other variable read and write requests.

If both of these conditions evaluate to false for any granted request then the incoming request is placed at the end of the queue of pending variable access requests. The completion of a thread or transaction causes some minimal locks to be released. Whenever a lock is released, each pending variable access request is processed in the same manner as an incoming request. The pending variable requests are processed from the head to the tail of the pending queue.

### 4.4.1.1 Concurrency Control

Minimal locking has not changed the nature of the Hermes/ST scheduling mechanism. Minimally locked objects still use the Humm extensions to two phase locking to serialise transactional threads and synchronise non-transactional threads (refer section 3.1.5). From a concurrency control perspective, all that minimal locking has changed is the granularity of a lockable unit. Namely, individual variables rather than objects are concurrency controlled.

Transactional objects also perform state restoration and persistence handling (refer chapter 3.1.2). Choosing the granularity of concurrency control to be at the level of a variable has repercussions on the nature and granularity of state restoration and persistence handling. This is outlined in the following sections.

### 4.4.2 State Restoration

State restoration is used to restore the memory version of a Hermes/ST object in the case that a visiting transaction fails. Hermes/ST state restoration is achieved through version
Granularity stacks (refer section 2.4.2). Because Hermes/ST objects are always cached in memory during their use, the version stack for each recoverable part of a transactional object is a memory data structure. Before any changes are made to the memory version of a transactional object, a copy of the relevant part of the object is placed on a version stack. This information is used to restore the object in the case that a (nested) transaction aborts.

The granularity of state restoration must be at least as fine as that of concurrency control. To see why this is so, consider the case that the granularity of state restoration is coarser than the granularity of concurrency control. In this example, state restoration is performed on objects and concurrency control is performed on variables. Such a scenario is depicted in Figure 4.2. Figure 4.2 (a) shows the state of an account object before any transactions have visited it. The account object is to be visited by two transactions, T1 and T2. T1 changes the address field to refer to 'University of Sydney' and T2 changes the balance variable to refer to 1000. Thus, if both T1 and T2 commit the account object should represent a balance of $1000 and an address at the University of Sydney. If both T1 and T2 abort then the account object should be in its original state as depicted in Figure 4.2 (a). If T1 commits and T2 aborts the account object should have an address representing the University of Sydney and a balance representing zero dollars. Similarly, if T1 aborts and T2 commits the account object should have an address representing the University of Wollongong and a balance representing one
thousand dollars.

The following interleaving of $T_1$ and $T_2$ can cause an erroneous result in the abort case. $T_1$ write locks the address variable and makes a copy of the account object. At this same point in time transaction $T_2$ commences execution on the account object. It write locks the balance variable and also makes a copy of the account object. Both state restoration copies now contain the original version of the account object. See figure 4.2 (b).

Transaction $T_1$ runs to completion and commits. Transaction $T_2$ now decides to abort for some reason (deadlock, programmer specified abort ...). Figure 4.2 (b) shows the histories of the transactions at this point in time. The aborting transaction $T_2$ will attempt to return the object to the state that it recorded as it entered the object. In doing so the account object will be returned to having a balance representing zero and an address representing the University of Wollongong. The state restoration of $T_2$ has caused the in memory version of the committed effects of $T_1$ to be lost and therefore is incorrect. Figure 4.2 (c) depicts this erroneous situation. This example demonstrates that state restoration granularity cannot be greater than the granularity of concurrency control.

As the granularity of state restoration must be at least as fine as the granularity of concurrency control, minimal locking needs a version stack for each variable. A variable's version stack can implement state restoration by only recording references to any old objects that it referred to. Reference recording is sufficient for state restoration because of Hermes/ST's uniform reference semantics with no containment. Therefore, when a variable is made to refer to a new object the old object is still accessible in the Hermes/ST system. Systems that employ containment semantics must deep copy [Mey88] the state restoration information for any variables that are changed by a transaction. For example, if an object has a variable that stores the bits that represent a date then a change to those bits entails making a copy of the date for state restoration purposes. The term “deep copy” refers to the need to copy the entire contained entity.

Hermes/ST consists of transactional, volatile and constant objects only. When changing a variable that refers to a constant object the version stack for that variable need only record a reference to the old constant object. A copy of the constant object is not needed because constant objects are immutable. When changing a variable that refers to a volatile or transactional Hermes/ST object it is sufficient to record only a Hermes/ST pointer to the old Hermes/ST object on the version stack. In the event of a transaction failure the old Hermes/ST pointer is restored into the variable. The referenced Hermes/ST object is responsible for restoring its own state.

4.4.2.1 Analysis of Hermes/ST Fine Grained State Restoration

Like most mechanisms, Hermes/ST fine grained state restoration has positive and negative aspects. The positive features are the following.

1. The fine granularity mechanism avoids performing more state restoration than is necessary. This is analogous to minimal locking only locking the parts of an object that are actually accessed. Only the variables that have been changed have state restoration performed on them.

2. The “copy operation” only involves pushing a reference onto the version stack and is more efficient than a deep copy operation. Hermes/ST uniform reference semantics without containment allows such reference pushing.

As both transactions write to different variables and minimal locking is used, they can both proceed in parallel.
The negative aspects of fine grained state restoration are as follows.

1. Each variable write access has to perform state restoration. Even though an object level state restoration mechanism has to deep copy the whole object, it only has to copy it once. The minimal locking state restoration mechanism is usually called many times to achieve the same result. Therefore the overhead of invoking the state restoration mechanism is repeated many times.

2. Uniform reference semantics does add an extra level of indirection to each variable access. Because each variable contains a reference to its contents, reading a variable entails a de-reference operation. Depending on the nature of this reference the de-reference operation can be more or less expensive. For example, Hermes/ST constant objects are implemented in the underlying Smalltalk system and are inexpensive to de-reference. However, transactional and volatile objects are more expensive to de-reference. Naturally, clever programming techniques can blur this distinction.

With respect to the granularity of the copy operation, no approach is better than others in all cases. If the transitive closure of objects is typically large then object level granularity is probably unwise. If the transitive closure of objects is typically small then the overhead of invoking state restoration once per variable access may be prohibitive. In the final analysis it is worthwhile remembering that locking individual variables dictates that state restoration be at least as fine.

4.4.3 Persistence Handling

Persistence handling is used to make permanent the effects of a committed transaction on Hermes/ST objects. As introduced in section 3.1.2 Hermes/ST permanence handling is achieved via a two phase commit protocol with individual objects recording their changes in a redo log. During a transaction, individual changes to an object can be asynchronously entered into the log. If the transaction decides to commit, phase one of the protocol ensures that all of the asynchronous writes generated during the transaction have completed. Phase two of the two phase commit is responsible for placing a commit record into the log.

As was the case for state restoration, the granularity of persistence handling must be at least as fine as the granularity of concurrency control. To see why this is so, consider the case that the granularity of persistence handling is coarser than the granularity of concurrency control. In this example, persistence handling is performed on objects and concurrency control is performed on variables. Such a scenario is depicted in Figure 4.3.

Figure 4.3 (a) shows the initial committed state of an account object. The account object is visited by two transactions, T1 and T2. T1 changes the address field to refer to 'University of Sydney' and T2 changes the balance variable to refer to 1000. Thus, as before, if both T1 and T2 commit the account object should represent a balance of $1000 and represent an address for the University of Sydney. If both T1 and T2 abort then the account object should be in its original state as depicted in Figure 4.3 (a). If T1 commits and T2 aborts the account object should have an address representing the University of Sydney and a balance representing zero dollars. Similarly, if T1 aborts and T2 commits then the account object should have an address representing the University of Wollongong and a balance representing $1000.

\(^5\)The set of objects that form the deep copy.
Figure 4.3: Incorrect Relationship between Persistence and Concurrency Control Granularity
The following interleaving of \( T_1 \) and \( T_2 \) can cause an erroneous result. \( T_1 \) and \( T_2 \) both execute the body of their transaction in parallel. This is possible because the granularity of concurrency control makes the resource set of both transactions disjoint. Both transactions initiate their commit actions. \( T_1 \) completes its two phase commit without an error, causing the object state to be recorded as 'Fazzolare', 'University of Sydney' and $1000. This is shown in Figure 4.3 (b).

Conversely, \( T_2 \) fails to complete the prepare phase of its two phase commit thus causing the transaction \( T_2 \) to abort. The fact that \( T_2 \) has aborted is not reflected by the account object. \( T_1 \) has caused the balance variable to be logged as $1000. As depicted in figure 4.3 (c), the final logged version of the account object is incorrect. This example demonstrates that persistence handling granularity cannot be greater than the granularity of concurrency control.

Hermes/ST consists of transactional, volatile and constant objects only. When logging a variable that refers to a constant object a deep copy of the constant object is logged. When logging a variable that refers to a volatile object only the Hermes/ST nil reference is logged. A copy of the old volatile object is not necessary because volatile objects are not reconstructed after a node crash. When changing a variable that refers to a transactional object it is sufficient to log the Hermes/ST pointer to the old transactional object. A copy of the old transactional object is not logged because it will already be present in the object store.

### 4.4.3.1 Analysis of Hermes/ST Fine Grained Persistence Handling

Fine grained persistence handling avoids performing more logging than is necessary. Fine grained persistence handling is common [GR93, BHG87]. Because persistence handling is typically performed by a disk that is much slower than primary storage this advantage can be emphasised if objects are large. For example, persistence handling of variables that refer to Hermes/ST objects only involves writing a Hermes/ST pointer into the log. This is more efficient than writing a deep copy of the object. As was the case with state restoration, it is worthwhile remembering that locking individual variables dictates that persistence handling be at least as fine grained.

### 4.4.4 An Example of Minimal Locking

To demonstrate the effects of precisely specifying the resource sets of an operation a minimally locked implementation of the HermesSTBranch openAccount: method is now presented.

```plaintext
openAccount: newAccount

InvocationScheme
lock: WriteLock

\[ \uparrow \text{self accountsTree add: newAccount} \]
```

The `openAccount:` method merely calls the `add:` method of the binary accounts tree for the branch. When minimal locking is being used any programmable lock specification

---

6Because, for example, a communication link to one of the objects involved in the two phase commit has temporarily failed.

7Within the limits of the disk atomicity model. I.e. only complete blocks can be transferred atomically. Group commit algorithms [Wei93b] can be useful when writing less than a block.
such as for HermesSTBranch openAccount: is ignored.

A minimally locked implementation of the branches' transactional binary tree add: operation is now presented. Accounts are indexed by their name variable. The binary tree used in a branch is specified via two Hermes/ST classes, HermesSTBinaryTree and HermesSTBinaryTreeNode. The definition of the HermesSTBinaryTree class and its add: method are as follows.

```plaintext
class HermesSTBinaryTree
superclass HermesCollection
instance variables root

add: data
" 
   data <HermesPointer> to <HermesBTData>
" 

↑self root isNil
ifTrue:
   [self root: (hermesSelf containNew; hermesClass: HermesSTBinaryTreeNode).
    self root contain: data]
ifFalse: [self root add: data]
```

HermesSTBinaryTree contains a single variable called root. This variable is used to refer to the binary tree. An empty binary tree is modelled by the root variable referring to the constant object nil. HermesSTBinaryTree add: data is passed a reference to the data object it is to store in the transactional binary tree. If root refers to the constant nil, HermesSTBinaryTree add: creates an instance of HermesSTBinaryTreeNode that refers to the argument data. This binary tree node creation clones the locking style of the parent node. Therefore, the created HermesSTBinaryTreeNode is by default also minimally locked. If there is already an instantiated HermesSTBinaryTreeNode then add: calls the add: of that instantiated node.

HermesSTBinaryTreeNodeNodes are defined as follows.

```plaintext
class HermesSTBinaryTreeNode
superclass HermesCollection
instance variables left right contents
```

Each HermesSTBinaryTreeNode contains three variables that are used to model a binary tree. The left variable is a Hermes/ST pointer to the left subtree of this node. The right variable is a Hermes/ST pointer to the right subtree of this node. An empty subtree is modelled by a parent binary tree nodes' left or right variable referring to the constant object nil. The contents variable is a Hermes/ST pointer to the data for this node. In the banking application, the data refers to instances of the HermesSTAccount class. The data could as well be integers or strings or whatever the tree needs to store.

---

8See section 3.1.4.1.
CHAPTER 4. IMPLICIT AND EXPLICIT CONCURRENCY CONTROL

The only restriction on data objects is that they implement a `compareData:` method. This method must compare its argument with itself and return one of the constant objects `#equal` or `#less` or `#more` according to the outcome of the compare. The `compareData:` method for `HermesSTAccount` is as follows.

```
compareData: account

< account> Hermes/ST pointer to another HermesSTAccount.

returns: #less, #more or #equal

account key = self key ifTrue: [↑#equal].
account key < self key
  ifTrue: [↑#less]
  ifFalse: [↑#more]
```

`HermesSTBinaryTreeNode add:` is implemented as follows.

```
add: data

| r |
r := self contents compareData: data.
r = #equal ifTrue: [↑#duplicate].
r = #less ifTrue: [self left isNil
  ifTrue:
    [self left: (hermesSelf containNew; hermesClass: HermesSTBinaryTreeNode).
     ↑self left contain: data]
  ifFalse: [self left add: data]].
r = #more ifTrue: [self right isNil
  ifTrue:
    [self right: (hermesSelf containNew; hermesClass: HermesSTBinaryTreeNode).
     ↑self right contain: data]
  ifFalse: [self right add: data]]
```

The `add:` method is a recursive implementation of a binary tree add. If the data to be added is equal to the data of the current node then `add:` returns the constant object `#duplicate`. If the data belongs in the left subtree (is `#less` than the current node) and the subtree is empty then the data is added as the new left subtree. If the data belongs in the left subtree and the left subtree exists then the `add:` method of the left subtree is recursively called. Adding to the right subtree is analogous. If the data belongs in the right subtree (is `#more` than the current node) and the subtree is empty then the data is added as the new right subtree. If the data belongs in the right subtree and the right subtree exists then the `add:` method of the right subtree is recursively called.

Adding data as either a new left subtree or a new right subtree involves creating an instance of `HermesSTBinaryTreeNode` and making it refer to the data. The `HermesSTBinaryTreeNode contain:` method does this. As was the case with `HermesSTBinaryTreeNode add:`, the default locking style of the `HermesSTBinaryTreeNode` object is the same as that of the parent object. In this case, the object will be minimally locked.
4.4.4.1 How Minimal Locking Increases Concurrency

Figure 4.4 illustrates the structure of a branch that contains three accounts. Consider opening an account for 'Bart'. According to the code above, the addition of Bart's account will read lock the accountTree variable, the root variable, the contents variable of "node 1", the name variable of Harry's account, the left variable of "node 1", the contents variable of "node 2", the name variable of Fred's account, and the left variable of "node 2". Furthermore, during the contain: operation, it will write lock the left variable of "node 2".

Consider opening an account for 'George'. According to the code above, the addition of George's account will read lock the accountTree variable, the root variable, the contents variable of "node 1", the name variable of Harry's account, the left variable of "node 1", the contents variable of "node 2", the name variable of Fred's account, and the right variable of "node 2". Furthermore, during the contain: operation, it will write lock the right variable of "node 2".

As can be seen, the set of variables written is disjoint. The sets of variables read intersect but read operations are compatible with each other (i.e they commute). Furthermore, no variable written by the opening of the 'Bart' account is read by the opening of the 'George' account and vice versa. Therefore, both operations are compatible with respect to read/write locking and can proceed in parallel. With the programmably locked specification (of section 4.2.2), it was only possible to add one account at a time. With minimal locking it is possible to add multiple accounts simultaneously. To the structure depicted in Figure 4.4 it is possible to add concurrently accounts for 'Bart', 'George', 'Justin', and 'Peter'.

Without having shown the code that searches an account tree and that is used by deposit methods, it should be clear that minimal locking achieves the same level of concurrency as does the programmably locked version. Minimally locked deposit operations only write lock the balance variable of the account that they alter. Furthermore, account deposit methods can now proceed in parallel with account opens, provided the deposit method is not to an account in that is in the process of being opened.

There are some sequences of openAccount: operations that cannot proceed in parallel. Starting with the accountTree structure of Figure 4.4 it is not possible to open an account with the name 'Bart' in parallel with an account that has the name 'Alex'. Adding the account named 'Bart' to the accountTree will write lock the left variable of "node 2". This will cause the add: of the account named 'Alex' to delay its read of the left variable of "node 2" until after the completion of the add: of the account named 'Bart'.

If more parallelism is required for account opening, minimal locking dictates that the account structuring mechanism needs to be changed. For example, a minimally locked and sparse hash table will tend to give better concurrency for account opening. This, in a sense is the main benefit of minimal locking. A structure is allowed to exhibit as much concurrency as individual reads and writes of its variables allow. This property is called maximal concurrency. Maximal concurrency can be extracted from any structure. This is exemplified by minimal locking being applied to all of the different classes HermesST-Branch, HermesSTBinaryTree, HermesSTBinaryTreeNode and HermesSTAccount.

4.4.5 Evaluation of Minimal Locking

The evaluation of Hermes/ST minimal locking is made against the same four criteria as the programmable lock approach

1. Performance: Minimal locking allows the application developer to develop classes that have a high degree of unnecessary contention avoidance. For example, a mini-
Figure 4.4: An example instantiation of a branch and its binary account tree.
mally locked binary tree implementation allows a high level of parallelism for both
count opens and account deposits. This avoidance of unnecessary contention
comes at the cost of high lock management costs. The high lock management costs
are introduced by the use of one concurrency controller per variable.

2. **Incremental Development:** Hermes/ST minimal locking allows an application
to be developed and tested as a (non-concurrent) volatile application. During this
phase, the application developer can test and validate the model in a sequential
environment. In the example distributed bank application, the application developer
can validate that the code for the binary tree additions is working as designed.
This can be achieved without the added complications of parallelism and failures.
When the sequential version of the binary tree works to the application developer's
saturation, it is only a small step to re-instantiate the code as an application
containing transactional objects. For example, a volatile **HermesSTBinaryTree** can
be instantiated in the following manner.

```plaintext
installVolatile
"Make a volatile branch object."

HermesSystem
  containNew;
  volatile;
  location: #groucho;
  args: (Array with: nil);
  hermesClass: HermesSTBinaryTree

A transactional **HermesSTBinaryTree** however can be instantiated as follows.

installTransactiona
"Make a transactional branch object."

HermesSystem
  containNew;
  transactional;
  location: #groucho;
  args: (Array with: nil);
  minimallyLocked;
  hermesClass: HermesSTBinaryTree

This simple evolution is made possible because the type of concurrency control to be
used by an object is an instance parameter and because by default, objects created
via **containsNew**; clone\(^9\) the same locking style as their parent.

3. **Reuse:** The same code can be used for a programmably locked and minimally
locked method. This is exemplified by the **openAccount**; method of section 4.4.4.
The consistent use of Hermes/ST access methods masks out the differences between
variable accesses in either style. Futhermore, the separation of method definitions

\(^9\)See section 3.1.4.1.
from lock specifications by programmable locking allows minimally locked objects to ignore any programmable lock specifications.

4. Ease of Specification:

One of minimal locking's most attractive features is its ability to extract high concurrency without the use of the explicit lock specifications that programmable locking requires. The programmer generally does not have to think about concurrency control and certainly does not need to specify it. The binary tree example above illustrates this point.

The implicit nature of minimal locking may also be problematic. Minimal locking cannot be used to specify all types of concurrency control specifications. Recall the programmably locked bounded buffer of the section 4.2.6. Minimal locking has no way of expressing the requirement that a get operation be delayed until after a put: operation when the bounded buffer is empty. Furthermore, minimal locking has no systematic way of avoiding any deadlocks that fined grained concurrency control may introduce.

4.5 Combining Minimal and Programmable Locking

Minimal and programmable locking each exhibit strengths and weaknesses. Programmable locking allows the specification of classes that can avoid unnecessary contention while limiting the number of locks used. Minimal locking allows the specification of classes that avoid unnecessary contention but does so by the introduction of many concurrency controllers. Minimally locked concurrency control is easy to specify but cannot be used for abstract data types such as the bounded buffer. Programmable locking involves extra specification. Both forms of locking allow reuse and incremental development.

Adapting the number of concurrency controllers used by minimal locking is examined in depth in the next chapter, chapter 5. In order to overcome the specificational deficiencies of minimal locking while maintaining as much ease of specification as possible, it is convenient in some circumstances to combine both forms of concurrency control. Hermes/ST allows the programmer to specify this through the boolean implicitCC: and explicitCC: instance creation parameters. If just implicitCC: is true then minimal locking is used. If just explicitCC: is true then programmable locks are used. The minimallyLocked and programmablyLocked instance creation parameters are merely shorthand instance creation parameters for implicitCC: true explicitCC: false and implicitCC: false explicitCC: true respectively. To stress the specificational differences between minimal and programmable locking, they are sometimes referred to as implicit and explicit locking.

If both implicitCC: and explicitCC: are false then the object is considered constant and read-only. The shorthand instance creation parameter for such objects is unLocked. Such objects can be used to apportion messages to other objects. For example, a Hermes/ST object may do nothing but call the methods of objects referred to by its variables.

If both implicitCC: and explicitCC: are true then the object performs both minimal and programmable locking. The shorthand instance creation parameter for implicitCC: true explicitCC: true is dualLocked. This type of locking is referred to as dual locking. Such a combination of locking types can reduce the amount of specification that is required for implementations of abstract data types.

To illustrate, reconsider the bounded buffer example of section 4.2.6. In that implementation it was necessary to specify PutLocks and GetLocks for the put: and get
methods respectively. A PutLock was responsible for ensuring that the buffer was not full and that only one put: operation happened at a time. The PutLock determined that the bounded buffer was not full by executing its guard method isFull. The execution of the guard method entailed ensuring that the guard method did not interfere with other guard methods or other get and put: operations. Both guard methods used a ReadLock to avoid interference with other active operations that may have been changing the state of the bounded buffer. These ReadLocks are not needed when dual locking is employed. The minimal locks of the bounded buffer will ensure that guard methods are synchronised with other bounded buffer operations.

In the HermesSTDualBB\textsuperscript{10} class both get and put: use a SyncLock, the definition of which is as follows.

```
class SyncLock
 superclass NoLock
instance variables guardMethod

isSchedulable
""

<table>
<thead>
<tr>
<th>full</th>
</tr>
</thead>
</table>
full := (self receiver) guard; hermesPerform: guardMethod.
|full not |
```

The SyncLock class is descended from the NoLock class. This is necessary because the synchronisation of the guard method will be performed by the minimal locks of the head and tail indexes. The SyncLock isSchedulable: method is defined to perform the guard method that is referred to by the variable guardMethod.

The definition of the HermesSTDualBB is also derived from the HermesSTBoundedBuffer class. The modified put: method is

```
put: anObject
""
InvocationScheme lock: [SyncLock guardMethod: #isFull]
""

|super put: anObject |
```

and the modified get method is

```
get
""
InvocationScheme lock: [SyncLock guardMethod: #isEmpty]
""

|super get |
```

Both methods have lock definitions that instantiate a SyncLock with an appropriate guard method. Put: uses isFull and get uses isEmpty. The definitions of isFull and isEmpty are inherited from the HermesSTBoundedBuffer class. Neither guard method now needs its own lock specification. Minimal locking automatically ensures that guard methods are synchronised with get and put: method invocations.

\textsuperscript{10}meaning the Hermes/ST dual locked bounded buffer.
4.5.1 Evaluation of Dual Locking

The arguments that applied to programmable and minimal locking regarding performance, reuse, incremental development and ease of specification apply to dual locking. Dual locking demonstrated that the implicitCC: and explicitCC: instance creation parameters can be applied in an orthogonal manner.
Chapter 5

Multi-Granular Concurrency Control

5.1 Introduction

This chapter extends minimal, programmable and dual locking to allow the application developer to have more control over the number of concurrency controllers employed by an application. In so doing the existing ease of specification, incremental development and reuse properties are maintained or enhanced. Section 5.2 introduces Hermes/ST's static multi-granular locking. Static multi-granular locking allows applications to be developed that define the most effective number of concurrency controllers prior to the application's instantiation.

The dynamic multi-granular locking of section 5.3 allows an application to change the number of concurrency controllers used by the application during the life of the application. These changes can be initiated directly by the application developer or indirectly via user defined triggers. Triggers can be specified to react to the level of parallel activity or other external factors. By changing the number of concurrency controllers that it employs, an application can react to changing access patterns to improve performance.

5.2 Static Multi-Granular Concurrency Control

5.2.1 NoCC Objects

One way to allow applications to regulate the number of concurrency controllers that they employ is to introduce objects that do not have concurrency controllers. The judicious use of such objects will allow the definition of Hermes/ST applications that acquire fewer locks than their minimally, programmably or dual locked counterparts. Naturally the introduction of such objects has to be done in such a manner so that the serialisable, atomic and permanent properties of transactions are maintained. Hermes/ST objects that have no concurrency controllers are henceforth referred to as noCC objects. NoCC objects are a special type of transactional object. NoCC objects still perform fine grained state restoration and persistence handling. Both the mechanisms are performed in the same manner as they were for transactional objects (Refer sections 4.4.2 and 4.4.3).

NoCC objects allow Hermes/ST applications to economise in both space and time. NoCC objects save memory space because they do not need to instantiate locks as data. NoCC objects shorten application execution time by avoiding lock management costs. NoCC objects also shorten application execution costs by reducing the communication of scheduling information to noCC objects. In order to supply transactional semantics the
Hermes/ST scheduling mechanism needs to be made aware of several thread events. Some of these events are only significant for lock acquisition and release purposes and need not be communicated to noCC objects. For example, a noCC object needs to be informed of top-level transaction completion events and nested transaction abort events in order to perform state restoration and persistence handling. However, nested transaction commit events do not need to be communicated to noCC objects. This is because Hermes/ST does not perform early writing [EME91] at the commit of a nested subtransaction. State restoration and persistence handling are deferred until top-level transaction commit. In contrast, a transactional object must be informed of nested transaction commit events so that any necessary lock scheduling by Hermes/ST can be performed.

5.2.2 Scheduling and NoCC Objects

The introduction of noCC objects introduces an extra burden on static multi-granular concurrency controlled applications in Hermes/ST. Hermes/ST ensures serialisability of transactional threads via the Hermes/ST scheduling mechanism introduced in section 3.1.5.1.7. This mechanism uses extensions to two phase locking to ensure that transactional messages to transactional Hermes/ST objects are performed in a serialised fashion. Non-transactional threads are synchronised, i.e. individual non-transactional messages are locked but locks are released at the completion of a message (see section 3.1.5.1.7).

A noCC object cannot implement the Hermes/ST scheduling mechanism for itself. This is because a noCC object does not have any concurrency controller to schedule interleaving method invocations on itself. Therefore, in order to guarantee that threads are correctly scheduled in the presence of noCC objects, the Hermes/ST system must arrange matters so that it is not possible for messages to interleave when visiting noCC objects. This objective is referred as the noCC schedulability objective.

One way to achieve the noCC schedulability objective is to only allow "one thread" to be active in a noCC object at a time. The Hermes/ST meaning of "one thread" is the set of messages in a message tree that are encompassed by an async; message creation parameter (refer section 3.1.5.1). Expressed another way, a thread is comprised of a set of synchronous messages. Because synchronous messages cause the suspension of the parent message while the synchronous child is being executed, it is not possible for a thread to interleave messages with itself. Because it is not possible for one thread to interleave with itself, and if only one thread is active at a noCC object at any one time, then the noCC schedulability objective will be achieved.

One method of ensuring that only one thread is active in a Hermes/ST object at any one time is by insisting that the concurrency control for threads that visit noCC objects be performed at some other surrogate Hermes/ST object. This surrogate Hermes/ST object can then schedule the threads in a correct manner on behalf of the noCC object. Such an object is called a guard object and is introduced in the next section. Guard objects perform scheduling for a group of noCC objects. Arranging matters within the Hermes/ST system so that all messages to groups of noCC objects have been scheduled by a guard object is a non-trivial exercise. This is because in the Hermes/ST system, as presented so far, it is possible for any Hermes/ST object to send a message to any other Hermes/ST object. The following sections introduce a variety of mechanisms, that in combination, manage message invocations so that the noCC schedulability objective is enforced.

1These messages can be transactional or non-transactional.
5.2.3 Guard Objects

As a first step in performing surrogate concurrency control for noCC objects, Hermes/ST introduces another type of object called a guard object. Because noCC objects can only have one thread active in them at a time, guard objects implement a mutual exclusion\(^2\) thread scheduling policy. Guard objects are another special type of transactional object. They perform state restoration and persistence handling in the same manner as a transactional object. Guard objects can be created using any combination of the implicitCC: and explicitCC: instance creation parameters (refer section 4.5).

5.2.4 Implicit Guard Objects

If the application developer creates guard objects with the instance creation parameters implicitCC: true; explicitCC: false, then a system defined guarded concurrency controller is associated with the guard object. As was the case in section 4.2.3 this concurrency controller owns a queue of pending requests and a set of granted requests. A request represents a method invocation and contains amongst other things, the method name, its arguments, and the message path. At any point in time, all requests in the set of granted requests are compatible according to the Hermes/ST scheduling mechanism of section 3.1.5.1.7. Furthermore, at any point in time, all requests in the queue of pending requests are not compatible. In order for an incoming request to proceed it must be Hermes/ST scheduling mechanism compatible with all requests in the set of granted requests.

An implicit guard object implements Hermes/ST mutual exclusion for any top-level messages that are sent to it. Hermes/ST mutual exclusion ensures that only messages that are descended from one top-level message are permitted to enter a guarded hierarchy at any one time. The Hermes/ST scheduling mechanism ensures that synchronous, asynchronous, transactional and non-transactional messages of each top-level message tree are scheduled in the correct fashion.

5.2.5 Explicit Guard Objects

If the application developer creates a guard object with the instance creation parameter explicitCC: true then Hermes/ST associates a programmable lock concurrency controller with the guard object (refer section 4.2.3). In this case it is the developers responsibility to ensure that the adequate locks are obtained over the group of noCC objects being protected by the guard object. For example, this type of behaviour can be useful when it is known that, say, read-only threads will be visiting a group of noCC objects. In such a case it is not necessary to impose a mutual exclusion execution order on the read-only threads. If say, a programmably locked guard object was specified, then a NoLock for each method invocation would achieve the desired result.

5.2.6 Hermes/ST Object Kind

With the introduction of guard and NoCC objects there are now four kinds\(^3\) of Hermes/ST objects. These are transactional, volatile, guard and noCC objects. The following function returns the kind of a Hermes/ST object:

\[
kind(O_{N1}) \mapsto x \text{ where } x \text{ can be one of transactional, volatile, guard or noCC}
\]

\(^2\)Refer to section 4.2.1 for the meaning of mutual exclusion.

\(^3\)See section 3.1.2.
CHAPTER 5. MULTI-GRANULAR CONCURRENCY CONTROL

Transactional and guard objects can be further categorised by combinations of implicitCC: and explicitCC: instance creation parameters. The kind of these objects will be qualified by the terms implicit and explicit when necessary.

5.2.7 Nested Encapsulation

Another step in the noCC schedulability objective is achieved via the introduction of nested encapsulation. Hermes/ST extends the traditional object–variable encapsulation structure so that Hermes/ST objects can encapsulate other Hermes/ST objects. A Hermes/ST object is either encapsulated by the Hermes/ST system or by another Hermes/ST object. Hermes/ST objects encapsulated by the system are called top-level objects. Hermes/ST objects encapsulated by other Hermes/ST objects are called encapsulated objects. A Hermes/ST object together with all of its encapsulated descendants is referred to by either of the collective terms encapsulating object or encapsulation hierarchy.

Nested encapsulation produces a tree structure for each encapsulating object. The same tree notation (parent, child, ancestor, descendant) that applied to a message tree of section 3.1.5.1, applies to an encapsulation hierarchy. Structuring the encapsulation hierarchy as a tree means that it is not possible for a Hermes/ST object to be encapsulated by more than one Hermes/ST object. This is consistent with the manner in which objects encapsulate their variables in most object-oriented languages. A Hermes/ST object can only be encapsulated by one parent, just as a variable can only be encapsulated by one object. The creation of a Hermes/ST object is specified by sending the intended encapsulating parent object a containsNew: message with various instance creation parameters. Section 3.1.4 contains some examples of Hermes/ST object creation. The encapsulation relationship between the parent and child object is maintained during the life of the created Hermes/ST object.

A Hermes/ST object’s position in the encapsulation hierarchy is used to uniquely name the object. The position of a Hermes/ST object is referred to as its name path. Name paths are generated automatically by Hermes/ST during Hermes/ST object creation and are used by Hermes/ST pointers to identify Hermes/ST objects. Each object records a list of name paths of its encapsulated children, the name path of itself and the name path of its encapsulating parent object. If the object is transactional then this information is recorded in the object store during the Hermes/ST object’s instantiation.

Figure 5.1 illustrates the Hermes/ST nested encapsulation structure. Encapsulation boundaries are represented as triangles. The components of a name path, called name path elements, that are used in this chapter have been simplified to integers for illustrative purposes. The actual Hermes/ST name paths uniquely specify the object through a concatenation of node name and sequence number. An object is denoted by using the name path as a sub-script. For example, \( O_{1.2.3} \) is encapsulated by \( O_{1.2} \) which is further encapsulated by \( O_1 \) which is, in turn, a top-level object.

5.2.8 The Access Rule

A further step towards ensuring the noCC schedulability objective is achieved by the Hermes/ST encapsulated objects access rule. Just as the encapsulation of variables by objects in an object-oriented language restricts the visibility of variables, so too Hermes/ST’s nested encapsulation structure restricts the visibility of Hermes/ST objects. With respect to illustrations such as figure 5.1, the encapsulated object access rule can be described as:

\( ^4 \)Of course, as many objects as necessary may refer (possess a Hermes/ST pointer) to an encapsulated Hermes/ST object.
CHAPTER 5. MULTI-GRANULAR CONCURRENCY CONTROL

74

Figure 5.1: An example of an encapsulation hierarchy. Circles represent Hermes/ST objects. Triangles represent encapsulation boundaries. The shaded part of a triangle represents an object's public interface. Objects $O_1$, $O_2$ and $O_3$ are top-level.

An object may send a message to any object that it can reach by leaving encapsulation boundaries (the triangles in figure 5.1) only, never by entering them.

This access rule provides an intuitive understanding of what encapsulation means. To explain by analogy, when an object encapsulates a variable it is only possible for the encapsulating object to directly access its variables. All other objects must indirectly access these variables through exported interfaces of the encapsulating object.

A more precise definition of the access rule is now presented. All examples are taken from the encapsulation hierarchy depicted in figure 5.1. A similar approach to the definition of messages and threads of section 3.1.5.1 is taken to define the access rule. As demonstrated in figure 5.1, Hermes/ST objects can be represented as nodes in an encapsulation hierarchy. This hierarchy forms a tree where the arcs represent encapsulation. Objects are named by their complete name paths. A name path is a non-empty sequence of name path elements. A name path element is an integer. Examples of the denotation of objects by name paths were given in the previous section.

The usual tree notations are used to describe the relationships between objects in an encapsulation hierarchy.

- The system top predicate ($*$) refers to the system root object $sysTop$. For example, $*(O_{1.2.2.2}) = *(O_{1.2.3}) = sysTop$
- The parent ($\cap$) and child ($\cup$) relationship holds between an object and its direct parent object. For example, $O_1 \cap O_{1.1}$ but $O_1 \not\cap O_3$. Conversely, $O_{1.1} \cup O_1$ but $O_3 \not\cup O_3$. Furthermore, $O_N \cup *(O_N)$ is true for all $N$ in name paths. In other words, each top-level object is a child of $sysTop$.
- 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 object is its own ancestor and descendant. For example, \( O_1 \leq O_{1.1} \) and \( O_{1.2.2} \leq O_{1.2.2} \) but \( O_{1.2.2.2} \not\leq O_{1.2} \). Conversely, \( O_{1.2.2.2} \geq O_{1.2} \) and \( O_{1.2.2.2} \geq O_{1.2.2.2} \) but \( O_{1.2} \not\geq O_{1.2.2} \).

The nested encapsulation access rule can now be stated. Given a sender object with a name path \( N_1 \), and a receiver object with a name path \( N_2 \), then the sender object can directly access the receiver object if the receiver object is an ancestor of the sender object or the receiver object is a child of an ancestor of the sender object. Using the notation above the rule is as follows:

\[
\text{canAccess}(O_{N_1}, O_{N_2}) \sim \text{true if and only if}
\]

\[
((O_{N_2} \leq O_{N_1}) \text{ or there exists another object } O_{N_3} \text{ such that } (O_{N_2} \cup O_{N_3} \text{ and } O_{N_3} \leq O_{N_1}))
\]

Some examples are

- \( \text{canAccess}(O_{1.2.2}, O_1) \) is true because \( O_1 \leq O_{1.2.2} \).
- \( \text{canAccess}(O_{1.2.2}, O_{1.2.2.1}) \) is true because \( O_{1.2.2.1} \cup O_{1.2.2} \) and \( O_{1.2.2} \leq O_{1.2.2} \).
- \( \text{canAccess}(O_{1.2.2}, O_2) \) is true because \( O_2 \cup \star \) is true. That is, \( O_2 \) is a child of \( \text{sysTop} \) and \( \text{sysTop} \leq O_{1.2.2} \).

With respect to figure 5.1 a Hermes/ST object (say \( O_{1.2.2} \)) is only allowed to directly invoke a method of:

- its direct children and siblings \( (O_{1.2.2.1}, O_{1.2.2.2} \text{ and } O_{1.2.1}, O_{1.2.3}) \).
- its parent and its parent's siblings (uncles) \( (O_{1.2} \text{ and } O_{1.1}) \).
- its grandparent and its grandparent's siblings (great uncles) \( (O_1 \text{ and } O_2, O_3) \).
- and so on...

Conversely, it \( (O_{1.2.2}) \) cannot directly invoke a method of:

- its grand children or deeper descendants \( (O_{1.2.2.2.1}, O_{1.2.2.2.2}) \).
- its cousins including first, second ... cousins \( (O_{2.1}, O_{2.2}, O_{2.3}) \).
- and so on...

Object interactions for which the \( \text{canAccess()} \) predicate (the access rule) is true are said to satisfy the access rule. Conversely, object interactions for which the \( \text{canAccess()} \) predicate is false are said to violate or defy the access rule.

5.2.9 The Order Rule

Another step towards ensuring the noCC schedulability objective is achieved by the Hermes/ST encapsulated object order rule. The order rule allows an application developer to limit the number of concurrency controllers employed by an application by imposing a sequence on the various types of concurrency controllers in an encapsulated structure.

A definition will make the presentation of the order rule concise. Objects that are instantiated with some combination of implicitCC: and explicitCC: instance creation parameters (i.e utilising minimal, programmable and dual locking) are henceforth collectively referred to as as-specified objects. As-specified describes the fact that these objects are concurrency controlled "as the user specified" them. Given this definition the order rule can be stated as follows:
An individual root to leaf path in an encapsulation tree can contain zero or more as-specified objects. This sequence of as-specified objects can optionally be followed by one guard object, which can be followed by zero or more noCC objects.

The order rule allows a spectrum of encapsulation hierarchies that allow various concurrency control granularities to be developed and tested. At one end of the spectrum, an encapsulation hierarchy consisting of completely as-specified objects allow applications that utilise many concurrency controllers to be developed. At the other end of the spectrum, an encapsulation hierarchy consisting of a guard object and its consequential noCC descendant objects, allows applications that utilise one concurrency controller to be developed. There are, of course, many points in the spectrum in between these two extremes. A noCC object's ancestor guard object is returned by the following:

\[
\text{ancestorGuard}(O_{N1}) \sim O_{N2} \text{ where } \\
O_{N2} \text{ is the nearest ancestor of } O_{N1} \text{ such that } \\
\text{kind}(O_{N2}) = \text{guard}
\]

### 5.2.10 The Sneaking Thread Problem

The combination of guard objects, nested encapsulation, the access rule and the order rule imply the following. A noCC object must have an ancestor guard object (order rule) and is only directly accessible by either that ancestor guard object or some descendants of the ancestor guard object (access rule). Any threads that access a noCC object via its ancestor guard object will be concurrency controlled in mutual exclusion with respect to other threads, by that guard object (refer section 5.2.3). It is however, still possible for a thread to be generated by a noCC object that satisfies both the order rule and the access rule and that defies the noCC schedulability objective.

Figure 5.2 illustrates an example of this problem. Even though the ancestor guard object \(O_1\) allows only one thread (\(T_1\)) to be active in its encapsulated subtree, the noCC object \(O_{1.1}\) can generate other threads (e.g. \(T_{1.1}, T_{1.2}\)) at \(O_{1.2}\). These two threads are not concurrency controlled, because \(O_{1.2}\) is a noCC object. This may violate the noCC schedulability objective, and is referred to as the sneaking-thread problem.
The sneaking thread problem can only occur *within* an encapsulating guard object hierarchy. The reason for this is that a noCC object cannot be a top-level object (order rule) and consequently cannot be directly accessed from outside its guarded hierarchy (access rule).

An instance of the sneaking thread problem is determined by the following predicate. Given a sender object $O_s$, receiver object $O_r$ and a message $m$, then:

\[
\text{sneakingThread}(O_s, m, O_r) \Rightarrow \text{true if and only if} \\
((\text{kind}(O_s) = \text{kind}(O_r) = \text{noCC}) \text{ and} \\
(\text{ancestorGuard}(O_s) = \text{ancestorGuard}(O_r)) \text{ and} \\
(m \text{ is an asynchronous message}))
\]

### 5.2.11 Avoiding the Sneaking Thread Problem

The concurrency controller of a guard object will correctly schedule any threads that are routed through it. As demonstrated by the sneaking-thread problem, some threads may miss being scheduled by the guard object. One solution to this problem could be to make noCC objects within a guarded hierarchy schedule any asynchronous invocation to other noCC objects within the guarded hierarchy. The problem with this approach is that in order to schedule asynchronous threads noCC objects would need to be able to perform some form of concurrency control. This is precisely what noCC objects are trying to avoid. This concurrency control would introduce scheduling costs into the noCC hierarchy. For example, a noCC object would have to be aware of the scheduling events that allow it to proceed with the execution of threads. This solution was implemented in an early version of Hermes/ST [Faz94].

A more elegant solution to the sneaking thread problem is called *rerouting*. Rerouting entails diverting any sneaking threads through the ancestor guard object. The order rule implies that such an ancestor guard object will always exist. This is consistent with the approach of insisting that the concurrency control for a noCC object always be performed by a surrogate object (refer section 5.2.2).

### 5.2.12 Rerouting Messages

Message rerouting works as follows. A message from a sender object to a noCC receiver object can be rerouted via a *surrogate* object. When a message is to be rerouted to a surrogate object, then instead of performing the method invocation at the intended receiver object, the message calls the *reroute*: method of the surrogate object. The *reroute*: method is defined in the *HermesObject* class and is therefore inherited by all user-defined Hermes/ST classes. In static multi-granular concurrency control, surrogate objects are always guard objects. From the point of view of concurrency control the invocation of the *reroute*: method of a guard object is treated in the same manner as a non-rerouted message invocation. Namely, if the guard object is implicitly concurrency controlled then a system defined mutual exclusion lock is obtained on behalf of the message. If the guard object is explicitly concurrency controlled then whatever lock has been associated with the surrogate object's *reroute*: method will be obtained.

Once some form of lock has been obtained by the rerouting thread, then the message is performed at the original intended receiver. In static multi-granular concurrency control this receiver object is a noCC object. The lock is released according to the normal Hermes/ST scheduling rules (refer section 3.1.5). All message parameters, method arguments and return values are preserved during message rerouting.
CHAPTER 5. MULTI-GRANULAR CONCURRENCY CONTROL

5.2.13 Rerouting Sneaking Threads

The rerouting of sneaking threads is performed by a *sneaking thread rerouting protocol*. This protocol is performed by Hermes/ST and does not need to be specified by the user. The routing protocol for asynchronous threads within a guarded hierarchy is as follows. If a sender object sends a message to a receiver object and both objects share the same ancestor guard object, then any messages created with the message parameter *async* are re-routed to the receiver's ancestor guard object.

For example, consider the scenario depicted in figure 5.2. Assume that all top-level objects are guard objects and the order rule is obeyed. When $O_{1,1}$ attempts to send an asynchronous message to $O_{1,2}$ the rerouting protocol will detect this as an example of the sneaking thread problem and divert the call to $O_1$.

5.2.14 Access Rule Violations - Revisited

Section 5.2.8 introduced the access rule which used the nested encapsulation structure to define a class of message invocations that satisfied the access rule and a class of message invocations that violated the access rule. The guiding principle for this dichotomy was whether or not *direct* object access was to be allowed. How should method invocations that violate the access rule be treated? There are at least two solutions to this question.

1. One solution is to disallow all such method invocations. Such an approach would imply that in order to invoke methods of objects that defy the access rule the application developer must explicitly route messages so that only legal method invocations are involved. This is the approach taken in languages that support object-variable encapsulation. Such a decision elevates the status of nested encapsulation from a structuring aide to language dogma.

   The problem with a dogmatic approach is that it does not lend itself easily to the specifications of algorithms that have to navigate an encapsulation hierarchy. For example, for a top-level object's method to invoke a method of an object that is encapsulated two levels deep such as $O_{1,2,2}$ of figure 5.1, it is necessary to invoke a method of object $O_1$ that in turn invokes a method of $O_{1,2}$ that in turn invokes the desired original method. This type of explicit message routing can be inefficient and may obfuscate the semantics of operations. Although this approach was implemented in an earlier version of Hermes/ST [Faz94] it has since been superseded.

2. A more pragmatic solution is to recognise access rule defying method invocations and to have the Hermes/ST system deal with them. Such an interpretation recognises that access rule defying method invocations are sometimes convenient and handles them in such a manner that the noCC schedulability objective is maintained. This is the approach that is currently used by Hermes/ST. An attempt by one Hermes/ST object to invoke a method of an object that would result in a violation of the access rule results in the message being re-routed by the Hermes/ST system. This system routing is described in the next section.

5.2.15 Rerouting Messages that Defy the Access Rule

The rerouting of messages that violate the access rule is performed by an *access rule violation rerouting protocol*. This protocol is automatically performed by Hermes/ST and does not need to be specified by the user. The protocol is as follows. If a message defies the access rule and the receiver object is a noCC object then the message is rerouted to the ancestor guard object of the receiver object. The order rule implies that such an
object will exist. For example, re-consider the hierarchy depicted in figure 5.1 and further assume that all top-level objects have been created as guard objects. If a noCC object, say \( O_{2,1} \), sends an access rule defying message to another noCC object, say \( O_{1,2,2} \), then Hermes/ST will recognise this and reroute the message via the receiver’s guard object. In this case the message will be rerouted via the receiver’s guard object \( O_1 \).

The access rule violation rerouting protocol is usually more efficient than forcing the user to explicitly reroute method invocations. This is because the access rule violation rerouting protocol does not require as many reroutes as explicit rerouting by the applications developer would. In the previous example the access rule violation rerouting protocol only rerouted the message once. This was to the guard object \( O_1 \). Conversely, an explicit rerouting by the applications developer would have to explicitly reroute a message from object \( O_{2,1} \) to object \( O_{1,2,2} \) via objects \( O_1 \) and \( O_{1,2} \). Naturally, the deeper the receiver object is encapsulated the more the difference between the number of reroutes required by explicit user rerouting versus system rerouting.

5.2.16 NoCC Schedulability

The combination of guard objects, nested encapsulation, the access rule, the order rule, the sneaking thread rerouting protocol and the access rule violation rerouting protocol ensure the noCC schedulability objective. The noCC schedulability objective is to ensure that only correct schedules\(^5\) will be permitted for messages that visit noCC objects. The approach taken to achieving this objective is to insist that any message that is invoked on a noCC object has had its concurrency control performed at some surrogate concurrency control object. A guard object is used for this purpose. Nested encapsulation structures objects in such a way that one guard object is the surrogate for a group of noCC objects.

Method invocations that satisfy the access rule and are part of a (non-sneaking) thread that visit a noCC object, must have a synchronous ancestor message that invoked a method of the noCC’s ancestor guard object (order rule and access rule). This ancestor guard object uses mutual exclusion to only allow one thread to be active in its encapsulation hierarchy at a time. An asynchronous thread that is created within a guarded encapsulation hierarchy is detected as an instance of the sneaking thread problem and is automatically rerouted via the ancestor guard object (sneaking thread rerouting protocol). Method invocations that violate the access rule and are destined for a receiver object whose kind is noCC are automatically rerouted to the receiver object’s ancestor guard object (access rule violation rerouting protocol).

5.2.16.1 The Static Navigation Protocol

The complete static navigation protocol can now be stated. It is used to determine how to correctly invoke a message of an object in a static multi-granular hierarchy. Any sender object may send a message to any receiver object in a static encapsulation hierarchy according to this navigation protocol. This protocol is automatically implemented by the Hermes/ST system. Given a sender object \( O_s \), a receiver object \( O_r \) and a message \( m \) the following pseudo code describes the static navigation protocol:

\[
\text{staticNav}(O_s, m, O_r)
\]

\[
\text{if} \ (\text{sneakingThread}(O_s, m, O_r) \text{ or } \left( (\text{canAccess}(O_s, O_r) \neq \text{true}) \text{ and } (\text{kind}(O_r) = \text{noCC}) \right) \text{ then}
\]

\[
\text{reroute via the ancestorGuard}(O_r)
\]

\[
\text{else}
\]

\(^5\) Transactional threads are serialised, non-transactional messages are synchronised.
CHAPTER 5. MULTI-GRANULAR CONCURRENcy CONTROL

send the message to \( O_r \).

Where the reroute mechanism was described in section 5.2.12 and send invokes the message \( m \) at the object \( O_r \) in the normal Hermes/ST fashion (see section 3.1.5).

The Hermes/ST pointer is a useful vehicle for caching information pertinent to the static navigation protocol. Recall that all transactional and volatile object invocations are made via Hermes/ST pointers. As introduced in section 3.1.4, Hermes/ST pointers are automatically created by the system when an object is created. Furthermore, as introduced in section 3.1.5.2 Hermes/ST pointers are automatically duplicated by the system when they are shared amongst remote objects.

These properties of Hermes/ST pointers make it possible for a Hermes/ST pointer to record not only the status of concurrency control (as-specified, guard or noCC) but also the position of an object's ancestor guard object. In static multi-granular concurrency control, both the status of concurrency control used and the position of the guard objects never change, so the Hermes/ST pointer's knowledge of this information is always up-to-date. This allows the static navigation protocol to be performed efficiently at the sender object. Namely, the static navigation protocol does not need to perform an expensive network access in order to acquire the information about a remote receiver object. Furthermore, by performing any necessary rerouting at the sender object, the static navigation protocol can reroute a message using only one message diversion, i.e. the reroute (cf. section 5.3.6). In both of these aspects, the protocol is efficient.

5.2.17 Specification via Highest and Lowest Markers

To facilitate the specification of object hierarchies, Hermes/ST introduces highest and lowest markers. The markers partition each encapsulation hierarchy into three regions as described by the order rule of section 5.2.9. Static multi-granular concurrency control is specified by positioning the highest and lowest markers at the same Hermes/ST object, or by omitting them entirely. The encapsulating top-level objects \( O_1, O_2 \) and \( O_3 \) of figure 5.3 illustrate different examples of this. Encapsulating object \( O_1 \), where neither lowest nor highest markers have been specified, produces a fixed, fine grain concurrency controlled, containment hierarchy that supports high concurrency. Encapsulating object \( O_2 \), which has a highest and a lowest marker positioned at the top-level object \( O_2 \), produces a fixed, coarse grain concurrency controlled containment hierarchy that supports no parallelism. Encapsulating object \( O_3 \), which has highest and lowest markers positioned at two contained objects \( (O_{3.1.1} \text{ and } O_{3.2}) \) produces the most general sort of static multi-granular concurrency control hierarchy, i.e. one in which there is high parallelism followed by no parallelism at fixed levels in the encapsulation hierarchy.

The introduction of highest and lowest markers allows the programmer to easily specify the fixed number of concurrency controllers that an application is to employ. The spectrum of fixed granularity encapsulation hierarchies introduced in section 5.2.9 can be specified by highest and lowest markers.

5.2.17.1 The Declaration of Highest and Lowest Markers

Highest and lowest markers can be explicitly specified through highest: and lowest: instance creation parameters. Both parameters take boolean arguments. For example, the highest: \texttt{true} instance creation parameter associates a highest marker with the instantiated object. Specification errors such as an object with a highest marker attempting to encapsulate another highest marked object cause an exception to be raised. If highest and lowest markers are not specified Hermes/ST uses defaults of highest: \texttt{false}; lowest: \texttt{false}.
CHAPTER 5. MULTI-GRANULAR CONCURRENCY CONTROL

5.2.18 Evaluation of Static Multi-Granular Concurrency Control

Static multi-granular concurrency control represents an interesting juncture in this thesis. As will be shown in detail in chapter 6 it allows the developer to specify applications that have good performance in known traffic conditions, can be easily specified with implicit concurrency control, can be reused and can be incrementally developed.

The assumption that traffic can be known at application specification does not always hold. Many applications have traffic patterns that vary during usage. It would be beneficial in such cases if an application could vary its concurrency control granularity according to runtime traffic conditions. Thus, if an encapsulation hierarchy finds itself in traffic that is suitable for a fine or coarse granularity of concurrency control it can react accordingly. The rest of this chapter is concerned with achieving such behaviour.

5.3 Dynamic Multi-Granular Concurrency Control

Dynamic multi-granular concurrency control allows the application developer to specify object hierarchies that can vary the granularity of concurrency control used by an application during its execution. This process is referred to as concurrency control movement. Concurrency control movement is achieved by changing the concurrency controller kind of objects in the hierarchy during execution. Two Hermes/ST supplied methods perform these changes to an encapsulation hierarchy. The downward concurrency controller movement of section 5.3.4 increases the number concurrency controllers in an encapsulation hierarchy. The upward concurrency controller movement of section 5.3.4 decreases the number concurrency controllers in an encapsulation hierarchy.
5.3.1 The Specification of Dynamic Multi-Granular Concurrency Control

The existing `highestCC:` and `lowestCC:` instance creation parameters are expressive enough to specify dynamic multi-granular concurrency control. A dynamic multi-granular concurrency controlled encapsulation hierarchy or dynamic hierarchy is specified by the placement of highest and lowest markers at different objects in an encapsulation hierarchy. In the region between the highest and lowest markers, Hermes/ST can vary the granularity of concurrency control to improve performance.

The encapsulating top-level object `04` (see figure 5.4) illustrates a containment hierarchy over which the system is free to determine the appropriate level of concurrency control granularity. Encapsulating object `04` has the highest marker positioned at itself and has not specified a lowest marker. This indicates that all objects below `04` in the contained hierarchy can be one of minimally locked, guard or noCC. The illustration depicts one possible scenario, where the guard objects exist at `04.1.1` and `04.2`. There are many other possible configurations for the encapsulating object `04`. Naturally, all configurations must comply with the order rule.

Encapsulating top-level object `05` (see figure 5.5) shows the most general form of dynamic multi-granular concurrency control. This kind of dynamic multi-granular concurrency control has highest and lowest markers positioned at different Hermes/ST objects. It has Hermes/ST objects that are above a highest marker and are always concurrency controlled as the user specified (`05`, `05.1`, `05.1.2`). It has Hermes/ST objects that contain a highest marker, which can be minimally locked or guard (`05.1.1`, `05.2`). It has Hermes/ST objects that are below a highest marker and above a lowest marker that can be minimally locked or guard or noCC (`05.2.1`, `05.2.2`, `05.2.3`, `05.2.1.1`, `05.2.3.1`, `05.2.3.2`). It also has a
CHAPTER 5. MULTI-GRANULAR CONCURRENCY CONTROL

5.3.1.1 Dynamic Hierarchies are Implicit

The observant reader will have noticed that figure 5.5 and figure 5.4 do not contain any as-specified objects at or below their highest markers. The reason for this is that dynamic multi-granular concurrency control is not defined over encapsulation hierarchies that contain explicit concurrency control. All objects at or below any highest markers must be implicitly controlled. That is, objects can only be minimally locked, implicit guard or noCC objects. One reason for this implicitCC restriction is to allow the Hermes/ST system to reason about the number of threads in an encapsulation hierarchy so that the concurrency controller movement algorithms maintain thread semantics. Another reason for the implicitCC restriction is because there is currently no provision for an application developer to express the guard locking behaviour of an explicit as-specified object (see section 7.2.2.1).

---

6In this example there is only one lowest marker specified. There can, of course, be more than one lowest marker in an encapsulation hierarchy.

7See section 7.2.2.1 for ideas about how this restriction might be removed.

8There are explicit guard objects, but these cannot be associated with individual as-specified objects.
5.3.1.2 Dynamic Hierarchies are Local

There is another restriction that is placed on objects that occur in a dynamic hierarchy. All objects that are at or below a highest marker must reside on one node. This local object restriction is not intrinsic. An earlier version of dynamic multi-granular concurrency control that does not have this restriction is detailed in [Faz94]. The local object restriction allows the concurrency control movement methods to avoid having to reason about partial state of a dynamic hierarchy caused by node failures. The atomicity of concurrency controller movement introduced by the local object restriction also negates the need to record changes to a dynamic hierarchy in non-volatile storage. Not having to access the disk aides in the performance of concurrency controller movement.

5.3.2 The Static Navigation Protocol - Revisited

Recall from section 5.2.16.1 that in order to function the static navigation protocol needs to be able to determine the kind of concurrency control employed by a receiver object and the position of the receiver object’s ancestor guard object. In a static multi-granular hierarchy this information is constant for each object and is easily cached in Hermes/ST pointers. However, in a dynamic multi-granular hierarchy, the introduction of concurrency controller movement causes both the concurrency control kind of objects and the position of guard objects to change over time. This invalidates the use of Hermes/ST pointers as a caching mechanism.

It would be impractical to try and keep all the Hermes/ST pointers up to date every time concurrency control movement takes place in some dynamic multi-granular hierarchy. This is due to the fact that there can be a many to one relationship between Hermes/ST pointers and Hermes/ST objects (refer section 3.1.5.2). Updating all the copies of Hermes/ST pointers that refer to objects within a movement hierarchy each time the hierarchy changes would be too expensive an operation.

The ability of dynamic hierarchies to change the kind of the objects they encapsulate invalidates some assumptions of the static navigation protocol, as it has been so far introduced. Instead of using the Hermes/ST pointer as a cache for information about both the concurrency control kind and the position of the receiver’s ancestor guard object, the dynamic navigation protocol acquires this information by visiting the receiver object. Both concurrency controller movement sections contain descriptions of their contributions to the dynamic navigation protocol. The dynamic navigation protocol achieves the same result as the static navigation protocol. Importantly, the noCC schedulability objective is maintained. The complete dynamic navigation protocol is presented in section 5.3.6.

5.3.3 Triggers

Concurrency control movement can be explicitly initiated via the concurrency control methods. The moveLocksDown:<level> moves the guard object down as many levels as the user specifies. The moveLocksUp method moves the guard object up to the object requesting the movement. Concurrency control movement can be implicitly initiated via trigger objects. Up triggers initiate upward concurrency controller movement, whereas down triggers initiate downward concurrency controller movement. A down trigger can be associated with each guard object. An up trigger can be associated with each minimally locked object that exists at or below the highest marker, and above the current guard object.

---

9See section 7.2.2.1 for ideas about how this restriction might be removed.
CHAPTER 5.  MULTI-GRA\-NULAR CONCURRENCY CONTROL

Legend

\[\rightarrow\] = encapsulates

* = top-level object

m = minimally locked

g = guard

n = noCC

\downarrow = down trigger

(a) Before

(b) After

Figure 5.6: “Before” and “after” states of an encapsulation hierarchy that has moved its concurrency controllers down one level.

An Hermes/ST object's trigger object can be specified via two parameters. The down Trigger: and upTrigger: instance creation parameters. The argument to both of these instance creation parameters is the name of a Smalltalk class that is a subclass of the DownTrigger and UpTrigger classes, respectively. Both the DownTrigger and UpTrigger classes define a method called trigger: that is called each time that a method is invoked on the Hermes/ST object. The Hermes/ST system invokes the trigger: method with arguments containing the message parameters and Hermes/ST pointers to both the sender and receiver object. This information and any other information such as the time of the day can be used by the application developer in the trigger method.

If concurrency controller movement is not to be initiated, then the trigger: methods should return false. If upward concurrency controller movement is to be initiated then UpTrigger trigger: should return true. If downward concurrency controller movement is to be initiated then DownTrigger trigger: should return the number of levels that the guard object is to be moved.

5.3.4 Downward Concurrency Controller Movement

Figure 5.6 (a) illustrates an example encapsulation hierarchy before it has moved its concurrency controllers down one level. The hierarchy consists of a single guard object (O1) and its associated down trigger encapsulating a sub-hierarchy of noCC objects (O1.1, O1.2, O1.2.1, O1.2.2). Figure 5.6 (b) shows the same example encapsulation hierarchy after it has moved its concurrency controllers down. The former guard (O1) object has become a minimally locked object and all encapsulated children (O1.1, O1.2) have become guard objects. More deeply encapsulated descendants (O1.2.1, O1.2.2) have remained guard objects.

The downward concurrency controller movement, or downward movement of figure 5.6 can be viewed as moving a guard object down some number of levels in the encapsulation hierarchy. This involves replacing an entire level of noCC objects with guard objects, and installing minimally locked objects above the guard objects. The application developer can specify how many levels to move the guard object down. Therefore, in the example structure illustrated by figure 5.6 (a) it is possible to move the guard object down as many as three levels. Such a movement would result in an “after” hierarchy that consists entirely of minimally locked objects.

Because there can be multiple lowest markers for each highest marker, or no lowest
marker specified, the movement algorithm can not always move guard objects down the number of levels that the application programmer specified. In order to satisfy the order rule, guard objects must not move below any lowest markers. Figure 5.5 serves to illustrate. If object \( O_{5.2.1} \) attempts to move its guard object down two levels then \( O_{5.2.1.1} \) will become a minimally locked object. \( O_{5.2.1.2} \) is marked lowest and will therefore become a guard object. Objects \( O_{5.2.1.1} \) and \( O_{5.2.1.2} \) will remain noCC objects.

A specification of the number of levels to move guard objects down as being \#all-Levels will cause the move down algorithm to move guard objects down as far as possible. In order to satisfy the order rule "as far as possible" is interpreted as no further than the lowest marker. If a lowest marker has not been specified then "as far as possible" is "past the last object" in the encapsulation hierarchy.

5.3.4.1 Downward Movement Pre-Condition

Before the move down algorithm can be performed, any thread active in the encapsulated hierarchy of the down-initiating object must terminate. One reason why there can be no threads active in the encapsulated hierarchy is because there is not enough information in the "before" hierarchy to construct the "after" hierarchy (refer figure 5.6). For example, it is not possible to convert a guard object into a minimally locked object while there is a thread active in the guard object. The problem is that a guard object does not contain any concurrency control information for individual variables. Minimally locked objects (one concurrency controller per variable) need this information.

The same argument applies to the conversion of noCC objects to guard objects. NoCC objects perform state restoration and persistence handling (refer 5.2.1). However, because state restoration does not record information about individual variable reads it is impossible to determine which parts of a noCC object have been read by a particular thread.\(^{10}\)

5.3.4.2 The Downward Movement Process

The following sequence of steps will perform a downward concurrency control movement in such a way that noCC schedulability objective is maintained:

1. Introduce "restricted concurrency" at the down-initiating object.
2. Wait until hierarchy contains no threads.
3. Convert the hierarchy un-interruptibly.
4. Schedule any delayed messages at the down-initiating object.

Step 1 is used to ensure the hierarchy is suitable for conversion to a guarded (mutual exclusion) hierarchy. Step 1 changes the concurrency controller of the down-initiating guard object to support restricted concurrency. Restricted concurrency will only schedule messages that are both Hermes/ST schedulable and that are ancestors or descendants of granted message requests. Restricted concurrency allows any existing thread active in the downward moving hierarchy to continue to execute, but prevents any new top-level threads from entering the hierarchy via the down-initiating object. If there is a thread active in the hierarchy when restricted concurrency is activated it can still generate synchronous and asynchronous, transactional and non-transactional messages. Asynchronous messages (sneaking threads) are rerouted to the guard object in the normal fashion.

\(^{10}\)The same is not true for write information. As state restoration is performed on individual variables it is possible to reconstruct which threads wrote which variables. This information could be used to reconstruct individual variable locks.
Because all threads eventually complete and no new top-level threads are allowed to commence, the hierarchy will eventually be emptied of messages (and hence threads). Step 2 is used to wait for this condition. The precise condition under which the hierarchy has no active threads is that the size of the set of granted requests of the down-initiating object is zero. Once this condition is achieved, restricted concurrency maintains an empty hierarchy. An empty hierarchy is a stronger predicate than restricted concurrency. The former allows no messages to be scheduled, while the latter allows certain synchronous and asynchronous, transactional and non-transactional messages to be scheduled.

Step 3 traverses and converts the downward movement hierarchy according to the application developer’s specification of the number of levels. This traversal is uninterruptible with respect to Hermes/ST threads. The down-initiating object is converted to a minimally locked object. Descendants of the down-initiating object that are between the highest and lowest markers are converted according to the level argument of the moveLocksDown method. After the completion of the traversal algorithm the order rule must be satisfied. Therefore, guard objects are not moved lower than any objects marked as lowest in the hierarchy. Traversals use the nested encapsulation information stored when objects are created (refer section 5.2.7) to find encapsulated objects.

Step 4 schedules any messages that were delayed by the down-initiating object during the previous five steps. There are potentially two types of pending messages. Messages that have been rerouted and messages that have not. Non-rerouted messages are scheduled according to the normal (non restricted) concurrency control rules. Rerouted messages need to be retried by the rerouting object. This is because such messages are now in the pending queue of a minimally locked object. They need to be processed by a guard object. The dynamic navigation protocol is informed via a #retryRerouted return value from the now minimally locked down-initiating object to retry the reroute of the message. The rerouting noCC receiver object will resend the reroute to its new ancestor guard object.

5.3.4.3 NoCC Schedulability

The local object restriction of section 5.3.1.2 implies that no partial failures of the downward movement can occur. The downward concurrency control movement either completes or upon the recovery of the node the hierarchy is returned to its originally specified state. Furthermore, the conversion process is performed uninterruptibly. The combination of no partial failures and an uninterruptible conversion of the hierarchy makes the downward movement atomic with respect to Hermes/ST threads. Therefore, only the “before”, “after” or “originally specified” hierarchies are visible to Hermes/ST threads. It is possible for messages to be rerouted to the downward initiating object while the hierarchy is waiting to be emptied. Any such messages are delayed for the duration of the downward movement (restricted concurrency) and retried after the completion of the downward movement (dynamic navigation protocol). The complete dynamic navigation protocol is presented in section 5.3.6.

11 Those messages that are ancestors or descendants of existing granted messages.
12 Un-interruptible traversals are trivially accomplished in Smalltalk. This is because the ObjectWorks Smalltalk does not pre-emptively schedule its processes. The lack of pre-emptive scheduling and the local object restriction guarantee that the traversal will be un-interruptible (in fact, atomic) with respect to Hermes/ST threads. In systems that support pre-emptive scheduling an un-interruptible traversal would entail more work.
CHAPTER 5. MULTI-GRANULAR CONCURRENCY CONTROL

5.3.5 Upward Concurrency Controller Movement

Figure 5.7 (a) illustrates an example encapsulation hierarchy before it has moved its concurrency controllers up. The hierarchy consists of a minimally locked object \( O_1 \) and its associated up trigger, which encapsulates a sub-hierarchy of Hermes/ST objects \( O_{1.1}, O_{1.2}, O_{1.2.1}, O_{1.2.2} \). Figure 5.7 (b) shows the same example encapsulation hierarchy after it has moved its concurrency controllers up. The former minimally locked object \( O_1 \) has become a guard object, and all encapsulated descendants \( O_{1.1}, O_{1.2}, O_{1.2.1}, O_{1.2.2} \) have become noCC objects.

Upward concurrency controller movement (or upward movement) can be viewed as moving the guard object up to the level of the up-initiating object. Unlike downward concurrency control movement the number of levels to move the guard object is implicit in the position of the up-initiating object and need not be specified by an application developer.

5.3.5.1 Upward Movement Pre-Condition

Unlike downward movement it is not necessary to wait until all threads active in the up-initiating object's hierarchy have terminated. Upward movement can be performed when there is one\(^{13} \) thread active in the hierarchy. Upward movement in the presence of one thread is possible because both minimally locked and guard objects can be converted to noCC objects while a thread is active in them. Minimally locked objects need only discard their programmable and minimal lock concurrency controllers. Similarly guard objects need only discard their mutual exclusion concurrency controllers. Only one thread is permitted to be active in the hierarchy during an upward movement because any more than one thread would defy the noCC schedulability objective in the resultant guard hierarchy. Section 6.3.1.2 discusses the utility of allowing one thread to be active during upward movement.

5.3.5.2 The Upward Movement Process

The following sequence of steps will perform an upward concurrency control movement in such a way that the noCC schedulability objective for threads is maintained:

\(^{13}\text{Refer to sections 5.2.2 and 3.1.5.1.6 for the definition of one thread.}\)
1. Introduce "restricted concurrency" at the up-initiating object.
2. Un-interruptibly find "phantom threads" and activate "object rerouting" to the up-initiating object.
3. Wait until hierarchy contains only one thread.
4. Convert the hierarchy un-interruptibly.
5. Schedule any delayed messages at the up-initiating object.

Step 1 changes the concurrency controller of the up-initiating object to support restricted concurrency. As was the case with downward movement, restricted concurrency only schedules messages that are ancestors or descendants of messages in the set of granted message requests. This allows existing threads active in the upward moving hierarchy to continue to execute but prevents any new top-level threads from entering the hierarchy via the up-initiating object.

The upward movement algorithm is further complicated because the up-initiating object does not have complete knowledge of all the threads active in its hierarchy. Nested asynchronous messages may have been generated by minimally locked objects below the up-initiating object before the upward movement commenced. These so called phantom threads may not have invoked a method of the up-initiating object and therefore the up-initiating object will not be cognizant of their existence. Phantom threads need to be accounted for in the calculation of the number of threads active in the upward moving hierarchy. Step 2 performs an un-interruptible traversal of the upward moving hierarchy that causes the up-initiating object to be made aware of any phantom threads. Step 2 also ensures that access rule defying messages and sneaking threads do not surreptitiously enter the upward moving hierarchy during an upward movement. To achieve this each object in the hierarchy has "object rerouting" activated. Activating object rerouting at an object causes all asynchronous and access rule defying messages to that object to be rerouted to the up-initiating object.

By the time that step 3 of upward movement process is reached the up-initiating hierarchy has complete knowledge of all threads that are active in that hierarchy. Furthermore, the use of object rerouting ensures that any access rule defying method invocations or asynchronous messages are rerouted through the up-initiating object. From this point on the up-initiating object has control any threads active and any threads attempting to enter the hierarchy. The up-initiating object uses restricted concurrency to quiesce the hierarchy. Because all threads eventually complete and no new asynchronous threads are allowed to commence, the hierarchy will eventually be quiesced to the state that only one thread is active, at which point the hierarchy can be converted (see section 5.3.5.1). When only one thread is active, this thread cannot interleave with itself by definition (see section 5.2.2) and therefore does not need to further delay the conversion of the hierarchy. The precise quiescence condition is that all messages in the granted set of requests are synchronous ancestors and descendants of each other. Once this condition is achieved, restricted concurrency maintains the quiescence condition. Quiescence is a stronger predicate than restricted concurrency. The former allows only synchronous ancestor and descendant messages to be scheduled, while the latter allows synchronous and asynchronous descendant and ancestor messages14 to be scheduled.

Step 4 restructures the upward movement hierarchy so that the up-initiating minimally locked object is a guard object and all objects below are noCC objects. This restructuring is un-interruptible with respect to Hermes/ST threads. This includes the one thread that may be active in the hierarchy during the conversion. When converting a minimally locked or guard object to a noCC object all concurrency control information is discarded.

14 These messages can generate transactions.
There will be no pending requests but there may be one or more granted requests. When converting an up-initiating minimally locked object to a guard object, the pending and granted requests of the up-initiating object are duplicated in the target guard object. The conversion of the hierarchy in step 4 is performed by another un-interruptible traversal of the hierarchy. This traversal also de-activates the object rerouting for objects involved in the movement. During this traversal objects retain the position of the up-initiating object for use as their ancestor guard object.

Step 5 schedules any messages that were delayed during the previous four steps. These messages are at their now correct guard object and scheduled according to the normal (non restricted) concurrency control rules.

5.3.5.3 NoCC Schedulability

The local object restriction of section 5.3.1.2 implies that no partial failures of the upward movement can occur. The upward concurrency control movement either completes or upon the recovery of the node the hierarchy is returned to its originally specified state. Furthermore, each traversal of the upward moving hierarchy (steps 2 and 4) is performed un-interruptibly. The combination of no partial failures and an un-interruptible conversion of the hierarchy makes these traversals atomic with respect to Hermes/ST threads. The first traversal activates object rerouting (see section 5.3.6 below) to ensure that restricted concurrency achieves the condition that at most, only one thread is active in the hierarchy before the upward movement is commenced. All other threads are routed to the up-initiating object which delays them (restricted concurrency) until the completion of the movement. The one thread that can be active in a upward moving hierarchy, by definition cannot interleave with itself. The second traversal converts the upward moving hierarchy into its after state. The atomicity of this traversal with respect to the one thread that may be present in the hierarchy ensures that if there is such a thread, it does not see any partial conversion state of the upward moving hierarchy.

5.3.6 The Dynamic Navigation Protocol

The complete dynamic navigation protocol can now be stated. It is used to determine how to correctly invoke a message of an object in a dynamic multi-granular hierarchy. Any sender object may send a message to any receiver object in a dynamic Hermes/ST encapsulation hierarchy according to this navigation protocol. This protocol is automatically implemented by the Hermes/ST system. Given a sender object $O_s$, a receiver object $O_r$ and a message $m$, then the following pseudo code describes the dynamic navigation protocol:

```plaintext
dynamicNav(O_s, m, O_r)
if (isRerouted(O_r)) then
  if ( (canAccess(O_s, O_r) \neq true) or (message is asynchronous) ) then
    reroute the message via the upInitiator(O_r)
  else
    perform the message
else if ( sneakingThread(O_s, m, O_r) or
  ( (canAccess(O_s, O_r) \neq true) and (kind(O_r) = noCC)) ) then
  reroute the message via the ancestorGuard(O_r)
  retry the reroute if #retryRerouted is returned
else
  perform the message
```

CHAPTER 5. MULTI-GRANULAR CONCURRENCY CONTROL

Where \( \text{isRerouted}(O_x) \) is a predicate that returns true if the object \( O_x \) has had object rerouting activated by an upward movement. Furthermore, \( \text{upInitiator}(O_x) \) is a function that returns a reference to the object that has been made the surrogate concurrency controller for an object (i.e. the up-initiating object).

The inability of the dynamic navigation protocol to use the Hermes/ST pointer as a cache for information about both the concurrency control kind and the position of the receiver's current surrogate object affects its implementation and performance. Specifically the dynamic navigation protocol is performed at the receiver object. The reason for this is so that expensive network accesses to determine information about remote receiver objects can be avoided. Performing rerouting at the receiver object rather than at the sender object means that in the dynamic navigation protocol, rerouted messages must be passed from the sender to the receiver and then to the surrogate rerouting object. In the static navigation protocol rerouted messages are passed from the sender object directly the rerouting object. The choice between expensive remote object invocations (perform the protocol at the sender object) or more message rerouting (performing the protocol at the receiver object) therefore represents a trade off for the dynamic navigation protocol. Because remote object invocations are more expensive then message reroutes (they involve network communications) and because remote object invocations should be more common than message reroutes, the decision to perform the dynamic navigation protocol at the receiver object was taken.

5.3.6.1 NoCC Schedulability

Without any concurrency control movements active in a dynamic hierarchy the dynamic navigation protocol ensures the noCC schedulability objective in the same manner as the static navigation protocol. Namely, a combination of guard objects, nested encapsulation, the access rule, the order rule, the sneaking thread rerouting protocol and the access rule violation rerouting protocol still ensure the noCC schedulability objective. The sneaking thread and access rule violation rerouting protocols now obtain information about the receiver object by visiting the receiver object rather than using the Hermes/ST pointer as a cache. This affects performance but does not change semantics.

Concurrency control movement introduces the potential for partial state of a dynamic hierarchy to exist that could invalidate the dynamic navigation protocol. The downward movement process uses a combination of restricted concurrency, atomic conversions and message retries to ensure that no thread can see this partial state (refer section 5.3.4.3). The upward movement process uses a combination of restricted concurrency, object rerouting and atomic traversal to ensure that the one allowable thread does not see any partial state of an upward moving hierarchy (refer section 5.3.5.3).

5.3.7 Deciding Whether to Statically or Dynamically Reroute

When performing a message invocation the sender object needs to be able to decide whether to use the static or the dynamic navigation protocol when accessing the receiver object. Because the type of multi-granular concurrency control hierarchy (static or dynamic) that an object exists in does not change during the life of a Hermes/ST object this information can be cached in the Hermes/ST pointer during object creation. An object created below or at a highest marker exists in a dynamic hierarchy. An object created in a static hierarchy exists in a static hierarchy. The navigation protocol to use depends on the type of multi-granular hierarchy of the receiver object. If the receiver object is in a static hierarchy the static navigation protocol is used. If the receiver is in a dynamic hierarchy the dynamic navigation protocol is used.
5.3.7.1 Conflicting Movement Initiations

Concurrency control movements can be caused by triggers or activated by the application developer. There may be multiple attempts to perform concurrency control movements at the same time. Individual dynamic hierarchies may perform movements in parallel but within one dynamic hierarchy only one movement can be performed at a time.

When concurrency control movements are requested by triggers or by the application developer all requests to perform concurrency controller movements are regulated by the Hermes/ST system. Hermes/ST allows only one concurrency control movement at any one time per dynamic hierarchy. A node-wide dictionary is used to track the status of individual dynamic hierarchies.
Chapter 6

Discussion

6.1 Introduction

This chapter contains two principal parts. Section 6.2 contains an evaluation of Hermes/ST multi-granular concurrency control. Multi-granular concurrency control is evaluated against the four criteria introduced in chapter 3. These are performance, incremental development, reuse and ease of specification. This evaluation shows how the use of multi-granular concurrency control can improve system performance. Furthermore, this evaluation shows that this improved performance is achieved in an easily specified manner and can be incrementally developed and reused.

The second part of the chapter is found in section 6.3 and contains a comparison with related systems. Hermes/ST has introduced several novel ideas. Each of these ideas is compared with related systems.

6.2 Evaluation of Multi-Granular Concurrency Control

6.2.1 Performance

This section evaluates the performance of multi-granular concurrency control in terms of one extended example, the deposit method of the distributed bank. Section 3.2 introduced the notion of throughput and some factors related to concurrency control granularity that affect it. This section relates these and other factors to the example deposit method of the distributed bank. These factors included: the multi-programming level, the level of lock contention, the degree of processor utilisation by a thread, the level of deadlock, and lock management costs. This section also presents the results of the experiment and their interpretation. Section 6.2.1.11 shows how the approach generalises to other methods and section 6.2.1.12 discusses the inherent costs of multi-granular concurrency control.

6.2.1.1 Throughput

Throughput is one indicator of system performance and measures the number of threads\(^1\) that the system can perform in a unit of time. In this evaluation only transactional threads are used, so throughput measures the number of transactions per second. Obviously, the higher the value of throughput the better.

\(^1\)Recall, from section 3.2.1 that the term thread is a collective term that includes transactional and non-transactional threads.
6.2.1.2 Concurrency Control Granularity

Consider depositing money to an account at a distributed bank's branch. Each branch object is instantiated as a minimally locked object and has a binary accounts tree which contains individual accounts. These accounts contain the pertinent information about each account holder. For the purposes of this section, Hermes/ST multi-granular concurrency control is applied to the branch's binary accounts tree. This means that the account tree can be instantiated with many possible granularities of concurrency controller. During this evaluation three granularities of concurrency control are considered.

1. Static multi-granular concurrency control with one concurrency controller. In this case the guard object is at the root of the binary tree and all other binary tree nodes and the accounts that they contain have noCC concurrency controllers. This type of account tree is referred to as a guarded account tree and models a coarse granularity of concurrency control.

2. Static multi-granular concurrency control with many concurrency controllers. In this case all objects that constitute the accounts binary tree are minimally locked. This type of accounts tree is referred to as a minimally locked account tree and models a fine granularity of concurrency control.

3. Dynamic multi-granular concurrency control. Such an account tree can be a minimally locked or a guarded account tree. There is a trigger associated with the root of the binary tree that uses the upward and downward concurrency controller movements to convert the tree between minimally locked and guarded. This type of account tree is referred to as a dynamically locked account tree and models a variable granularity of concurrency control.

6.2.1.3 Multi-Programming Level

The multi-programming level (MPL) represents the total number of threads that are active while performing a specific thread. As a general rule thread execution time increases as the MPL increases [ACL87, BHG87, DA91]. The reader that has used a time sharing system has probably experienced this sort of phenomena. As the system load increases more and more time is needed to complete each task. Beyond the obvious condition that tasks get sequenced there are extra overheads such as task context switches [Tan87, SPG91]. In this evaluation the MPL is modelled as a changing input parameter.

6.2.1.4 Lock Contention

Lock contention causes threads to wait until the thread holding a conflicting lock has released that lock. Transactional threads employ strict two phase locking, which has the consequence that locks can only be released at the completion of a transaction. Therefore, transactional threads that exhibit lock contention become sequenced at the first point of contention. This sequencing results in less concurrent activity in the system, which in turn affects throughput (see section 6.2.1.5). There are two types of contention in the Hermes/ST system that can affect throughput.

1. Necessary contention occurs when two threads must access the same object or variable in a conflicting manner. In this evaluation parallel deposits to the same account would be an example of necessary contention. Such deposits need to write to a common balance variable of the account. This form of contention is important. but
unrelated to the granularity of concurrency control and is not modelled in this evaluation.

2. Unnecessary contention occurs when the granularity of concurrency control is too coarse. For example, a branch can theoretically support as many parallel deposits as it has accounts. However, a branch with a guarded account tree can only execute one deposit at a time. Unnecessary contention affects throughput by reducing the amount of parallel activity that is allowed by an the system. This form of contention is modelled in this experiment via the different types of account trees used. A minimally locked account tree allows concurrent deposits to different accounts and is meant to model a system with a low degree of unnecessary contention. A guarded account tree only allows one deposit method to be active at a time and is meant to model a system with a high degree of unnecessary contention.

6.2.1.5 Processor Utilisation by Threads

Threads utilise the central processing unit (CPU) to varying degrees. For example, a thread that includes some non-CPU bound task (e.g. waiting for the user to type a character) does not fully utilise the CPU. Less than full utilisation of the CPU by a thread results in thread idle time. Operating systems attempt to keep the CPU fully utilised by scheduling other threads during thread idle time.

Distributed computer systems increase the likelihood of thread idle time. This is because remote object invocations introduce thread idle time. As introduced, a transactional deposit method includes a small component of thread idle time. This thread idle time is due to the persistence handling required by a transactional deposit message. Increased thread idle time (remote object accesses) can be introduced into the deposit example in several ways. For example, it may be necessary to have some form of password authentication during deposits. In distributed systems such as OSF’s Distributed Computing Environment (DCE) [Lib92, Cor94] the security server is typically not contained on every node in the system. The remote access involved in accessing such a security server would increase a deposit thread’s idle time.

As another example, there could be constraints over the various accounts that the account holder owns. For example, there could be a rule such as the following. “The sum of all the positive and negative account balances for each account holder must always be positive”. If account holders owned different accounts at different branches then this type of rule would mean that each deposit would have to visit possibly remote branches to ensure the rule was satisfied. Any visits to remote branches would increase a deposit thread’s idle time.

It should be clear that the assumption that distributed applications will generally contain remote object invocations does not require a leap of faith by the reader. Because Hermes/ST is a distributed programming environment, the effect of thread idle time is modelled in this evaluation. To keep the application simple, various remote object invocation times are simulated via deposit idle times.

If, in a concurrent execution of threads, each thread has no idle time then such an execution will result in lower throughput than a sequential execution of the same threads. This phenomenon is due to the increased thread creation and management overheads that are introduced by concurrency. If each thread contains idle time then a concurrent execution of threads may increase the throughput of the system. This is because the CPU intensive part of the execution of one thread can proceed in parallel with the idle time of

\[\text{Normally, the invoking thread must wait for a result to be returned.}\]

\[\text{However, response time and fairness will be improved.}\]
other threads. An increase in throughput will occur if the gains of concurrently executing threads that contain idle time, outweigh any increased thread management overheads due to this concurrency.

6.2.1.6 Deadlock

Deadlock is detected in Hermes/ST via a timeout mechanism. A message involved in a deadlocked thread can be retried or can abort the invoking thread. Thread abortion due to deadlock decreases the throughput of the system. This is because the aborted thread has wasted CPU time.

Deadlock can occur within and amongst minimally locked hierarchies as well as amongst guarded hierarchies. Guarded hierarchies cannot deadlock internally as there is only ever one thread active in them at any one particular point in time. As an example of how deadlock occurs in a minimally locked hierarchy consider two transactions that are attempting to deposit to the same account. The following piece of code is eventually executed by both transactions:

```
self balance: self balance + amount
```

If both transactions interleave in such a manner that they both read the balance variable before⁴ they attempt to write to it, then a deadlock will occur. Deposit deadlocks were easily avoided via a short-term Smalltalk semaphore. Naturally, the expected level of deadlock and the ease of deadlock elimination will vary from application to application.

6.2.1.7 Lock Management Costs

The cost of lock management in Hermes/ST is related to the number of concurrency controllers used by an application. The lock management costs of a guarded account tree are lower than the lock management costs of a minimally locked account tree. A guarded account tree has one lock for each active deposit message. A minimally locked account tree has one lock for each variable accessed by each active deposit message.

Lock management costs affect throughput by changing the execution time of a deposit method. The deposit method of a minimally locked account tree will take longer to execute than a deposit method of a guarded account tree. In this evaluation the type of account tree in use, models the lock management costs.

6.2.1.8 Hermes/ST Overheads

A single transactional deposit to the root of the guarded (one lock) account tree consumes, on average, .322 of a second. This time is slow, with the time mostly spent performing the Hermes/ST overheads of a transactional deposit method. These include writing to the log, creating transaction data structures and Hermes/ST message invocation. These overheads are large for two reasons.

1. Most significantly, Hermes/ST system is a research prototype and any aspects of transactional systems that are not pertinent to this thesis (e.g log handling, transaction data structures, method invocation) have not been optimised.

⁴This is possible in Hermes/ST, even without pre-emptive scheduling. For technical reasons individual variable accesses of minimally locked variables yield the processor.
2. To a lesser degree the fact that Hermes/ST is built on top of Smalltalk adds to its overheads. Smalltalk has many factors such as virtual machine interpretation and execution time message lookup which cause it to have less than optimal performance.

6.2.1.9 The Deposit Experiments

Figures 6.1, 6.2, 6.3 and 6.4 graph the relationship between throughput and the MPL as CPU utilisation is varied for a transactional deposit method. The throughput for both a guarded and minimally locked account tree is measured. Unfilled squares represent the throughput values for a guarded account tree, whereas filled squares represent the throughput values for a minimally locked account tree.

The vertical axis of each figure depicts throughput measured as the number of transactional deposits per second. The horizontal axis depicts the MPL at which a given throughput is achieved. All graphs vary the MPL from 1 to 25. A MPL of \( n \) is accomplished by performing a deposit while there are exactly \( n - 1 \) other deposits active in the account tree. Each branch has been pre-installed to contain a balanced accounts binary tree with thirty two account names. Each deposit, including the background deposits that constitute the MPL, is performed to an account name chosen in a cyclic fashion from an ordered list of the pre-installed account names. Such a mechanism eliminates necessary contention (section 6.2.1.4).

Thread idle time for deposit methods is executed after each deposit changes the balance of its intended account. Figures 6.1 has no thread idle time, figure 6.2 has a thread idle time of 1 second, figure 6.3 has a thread idle time of 2 seconds and figure 6.4 has a thread idle time of 5 seconds.

6.2.1.9.1 Conclusions Drawn  Figures 6.1, 6.2, 6.3 and 6.4 show that below a certain level of CPU utilisation by a thread there is no one level of concurrency control granularity that is best for all multi-programming levels.

With full utilisation of the CPU by a thread, as depicted in figure 6.1, the throughput of a guarded account tree is always higher than that of a minimally locked account tree.

\(^5\)Of course, Smalltalk also has many other more desirable properties.
CHAPTER 6. DISCUSSION

Figure 6.2: Throughput versus Multi-Programming Level (MPL) - 1 second idle time

Figure 6.3: Throughput versus Multi-Programming Level (MPL) - 2 second idle time

Figure 6.4: Throughput versus Multi-Programming Level (MPL) - 5 second idle time
This throughput pattern is to be expected because with full CPU utilisation, throughput is most affected by the cost of thread execution. A minimally locked account tree has lower throughput than a guarded account tree due to increased parallelism and increased lock management costs. The execution costs of a minimally locked account tree versus a guarded account tree is exaggerated by the un-optimised Hermes/ST implementation. For technical reasons each minimally locked variable access is performed as a process in Hermes/ST. A more realistic implementation would avoid such non-intrinsic overheads. Such an implementation would only strengthen the results of this thesis. With less overheads, the throughput of the minimally locked curves would uniformly rise (see saturation below).

Figures 6.2, 6.3 and 6.4 shows how decreasing the level of CPU utilisation by a thread (increasing thread idle time) changes the relationship between the number of concurrency controllers and throughput. In these figures the throughput of a minimally locked account tree rises to above that of a guarded account tree and falls back below it as the MPL is increased. The intersection of the throughput of a guarded account tree with the throughput of a minimally locked account tree is called a cross-over point. Figure 6.2 shows both cross-over points whereas figures 6.3 and 6.4 show the initial cross-over point with the final cross over point falling outside the range of measured MPL.

As the percentage of CPU utilisation of a deposit method falls two things happen.

1. Cross over points become further apart. In figure 6.2 the cross-over points are at MPL 1.5 and MPL 15. In figure 6.3 the cross over-points are at MPL 1.5 and (by extrapolation) MPL 30.

2. The difference between the maximum throughput of a minimally locked accounts tree and a guarded account tree gets larger. In figure 6.2, figure 6.3 and figure 6.4 these differences are .24, .44 and .54 respectively.

Such cross-over points may or may not exist for a given distributed application. Section 6.2.1.10 shows how Hermes/ST can use dynamic multi-granular concurrency control to take advantage of cross-over points. If an application does not have cross-over points then Hermes/ST can still use static multi-granular concurrency control to determine the best granularity of concurrency controller (across the expected range of MPLs) to achieve maximum throughput (see section 6.2.2).

Finally, the perturbations in the shape of the minimally locked and guarded account throughput curves are caused by Hermes/ST implementation factors. For example, each node contains one Smalltalk Dictionary structure to store the relationships between HermesST threads and the (more than one) Smalltalk processes that are used to implement the thread. A Smalltalk dictionary uses hashing as an indexing mechanism and thus does not exhibit constant creation, deletion, and searching times.

6.2.1.9.2 Analysis of the Curves In each graph the throughput of the guarded hierarchy is approximately constant. The throughput of a guarded hierarchy is close to constant because a guarded hierarchy only allows one deposit method to be active at any one time. Each guarded hierarchy achieves about the maximal throughput allowed (given the limits of Hermes/ST). For example, in figure 6.1 the maximum expected throughput is $\frac{1}{322} = 3.1$ deposits per second. The average throughput achieved in figure 6.1 is about 3.1 deposits per second.

For a minimally locked hierarchy the value of throughput as the MPL increases also follows a pattern. Throughput rises to a peak and then drops off. This pattern conforms to the throughput versus MPL pattern found in the literature [ACL87, BHG87, DA91].
For figure 6.1, consider the peak to be at MPL 1. The rising part of the throughput line for a minimally locked account tree is due to benefits of executing thread idle times in parallel. For example, the sequential execution of two deposits each containing a 2 second idle time should only take about $2 + .322 + 2 + .322 = 4.644$ seconds whereas a parallel execution of the same methods should only take $2 + .322 = 2.322$ seconds.

Throughput rises approximately proportional to the MPL until Hermes/ST saturation occurs. Saturation is the MPL at which any thread idle time is completely used. Unfortunately, because of the high execution cost of a deposit, saturation occurs at a low MPL in Hermes/ST. For example, with an idle time of 1 second and an execution time of .322 seconds\(^6\) the Hermes/ST system nearly saturated at MPL 3. Figure 6.4 shows this proportional rise the most clearly. To execute a deposit at MPL 1 with a 5 second delay should take about $5 + .322 = 5.322$ seconds. This equates to a throughput of .19. Figure 6.4 has a throughput of .187 at MPL 1. As the MPL rises the throughput should rise proportionally until the saturation point is reached. Up to a MPL of about 8 the throughput of figure 6.4 does just that. The rise in throughput, although linear, is not a multiple of MPL because as MPL rises so to does the time taken to execute a deposit.

The falling part of the throughput line for a minimally locked account tree is caused by the costs of executing parallel deposits eventually outweighing the previously mentioned benefits of executing threads that contain idle time in parallel. This happens because the execution time of each deposit eventually becomes larger as there are more and more concurrent deposits. The increase in deposit execution time is caused by the increased thread(see section 6.2.1.3) and lock(see section 6.2.1.7) management costs that result from concurrency increases.

6.2.1.10 Utilising Cross-Over Points

Depending on the traffic, a particular object structure can change the number of concurrency controllers it possesses in order to improve throughput. Dynamic multi-granular concurrency control allows the application developer to write triggers that respond to changes in the MPL. These triggers can initiate concurrency controller movements that in turn change the number of concurrency controllers so that better throughput can be achieved.

For example, consider figure 6.2. There are two cross-over points. The first is at MPL 2 and the second is at MPL 15. Below a MPL of 2, better throughput is achieved by a guarded account tree. Between MPL 2 and MPL 15 better throughput is achieved by a minimally locked account tree. Above MPL 15, better throughput is again achieved by a guarded account tree. To take advantage of these cross-over points account tree triggers can work in the following manner. The down trigger calls moveLocksDown:#allLevels if the MPL is in the range 2 to 15. This moveLocksDown:#allLevels method converts a guarded account tree into a minimally locked account tree. The up trigger calls moveLocksUp if the MPL rises above 15 or falls below 2. This moveLocksUp method converts a minimally locked account tree into a guarded account tree.

For a dynamically locked version of an account tree with thirty two accounts the cost of a downward concurrency controller movement for a guarded account tree is 2.76 seconds on average. Using the same dynamically lock account tree, the cost of an upward concurrency controller movement (with no threads active) is 3.11 seconds on average. The cost of these movements is heavily influenced by the slow object invocation times in Hermes/ST (see section 6.2.1.12.2). Naturally the cost of a concurrency controller movement is proportional to the number of objects to be converted.

---

\(^{6}\)This time will rise as the MPL rises.
Concurrency controller movement will produce throughput benefits whenever the sum of the differences in execution times is greater than the cost of concurrency controller movement. For example, in figure 6.2, at MPL 4, the minimally locked account tree takes .98 seconds to perform a deposit whereas a guarded account tree takes 1.30 seconds to perform the same deposit. The difference in execution times is .32 of a second. Therefore, at a MPL of 4, it is worthwhile converting a guarded account tree to a minimally lock account tree (via \texttt{moveLocksDown:\#allLevels}) if more than $\frac{.75}{.32} = 8.63$ parallel deposits are expected.

Of course, an application would use the expected traffic patterns to calculate any throughput difference over its entire MPL range. In general, if an application has one or more cross-over points and the traffic patterns are such that the cost of a concurrency controller movement (including the time taken to quiesce the hierarchy) is less than the gains in throughput to be had from such a movement, then dynamic multi-granular concurrency control is applicable. Speculation about the expected traffic patterns for individual applications is beyond the scope of this thesis. Even if dynamic multi-granular concurrency control is not cost effective, the application developer can still use static multi-granular concurrency control to choose a fixed granularity of concurrency control that exhibits the best throughput for the most common or average MPL (See section 6.2.2).

So far, only a reactive use of throughput curves has been suggested. An application can monitor its MPL and at any cross-over points can perform the appropriate concurrency controller movement. In systems where response time is not important, throughput curves could be used in a more pro-active fashion. For example, a batch system could limit its MPL to the value that produces the highest throughput. This limiting of MPL could be performed by the batch submission sub-system. In terms of the banking application, such batch operations could be for example, the audit of a branch's daily business.

6.2.1.11 Including More Operations

In terms of distributed computing there is really nothing special about the deposit method test. It is meant to illustrate a typical component of a distributed application. Naturally, a distributed bank application would have many more operations than just deposits. Accounts have to be opened, closed, moneys transferred, interest calculated and so on. The testing that applied to deposits applies to the completed distributed bank or indeed, any distributed application. Throughput versus MPL graphs would be produced for an application as a whole. Multiple methods would be included at their expected frequency of invocation. These graphs, which may or may not include cross-over points can be used to choose either static or dynamic mutli-granular concurrency control for a particular application.

6.2.1.12 The Costs of Multi-Granular Concurrency Control

Multi-granular concurrency control has by it nature introduced some added costs into the processing of Hermes/ST threads. These costs are small and do not detract very much from performance.

6.2.1.12.1 Object Creation Nested encapsulation necessitates some extra object store activity during the creation of a Hermes/ST object. The nested encapsulation structure needs to be maintained on disk so that transactional object structures can survive node crashes. This means that during the creation of a Hermes/ST object the parent
encapsulating object needs to update its record of what objects it currently encapsulates. Furthermore, the extra flags that are needed to restore an object in the event of a node crash, such as an object's concurrency control status, its position in relation to highest and lowest markers ... also need to be recorded during object creation. Because disk drives can only write complete blocks of data to a disk at time and the amount of extra information that needs to be stored is certainly less than a block, the extra information may fit into the remainder of the object size divided by the block size. Therefore the extra overhead is at most one block and sometimes no blocks. This extra object store activity is a fixed one time cost for the life of a Hermes/ST object. Given the assumption that objects are utilised many more times than they are created, this cost can be considered negligible.

6.2.1.12.2 Object Invocation Although Hermes/ST has a high object invocation costs the checking that message invocations satisfy the dynamic (refer section 5.3.6) or static (refer section 5.3.6) navigation protocols is a negligible component of the method invocation process. Without rerouting, both protocols have comparable object access times. For example, to read the accountsTree variable of a instantiated HermesSTBranch takes about 23 milliseconds. However, checking of the navigation protocols conditions (access rules etc.) takes about 0.10 milliseconds. This represents less than half a percent of the total cost of a method invocation.

Any necessary rerouting of message invocations in both the static and dynamic navigation protocols does of course increase the object invocation times. Each reroute is about as expensive as an object invocation. The static navigation protocol reroutes at the sender object while the dynamic navigation protocol reroutes at the receiver objects. Thus, rerouting in the dynamic navigation protocol involves one more object invocation than in the static navigation protocol. The need to reroute method invocations should be quite low. Instances of the sneaking thread problem (refer section 5.2.10) and access rule violations (refer section 5.2.15) should be rare. There are none in the distributed bank. Object rerouting can only happen during the upward movement (refer section 5.3.5.2). Concurrency controller movements are expected to be fairly infrequent. Rerouting of messages is, however, unavoidable if noCC schedulability is to be maintained.

6.2.2 Incremental Development

Because the level of concurrency control granularity is independent of class description, the same object structure can be instantiated with many different levels of concurrency control granularity. Thus Hermes/ST allows an incremental development strategy. An application can be first written and tested for various MPL levels using various numbers of concurrency controllers. Only two levels (minimally locked or guarded) were shown in the deposit example. If there are any useful cross-over points, these can be used to derive triggers that can be deployed over a dynamic multi-granular concurrency controlled application. If there are no useful cross-over points with respect to the expected traffic then a fixed level of concurrency control can be selected to best suit the application. This fixed level of concurrency control can be graduated to fine levels without having to re-write the application. The application developer need only respecify highest and lowest markers (See section 6.2.4).

7To aid in performance, the containment structure of top-level objects is not explicitly recorded in the Hermes/ST system object.
6.2.3 Reuse

Another result of making the granularity of concurrency control independent of a class description is that the same class can be reused by other applications with a different granularity of concurrency control. For example, the \texttt{HermesSTBinaryTree} class that is used to implement the binary accounts tree of the distributed bank application can be reused by other applications. Any Hermes/ST objects that are capable of being compared to other objects can be added to, searched for, etc. in a HermesSTBinaryTree binary tree. Furthermore, an application that wishes to reuse the HermesSTBinaryTree class can redefine the level of granularity of concurrency control of its own instantiation of HermesSTBinaryTree. This is easily achieved via highest and lowest markers. (See section 6.2.4)

6.2.4 Ease of Specification

The creation of branches with various types of account trees is easily specified in Hermes/ST. The \texttt{HermesSTBranch} class contains the following method:

\begin{verbatim}
accountTreeHigh: high low: low implicit: implicit explicit: explicit

self accountsTree: (hermesSelf
    containNew;
    highestCC: high;
    lowestCC: low;
    implicitCC: implicit;
    explicitCC: explicit;
    upTrigger: BTUpTrigger;
    downTrigger: BTDownTrigger;
    args: (Array with: nil);
    hermesClass: HermesSTBinaryTree)
\end{verbatim}

\texttt{HermesSTBranch accountTreeHigh:low:implicit:explicit:} can be called in various ways to instantiate various types of binary account trees. Assuming there is a Hermes/ST pointer to a branch called \texttt{branch} then to install a dynamically locked version of the accounts binary tree the following code fragment can be used:

\begin{verbatim}
\end{verbatim}

To install a minimally locked version of the accounts binary tree the following code fragment can be used:

\begin{verbatim}
\end{verbatim}

To install a guarded version of the accounts binary tree the following code fragment can be used:

\begin{verbatim}
branch accountTreeHigh: true low: true implicit: true explicit: false.
\end{verbatim}

To install a programmably locked version of the accounts binary tree the following code fragment can be used:
CHAPTER 6. DISCUSSION

branch accountTreeHigh: false low: false implicit: false explicit: true.

To install a dual locked version of the accounts binary tree the following code fragment can be used:

branch accountTreeHigh: false low: false implicit: true explicit: true.

As can be seen from the above code definitions the selection of the appropriate version of accounts binary is ultimately a matter of setting four instance creation parameters. These are highestCC:, lowestCC:, implicitCC: and explicitCC:. Because the values of the flags are “cloned” (refer section 3.1.4.1) from parent to child in a sensible manner, the specification of many versions of a data type can be performed in a concise manner. For example, the specification of a guarded account tree is made by instantiating the root node as being highestCC: true lowestCC: true implicitCC: true explicitCC: false. This specification is enough to inform the Hermes/ST system that the root node should be a guard object and all descendant nodes should have no concurrency controllers.

The observant reader will have noticed that the definition of HermesSTBranch accountTreeHigh:low: implicit: explicit: includes instance creation parameters for triggers in all cases. This is not a problem because Hermes/ST only invokes the BTUpTrigger trigger: when dynamic multi-granular concurrency control is in effect. I.e. when the object is between the highest (highestCC:) and lowest (lowestCC:) markers. Therefore, these triggers are ignored when they are not needed.

Another feature of the trigger classes BTUpTrigger and BTDownTrigger is that they were constructed so that they could be turned “on” and “off” by the application. For this application, triggers are made in an “off” state, which means that they will not request lock movement. Assuming there is a variable branch, which contains a Hermes/ST pointer to a branch, then triggers can be turned on and off by the application with the following two lines of code respectively:

(branch accountsTree) meta; triggersOn.
(branch accountsTree) meta; triggersOff.

Such a specification of triggers allows a further simplification in the instantiation of a binary account tree for testing purposes. Both static and dynamic multi-granular concurrency control can be tested on one dynamically locked account tree. To test a guarded account tree the dynamically locked account tree is converted to a guarded account tree and the triggers are turned off. To test a minimally locked account tree dynamically locked account tree is converted to a minimally locked account tree and the triggers are again turned off. To test a dynamically locked account tree triggers are turned on. Furthermore, for programmer convenience lock movement can be initiated without the use of triggers. Assuming a Hermes/ST pointer to a branch called branch, a downward lock movement, which produces a guarded account tree from a minimally locked tree is performed by:

branch accountsTree meta; moveLocksDown: #allLevels.

An upward lock movement, which produces a minimally locked account tree from a guarded account tree is performed by:
6.2.4.1 Mixing Explicit with Dynamic Multi-Granular Concurrency Control

Even though dynamic multi-granular concurrency control must be implicit (refer section 5.3.1.1) it is still possible to mix implicit and explicit concurrency control in the same encapsulation hierarchy. The following code demonstrates this:

```Smalltalk
installEg

An example of how implicit dynamic multi—granular concurrency control can co—exist with explicit concurrency control.
```

```Smalltalk
| branch |
branch := HermesSystem
    containNew;
persistent;
alias: #branch1;
location: #groucho;
hermesClass: HermesSTBranch.
branch
    accountTreeHigh: true
low: false
implicit: false
explicit: false.
branch printSpoolerExplicit: true implicit: false

The above code installs both a dynamically locked binary account tree and a dual locked cheque spooler in the same branch. The code for the creation of the cheque spooler is as follows:

```Smalltalk
printSpoolerExplicit: explicit implicit: implicit

self printSpooler: (hermesSelf
    containNew;
size: 10;
implicitCC: implicit;
explicitCC: explicit;
hermesClass: HermesSTProgBB)
```

6.3 Comparison with Other Systems

This thesis has introduced several ideas. Because no one system embodies all these ideas this section is indexed by Hermes/ST concepts rather than being indexed by individual other systems. Multi-granular concurrency control is the main contribution of this thesis. Multi-granular concurrency control is intended as a tool that allows the application developer to specify reliable distributed object-oriented applications. The author knows of
no direct analogue to dynamic multi-granularity concurrency control. Intention locking is most similar and its relationship to dynamic multi-granularity concurrency control is discussed in section 6.3.1. Section 6.3.2 compares static multi-granular concurrency control with comparable systems. Minimal locking is compared in section 6.3.3. Programmable locking is compared in section 6.3.4. Dual locking is a combination of minimal and programmable locking and therefore the comparisons of minimal and programmable locking apply to it. Dual locking does not have a section. Section 6.3.5 evaluates the Hermes/ST approach to concurrency control against two commercial relational database systems. The concurrency control solutions of two commercial object-oriented databases are compared to Hermes/ST in section 6.3.6. Finally, nested encapsulation is compared in section 6.3.7.

### 6.3.1 Dynamic Multi-Granular Concurrency Control

Dynamic multi-granular concurrency control allows applications to vary their level of concurrency control granularity during the lifetime of an encapsulation hierarchy. Section 6.2.1 showed that the level of concurrency control granularity and hence the number of locks used can affect system performance. Another approach to affecting the throughput of a system by varying the number of locks is called intention locking.

#### 6.3.1.1 Intention Locking

Intention locking [BHG87, GR93] was originally introduced for databases systems in order to allow transactions (non-nested) to individually determine the number of locks that they need to acquire in order to perform a task. This avoided the need to have a uniform "optimal" locking granularity for all data items in a database. Each transaction could lock database entities at the granularity that was appropriate to it. Intention locking has been applied to distributed programming environments supporting nested transactions, such as Encina's Trans-C [Lib92, Cor94] and an un-named system in [HR93].

Intention locking required the introduction of a logical lock type graph [BHG87]. The lock type graph is used to describe the hierarchical structure over which intention locks will be acquired. For example, the logical lock type graph for a distributed bank could be $\text{Bank} \rightarrow \text{Branch} \rightarrow \text{Account}$ [BHG87].

In addition, new locking modes called intention write, intention read and shared intention write were introduced and the lock compatibility matrix [BHG87] was extended to include them. Intention locks are less restrictive than their "real" counterparts. Thus, for example, two intention write locks do not conflict and can be held on the same resource by different transactions. Transactions use intention locks as a means of informing other transactions that they intend to obtain a real lock at some finer level in the lock type graph. For example, a transaction that has acquired an intention write lock on the $\text{Bank}$ lock type entity and an intention write lock on one $\text{Branch}$ lock type entity is signaling its intention to write to one or more accounts.

Finally, an intention lock acquisition and release protocol [BHG87] was introduced to ensure that locks are acquired and released in such a manner that serialisability of transactions is ensured. This protocol ensures two conditions

1. Before acquiring a read or write lock on a lock type entity ($\text{Bank}$, $\text{Branch}$ or $\text{Account}$) a transaction must first set the appropriate intention locks on all ancestors of the lock type entity. For example, before an account is changed a deposit method should have acquired intention write locks on the $\text{Bank}$ and $\text{Branch}$ in question.
CHAPTER 6. DISCUSSION

2. When a "real" lock is acquired on a particular lock type entity then it is not necessary for a thread to acquire locks on any descendant lock type entity. For example, a method that added interest to all accounts in a branch may wish to lock the whole branch rather than individual accounts. This is achieved by a thread acquiring a "real" write lock on the Branch lock type entity.

As an example of what intention locking is trying to achieve, consider depositing to and calculating interest on, all accounts in a branch. Individual deposit methods might like to lock individual accounts. Such a fine granularity of locks allows deposits to be performed in parallel. With multiple branches or idle time in transactions, parallelism will tend to increase system throughput. The interest calculation might like to lock a whole branch. Assuming that the interest calculation is entirely CPU bound then such a coarse granularity of locks would decrease the locking overhead of the interest calculation and make it execute faster. Intention locking allows both operations to maintain their own idea of the "optimal" locking granularity for the distributed bank and still maintain serialisable transactions.

The mechanics of intention locking for the above example are as follows. If a deposit is changing a balance, then it will have intention write locked the Bank, one Branch and be holding a real write lock on some account. Other deposits to separate accounts can execute in parallel because intention write locks on the same Bank and Branch do not conflict with each other. If now an interest calculation attempts to run at a branch, then it will be delayed until all active deposits are completed. This is because the interest calculation's attempt to acquire a real write lock on that Branch will conflict with any deposit intention write locks on that Branch. With strict two phase locking the conflict will cause the acquisition of the interest calculation's Branch write lock to be delayed until all deposits have committed or aborted.

6.3.1.1.1 Principal Difference The principal difference between intention locking and (dynamic) multi-granular concurrency control lies in the definition of the logical lock type graph. Intention locking uses a static logical lock type graph that can be independent of the physical representation of the data. For example, Bank – Branch – Account is independent of whether the accounts are stored in a binary tree or a Btree. Hermes/ST multi-granular concurrency control uses the physical encapsulation structure to represent the lock type graph. The important point to be made here is that concurrency control movements modify the nature of this physical lock type graph. Thus Hermes/ST uses a dynamic physical lock type graph. For example, the conversion of a guarded account tree to a minimally locked account tree can be seen as the conversion of the branch and accountTree physical lock type graph into the branch, accountTree followed by node account "for as many levels of the tree as there are" physical lock type graph.

Both intention locking and Hermes/ST's multi-granular concurrency control have a similar goal of improving performance by varying the number of locks (or concurrency controllers) used by an application. However, they go about this process in a different manner. Intention locking allows individual methods to utilise the static logical lock type graph in order to maintain their own idea of optimal locking granularity. This approach can be seen as a method centered approach. For example, as introduced above, a deposit method can have a concurrency control granularity that is different to that of an interest calculation method.

Hermes/ST's multi-granular concurrency control forces each method to adhere to the currently active physical lock type graph. However, the concurrency control movements

---

8This "graph" is actually a "tree".
allow methods to change the currently active physical lock type graph. All methods share
the common physical lock type graph and hence a common object locking granularity.
This approach can be seen as a data centered approach. For example, a deposit method
acquires locks to whatever granularity of concurrency control an account tree is currently
using. Many locks in the minimally locked account case and one lock in the guarded
account tree case. Furthermore, a deposit method can either directly change the physical
lock type graph by initiating concurrency controller movement or can indirectly effect the
physical lock type graph via triggers.

Intention locking has been the standard approach to the locking granularity issue in
databases for a long time [GR93]. Both intention locking and multi-granular locking have
appeal. The choice of whether a data centered or a method centered approach should be
used will depend on the application at hand.

6.3.1.1.2 Incremental Development Differences Both intention locking and Her­
mes/ST's multi-granular concurrency control allow an application developer to incremen­
tally determine an appropriate level of lock (or concurrency controller) granularity. The
application developer can experiment with different locking granularities for individual
methods and see their effect on overall system performance in order to determine the
"best" set of locking granularities for a particular application.

When intention locking is being used, different concurrency control granularities can
be employed by different methods on the same object structure. For example, a deposit
method can have a concurrency control granularity that is different to that of an interest
calculation method. Various interpretations of the static logical lock type graph can be
written and the "best" picked.

Hermes/ST's multi-granular concurrency control allows the same approach. The ap­
plication developer can experiment with different concurrency control granularities for
the common object structure to determine the best concurrency control granularity for
a particular application. If static multi-granular concurrency control is used then all
methods must decide on a common physical lock type graph. If dynamic multi-granular
concurrency control is used then individual methods can change the physical lock type
graph, or triggers that work for the "common good" can be derived. Section 6.2.1.9 gave
an example of how this might be carried out.

6.3.1.1.3 Reuse Differences When intention locking is used in distributed program­
ing environment, the need for a hierarchical ordering of an application's data via a lock
type graph deters reuse. Each method of a class must explicitly acquire locks so as adhere
to the lock type graph. It is not clear how the application developer can overwrite this.
For example, when adding interest to accounts it might be desirable to reuse the deposit
method but change its locking granularity from that of an Account to that of a Branch.
Because the lock acquisition code is typically a part of the deposit method then short of
writing another deposit method at a different granularity this is hard to do.

Conversely, nested encapsulation provides Hermes/ST with an implicit physical lock
type graph (see section 6.3.1.1.1). Hermes/ST multi-granular concurrency control localises
knowledge about this hierarchy to a few instance creation parameters. The highest and
lowest markers and triggers. The reuse of a class with different concurrency control
granularities is merely a matter of reinstating the class with different highest and
lowest markers and if dynamic multi-granular concurrency control is used, some new
triggers. For example, if the method that adds interest to accounts needs to operate
at a different level of granularity it can do so by triggering either downward or upward
concurrency controller movement. This movement is independent of whether this method needs to reuse the deposit method or not.

6.3.1.4 Specificational Differences Intention locking imposes a specificational burden on the application developer. The need to maintain serialisability of transactions by adherence to the lock acquisition and release protocol for an applications lock type graph is extra specificational work for the programmer. In the above example the intention locked deposit method was forced to explicitly acquire intention locks for the Bank and the Branch lock type entities. This was in addition to the acquisition of the write lock needed for the Account lock type entity.

Conversely, Hermes/ST multi-granular concurrency control requires no such specification from the application developer. The encapsulated hierarchy navigation protocol that is defined by Hermes/ST ensures that methods acquire and release locks in a manner that ensures serialisability. This navigation protocol is independent of the granularity of concurrency control that an application is using. For example, a programmably locked deposit method for a multi-granular concurrency controlled branch need only concern itself with the acquisition of AccountWriteLocks. As an even better example, a minimally locked deposit method need not concern itself with the acquisition of any locks. The Hermes/ST system uses the read and write accesses to data to acquire the appropriate locks on behalf of the application.

6.3.1.2 Lock Escalation and De-Escalation

When intention locking is used, individual methods are free to interpret the static logical lock type graph in their own manner. However once interpreted the chosen granularity of locking must be adhered to under all execution conditions. It is not possible for a method to change its locking granularity according to execution time factors. This static nature of intention locking is especially unattractive in a distributed system where applications can have diverse access patterns that may not be known at the time of specification. For example, a distributed bank may open more and more branches as time passes.

Lock escalation [BHG87, GR93] and lock de-escalation [GR93] are attempts at addressing the static nature of intention locking. When lock escalation is used a transaction acquires locks according to its static intention locking specification until a lock escalation threshold [BHG87, GR93] is reached. At this point the transaction considers the specification to be inadequate (e.g. too many locks acquired) and tries to reduce the number of locks it possesses through lock conversion [BHG87, GR93]. For example, an interest calculating method that uses intention locking could acquire intention write locks on the Bank and a Branch and "real" write locks on any Accounts for which interest has been calculated. If the branch has many accounts and the traffic is low then the method may decide to convert all the real Account locks into one Branch lock and proceed. Lock de-escalation is similar but de-escalation requires a mechanism to keep track of the locks that an operation would have needed if it were ever to de-escalate. For example, an interest calculation that acquires locks at the level of the Branch would need to record what accounts it has added to if it decided to convert its Branch lock to many Account locks.

6.3.1.2.1 Lock (De-)Escalation versus Concurrency Controller Movement

Lock escalation and de-escalation do not have the same objectives as the dynamic multi-granular concurrency control movements. Lock escalation and de-escalation allow active transactions to change the number of locks that they acquire over a fixed logical lock
type graph. Conversely, concurrency controller movements result in changes to an application's current physical lock type graph. For example, downward concurrency controller movement is delayed until there are no threads active in a downward moving hierarchy. The reasons for this are given in section 5.3.4. Thus downward concurrency controller is not comparable to lock de-escalation.

Because upward concurrency controller movement can be performed in the presence of one thread it can achieve the same result and hence be compared to lock escalation. Lock escalation allows a transaction (that presumably is unhappy with a logical lock type graph) to reduce the number of locks it has acquired and will acquire by converting many fine granularity locks into one coarser granularity lock. Upward concurrency controller movement allows the same sort of behaviour. A trigger can be programmed to count the number of locks that a Hermes/ST thread has acquired. This same trigger can be programmed to initiate an upward concurrency controller movement at the appropriate object if the Hermes/ST thread acquires more than a “lock escalation threshold's” worth of locks.

Both lock escalation and upward concurrency controller movement may be delayed. For example, in the lock escalation case, an interest calculation method that was attempting to convert many Account write locks to one Branch write lock would need to wait until any other transactions that hold locks on the Branch have completed. In the upward concurrency controller movement case the movement is delayed until there is only one active thread in the encapsulated hierarchy.

A final point is that upward concurrency controller movement is more expensive than lock escalation. This is because all objects below the triggering object in the encapsulation hierarchy have to be communicated with during concurrency controller movement (see sections 5.3.5 and 5.3.4). Lock escalation, on the other hand, need only convert as many locks as the method has already acquired. Typically, there will be more objects in an encapsulation hierarchy than there are locks that need to be converted. For example, to convert a guarded account tree into a minimally locked account tree, all accounts and nodes in the binary tree need to be converted into minimally locked objects. A calculate interest method only needs to convert the locks of as many accounts as it has visited so far. To put this point in context, it is worthwhile remembering that upward concurrency controller movement was not designed to implement lock escalation. Such behaviour is merely a side effect of its ability to be able to perform this movement while there is one thread active in an encapsulation hierarchy.

### 6.3.1.3 Co-Existence

Hermes/ST’s multi-granular concurrency control does not preclude intention locking. Intention locking and multi-granular concurrency control could co-exist. When mostly one consistent object centered model of concurrency control granularity is required for an application then multi-granular concurrency control is easy to specify, develop, reuse and modify during application execution. When more than one model of concurrency control granularity is to be maintained over one object topology at any one time, then intention locking may be better suited.

Hermes/ST is well suited to an implementation of intention locking. In Hermes/ST, a logical lock type graph is defined automatically by the nested encapsulation hierarchy. The encapsulation hierarchy navigation protocol already achieves some of the required conditions of the lock acquisition and release protocol of section 6.3.1.1. The acquisition of locks in root to leaf order is guaranteed if the access rules are not violated. Care would have to be taken that asynchronous threads would have to abide by the lock acquisition
and release protocol. The rerouting mechanism would be useful here.

6.3.2 Static Multi-Granular Concurrency Control

Most distributed programming environments support some form of static concurrency control granularity but do not have the flexibility of Hermes/ST's static multi-granular concurrency control. Among these are Argus [Lis88], Avalon [EME91], Arjuna [Shr92] and Clouds [GRJL92]. The specification of concurrency control in systems like Argus, Avalon and Arjuna is tied to the implementation structure. Thus in all three the implementation of a binary tree with one lock is different to the implementation of a binary tree with many locks. Static multi-granular concurrency control in Hermes/ST allows the granularity of concurrency control to be specified independently of the object structure for the intended abstract data type. This is achieved by having highest and lowest markers that attach the concurrency control granularity to an object structure during the creation of objects in that structure.

6.3.2.1 Distributed Eiffel

Distributed Eiffel [GRJL92] supports large and fine grained concurrency control for objects. It also allows applications to be redeployed with different concurrency control granularities. This is achieved by specifying whether a class is to be instantiated as a fine grained object or as a coarse grained object. This is similar to static multi-granular concurrency control with the main difference being that Distributed Eiffel only allows two granularities of object instantiation. Static multi-granular concurrency control allows as many levels of granularity as the encapsulation structure is deep.

6.3.3 Minimal Locking

Argus, Avalon/C++ and Arjuna do not provide minimal locking. In all of these systems, the acquisition of a lock is an explicit part of the operation definition. In Argus an atomic type is accessed via an explicit call to read_lock(atomic_object) or write_lock(atomic_object) [Lis88]. In the Avalon/C++ system, operations that are subclassed from the atomic class acquire read or write locks for the operation through read_lock() and write_lock() methods of the atomic class [DHW88]. Locks so acquired are easily thought of as pertaining to the method. Thus a read only method should acquire a read_lock(), and a method that changes object state should acquire a write_lock(). In Arjuna, classes derived from LockCC acquire locks through calls to its setlock() method. Thus setlock(new Lock(READ)) acquires a read lock for an operation while setlock(new Lock(WRITE)) acquires a write lock [PS88].

Minimal locking is attractive for the following reasons:

- The first and most obvious is that minimal locking relieves the programmer of the burden of specifying concurrency control. For many data types, minimal locking provides adequate concurrency control for "free".
- Minimally locked data types are always correctly concurrency controlled. The possibility of concurrency control specification errors is eliminated. Such errors can be hard to identify. Some examples include: declaring an operation to be a reader instead of a writer; forgetting to declare a method as a reader or a writer; over-specifying a method because the lock granularity is inappropriate.

8LockCC is renamed LockManager in [Shr92].
• Minimally locked data types avoid unnecessary data contention. A minimally locked data type allows as much concurrency control as an object structure permits.

Minimal locking, although attractive, is deficient in the following ways.

• Some abstract data types have synchronisation constraints that are not expressed by minimal locking. For example, a “get” operation on a bounded buffer (refer section 4.2.6) has to be delayed until after a “put” operation has been performed. Such behaviour is not expressed by minimal locking.

• Minimal locking may introduce deadlock and starvation problems.

• Minimal locking is expensive.

These problems have been addressed by Hermes/ST multi-granular concurrency control.

6.3.4 Programmable Locking

Argus, Avalon/C++ and Arjuna all support user-defined concurrency control. Type-specific concurrency control in Argus [WL85] has a different goal to its counterpart in Hermes/ST. The goal of Argus’ user-defined atomic types is to permit higher concurrency than strict two phase locking allows. One goal of Hermes/ST’s user-defined programmable locking is to further restrict concurrency allowed by minimal locking in order to avoid problems such as deadlock and starvation. Therefore, the mechanisms are not further compared.

Similar arguments apply to Avalon/C++. However, some aspects of Avalon/C++’s approach to user-defined locking [DHW88] do compare with the Hermes/ST approach. The idea that locks are specified via inheritance is shared. Avalon/C++ provides the subatomic class as a starting point for defining a user-defined hierarchy of locks. This use of inheritance is analogous to Hermes/ST programmable lock inheritance. However, since method declarations contain concurrency control information in Avalon/C++, they cannot be easily reused.

User-defined locking in Arjuna [PS88] is similar to the Hermes/ST programmable lock approach. The lock concurrency controller class LockCC exports operations setlock and releaselock. releaselock is called implicitly at transaction termination time. An application calls setlock which then calls lockconflict which in turn calls the != operator. The != operator is analogous to the isCompatible: method in the Hermes/ST programmable lock approach. It can be overridden in user-defined locks.

Arjuna, like Hermes/ST, permits object state to be passed to locks during the instance creation of a lock. However, it does not support inspection of object state through guard methods (see section 4.2.4.2). Thus, it is not clear how an operation such as a bounded buffer “get” of section 4.2.6 can be specified.

Consistent with programmable locks in Hermes/ST, locks in Arjuna are organised in an inheritance hierarchy and are specified independently of their use. Thus Arjuna’s user-defined lock specifications can be re-used and can be extended via inheritance. Locks, however, are not associated with a method as they are in Hermes/ST, but instead are a part of the Arjuna method definition. Thus, concurrency specifications are not easily reused. Therefore, Arjuna lacks some of the reuse advantages that Hermes/ST provides.
6.3.5 Commercial Relational Database Systems

This section evaluates multi-granular concurrency control against Oracle7 [Sta94, OA94] and Sybase System 10 [SR90], two commercially available relational databases. Relational database systems have data models based on relations [Cat91, GR93]. A relation is a logical data model and is best thought of as a table. The rows of a table are called tuples [Cat91, GR93] and the columns of a table are called attributes [Cat91, GR93]. This logical data model is then mapped onto a physical data model. The mapping of the logical relations to physical secondary storage is performed through disk pages. Fast access to these pages is achieved through a variety of mechanisms. These include: Hashing, B-Trees and parent-child links. Sybase and Oracle both support B-Tree indexing. Relational databases also support uniform creation and access of data via some form of query processing language. SQL [Cat91, GR93] is the standard relational query processing language and is supported by both Sybase and Oracle. Hermes/ST is a distributed object system supporting nested transactions (refer chapter 3) rather than a relational database. Hence Hermes/ST does not support the relational data model or ad hoc queries via SQL. Conversely, relational databases (including Sybase and Oracle) do not support nested transactions [Cat91, GR93].

Concurrency control in both Sybase and Oracle is pessimistic and based on locking (refer section 2.3.1.2). Both systems support strict two-phase locking (refer section 2.3.1.3). Hermes/ST also supports strict two-phase locking but includes Humm's [Hum94] extensions for nested transactions. Relational database systems (including Sybase and Oracle) that support SQL usually implicitly lock data during the execution of a query processing language (SQL) statement. Hermes/ST performs both implicit (minimal locking) and explicit locking (programmable locking) (refer section 5). Sybase further allows user-defined transactions through the addition of BEGIN, COMMIT, ROLLBACK and SAVE TRANSACTION SQL statements [SR90]. Hermes/ST allows the application developer to define (nested) transactions through the object invocation process (refer section 3.1.5).

Both Oracle and Sybase have network extension packages that allow distributed transactions. The concurrency control for distributed transactions is still based on strict two-phase locking, but recovery for these transactions is now based on a distributed two-phase commit algorithm (refer section 2.3.2.2.2). Hermes/ST also uses the distributed two-phase commit algorithm. Sybase has introduced restrictive shared locks [SR90]. These locks allow the system control over deadlock (refer section 2.3.1.3.1) via a dataserver [SR90]. The Hermes/ST approach to deadlock is via the programmable lock approach (refer section 4.2). Currently Hermes/ST does not provide system level deadlock prevention. Deadlock in Hermes/ST is handled via a simple timeout mechanism (refer section 3.1.7). Both Oracle and Sybase also support replication. Sybase system 10 has a replication server and Oracle has a log transaction manager [Sta94]. Triggers [OA94] in Oracle are not to be confused with triggers of dynamic multi-granular concurrency control (refer section 5.3.3). Sybase triggers are used to maintain referential integrity whereas Hermes/ST triggers are used to initiate lock movement algorithms.

The granularity of locking in relational databases can be based on the physical or logical data model. Both Sybase and Oracle support implicit page level locking through SQL. Hermes/ST does not support physical page locking. The reasons for this were detailed in sections 4.4.3 and 4.4.2. To recap, the granularity of state restoration and persistence handling must be at least as fine as the granularity of concurrency control. Minimal locking (refer section 4.4) requires individual variables to be separately locked. Oracle and Sybase also support various levels of granularity at the logical level. Tables, rows and records (Sybase only) can be locked. Sybase and Oracle do not support intention
locking, but relational databases such as Ingres and Informix do [SR90]. Section 6.3.1 discusses intention locking, lock escalation and their relation to Hermes/ST in detail.

6.3.6 Commercial Object-Oriented Databases

This section evaluates multi-granular concurrency control against Versant [HPbC93, Cat91, GH91, Hug91] and Objectivity/DB [Cat91], two commercially available object-oriented databases. Object-oriented database systems were introduced to overcome perceived deficiencies with relational databases. These deficiencies include a lack of multimedia management, a lack of temporal data (versioning), a lack of complex data, a lack of procedural data (programs) and a lack of support for long duration transactions. Object-oriented database systems have data models based on objects [Cat91, GR93]. As is the case in Hermes/ST an object in an object-oriented database encapsulates data (state) and exports methods (that define behaviour) (refer section 3.1). Objects are permitted to contain references to other object. Such objects are referred to as composite objects.

As was the case with relational databases, this logical data model is mapped to a physical data model. The mapping of the logical relations to physical secondary storage is also performed through disk pages. Object-oriented databases may or may not provide fast access to objects via some form of object server [Cat91, Hug91, HPbC93]. For example, Versant lets users cluster frequently accessed object as an object group [HPbC93]. Objectivity/DB supports composite objects but Versant does not. Hermes/ST also supports composite objects called encapsulated objects (refer chapter 3). One key difference is that encapsulated objects in Hermes/ST enforce information hiding whereas composite objects in an object-oriented database do not.

In attempting to resolve the deficiencies (e.g. long transactions) of relational database systems, most object-oriented databases have tried to extend the semantics of transactions. There have been many approaches to long transactions in object-oriented databases [BK91]. These include versioning, soft locks, dirty reads, queing options and lock events [Cat91, Hug91, HPbC93]. Versioning is used in both Objectivity/DB and Versant. Versioning allows applications to create and manipulate different copies of an object. This is useful in say, a CAD/CAM environment, where several designers need to work on the same copy of a component at once. Traditional transactional read/write locking rules such as those of Hermes/ST (see section 3.1.5.1) do not allow this to happen. Soft locks allow transactions to inform other transactions when the locks they wish to acquire are already held by other transactions. This allows transactions to take actions other than simply delaying when there is a lock conflict (refer section 3.2.1.1.2). Hermes/ST does not support soft locks. Dirty reads allow transactions to see the uncommitted state of another transaction. Hermes/ST does not allow dirty reads. The above mentioned mechanisms all give rise to non-serialisable schedules (refer section 2.3.1). This is contrary to the traditional semantics of transactional based systems [GR93]. Such mechanisms represent interesting research, but are outside the scope of Hermes/ST and this thesis. The multi-granular concurrency control as applied to nested transactions in this thesis allows less concurrency than the above mentioned approaches in some situations, but multi-granular concurrency control always maintains serialised schedules. This is the key difference between these exotic [GR93] concurrency control mechanisms and multi-granular concurrency control.

Object-oriented databases can also perform more traditional forms of distributed concurrency control. Systems supporting just optimistic concurrency control such as GemStone [B0S91] are not comparable with Hermes/ST's pessimistic concurrency control. Systems such as ORION [Hug91] support intention locking. Intention locking was com-
pared in section 6.3.1.1. Versant and Objectivity/DB\(^\text{10}\) perform traditional concurrency control either at the level of pages, or at the level of objects (perhaps including composite objects). Other object-oriented databases such as ObjectStore [LLOW91] or at the level of classes [Cat91].

Page, object or composite object locking can all be seen as static forms of concurrency control. They do not allow applications to vary the level of concurrency control that they employ to suit changing object invocation patterns. Sections 6.2.1.9 and 6.2.1.10 demonstrated the utility of being able to vary the level of concurrency control (dynamic multi-granular concurrency control) according to object invocation patterns. Hermes/ST's static multi-granular concurrency control does not support page level locking, but does support the locking of groups of (encapsulated) objects. In Hermes/ST, a variable, rather than a page or object, is the smallest lockable unit. This allows an application to determine the level of concurrency control granularity that is best suited to it across a broader range of granularities. When static multi-granular concurrency control is used, data items from single variables to entire hierarchies can be locked by a single concurrency controller. For example, with respect to the distributed bank deposit example, locking a page worth of nodes for each deposit could unnecessarily restrict parallelism in a high contention environment.

### 6.3.7 Nested Encapsulation

Most object-oriented systems support at least encapsulation of data by an object. Languages such as Smalltalk [GR89], C++ [Str86] and Eiffel [Mey88] only allow the instance variables of an object to be accessed via well defined interfaces. For example, in Smalltalk, if a class owns an instance variable and access to that variable is required from outside the class, then a method must be supplied to achieve this.

Some object oriented languages such as C++ extend this concept further, to define a scope of visibility for methods. For example, the private protected and public keywords of C++ allow the applications developer to limit the visibility of methods. A private method is only visible to methods of the class that owns the private method. A public method is visible to all classes. A protected method alters its visibility depending on the where it is declared in the class hierarchy. Other languages impose a flat structure on method visibility. For example, in Smalltalk all methods of all objects can be invoked once a reference to an object has been acquired.

Hermes/ST nested encapsulation further extends encapsulation by imposing a hierarchy on the visibility of objects. From a Smalltalk perspective this changes the flat method visibility structure to a hierarchical one. From a software engineering perspective this can be seen as implementing information hiding [Mey88, BL92] in a consistent, system-wide, fashion. From a distributed programming environment perspective, nested encapsulation is used as a tool that allows the Hermes/ST designers to reason about the access paths of threads in an object hierarchy. Recall from chapter 5 that the combination of nested encapsulation, the order rule and the access rule guaranteed serialisability for threads while allowing the Hermes/ST system to vary the number of concurrency controllers.

At least one other distributed programming system has seen the possibilities that visibility hierarchies for objects provide. The Raven system [FAC+94] from the University of British Columbia is also an object-oriented distributed programming environment. The Part - Of reference property for Raven objects allows the Raven system to view objects as single clusters. This information can be used to aide in object persistence handling or

---

\(^{10}\) Objectivity/DB allows the applications developer to choose either pessimistic or optimistic concurrency control.
in object migration. For example, when migrating an object that belongs to a cluster the whole cluster can be migrated. Assuming that migration of whole clusters is more efficient than the migration of the individual parts and further assuming that if an application accesses one part of a cluster it is likely to access further parts of the cluster then cluster migration will be more efficient than demand migration of individual objects.
Chapter 7

Conclusions and Future Research

7.1 Conclusions

Distributed systems are emerging as an important technology in the computer industry. Distributed systems are inherently more complex than their non-distributed counterparts. Amongst several factors that add to the complexity of distributed systems are partial failures and concurrency. Distributed programming environments offer abstractions that aid in the production of distributed applications. Hermes/ST is a research prototype distributed programming environment. Hermes/ST offers the application developer (nested) transactions as an abstraction to deal with the complexities of distributed systems. Hermes/ST has several novel features including those presented in this thesis and those presented in Ranson [Ran95] and Humm [Hum94].

This thesis deals with the Hermes/ST approach to concurrency control for distributed applications. Distributed applications are produced as a combination of transactional, volatile and constant objects. These objects can interact via messages that combine to form threads. Threads can be transactional or non-transactional, and synchronous or asynchronous. Transactional threads are serialised via Humm's extensions to the pessimistic two phase locking for nested transactions. Non-transactional threads are scheduled so as to maintain the serialisability of transactional threads. Non-transactional threads are not two phase. Thus locks are held only for the duration of the message.

One important aspect of concurrency control is the granularity at which it is applied. The granularity of concurrency control affects the throughput, response time, and fairness characteristics of a distributed programming environment. The granularity of concurrency control affects throughput by affecting the level of unnecessary contention, lock management overheads and deadlock. The effect of unnecessary contention can be magnified in a distributed system supporting two phase locking. The reasons for this is that the processor utilisation by threads in distributed applications is typically lower than that of their centralised counterparts. For example, a remote object invocation by one thread, typically leaves the processor free to execute other threads. A deposit operation that authenticates a password via a central authentication service is an example of such a low processor utilising thread.

7.1.1 Hermes/ST Concurrency Control

Hermes/ST has presented several approaches to concurrency control in a distributed programming environment. The Hermes/ST programmable lock approach allows an application developer to explicitly define concurrency control specifications and associate these specifications with methods. Programmable locks allow the application developer to trade
off the number and type of locks that an application will have against the effects of these locks on throughput. The programmable lock approach allows the application developer to control the level of unnecessary contention, deadlock and lock management costs of an application. However, the programmable lock approach suffers from the following deficiencies:

- Programmable lock specifications must be explicitly supplied by the application developer. Such explicit concurrency control specifications are error prone.

- Programmable lock specifications can be hard to produce. For example, in the distributed bank application it is desirable to perform as many operations in parallel as possible. Deposit operations can avoid unnecessary contention while using relatively few locks in a straightforward manner. Individual deposit operations can proceed in parallel if they access different accounts. However, avoiding unnecessary contention is harder when say, a new account needs to be opened. This is because the effect of opening an account is related to the structure of the binary account tree that orders the accounts. In the distributed bank example, it is non-trivial to specify a single isCompatibleWith: method for an account opening lock that allows as much parallelism as the binary account tree can structurally support.

Minimal locking allows applications to exhibit a high level of parallelism without generally\(^1\) needing any explicit concurrency control specifications. High parallelism is achieved via fine grained locking of a uniform reference object model. Although minimal locking avoids unnecessary contention, it does so at a price. Minimal locking's fine grained concurrency control is expensive. Each variable access involves a lock acquisition which is reflected in extended thread execution times.

In order to allow the application developer to keep the advantages of minimal locking (avoiding unnecessary contention and implicit concurrency control specification) while controlling the lock management costs, Hermes/ST introduces multi-granular concurrency control. In Hermes/ST's static multi-granular concurrency control, objects are grouped into hierarchies via nested encapsulation. Guard objects are introduced that perform surrogate concurrency control for a group of objects that do not acquire any locks (noCC objects). Serialisability is guaranteed for threads by the static navigation protocol. Static multi-granular concurrency control allows an application developer to experiment with different levels of concurrency control granularity across a range of object access patterns. This allows the application developer to pick the "best" level of concurrency control granularity for the expected application access patterns. Furthermore, in applications where response times are unimportant, such as in batch applications, the analysis of various static concurrency control granularities permits the application developer to fix the degree of parallelism at a level that provides optimum throughput.

In Hermes/ST's dynamic multi-granular concurrency control, applications can alter the number of concurrency controllers (hence locks) that they use according to execution time factors. Conceptually, the position of the guard objects can be moved in the encapsulation tree. Downward concurrency controller movement introduces more concurrency controllers into an object hierarchy. Upward concurrency controller movement removes concurrency controllers from a hierarchy. Dynamic multi-granular concurrency control allows the application developer to build applications that can react to varying access patterns by adding or removing concurrency controllers. Concurrency control movement can be initiated directly via special meta messages to an object hierarchy, or indirectly via triggers. By experimenting with various levels of system activities at various levels

\(^1\)But see section 4.2.4.2.
of concurrency control granularity, applications can determine suitable conditions to perform concurrency controller movement. These conditions can be monitored by triggers that in turn can activate the appropriate concurrency controller movement. Furthermore, meta messages can be used to initiate concurrency controller movement during periods of unexpected traffic. For example, with respect to a distributed banking application, an audit method may need to be run irregularly and out of business hours. Under such circumstances (no parallel activity required), a temporary upward movement of locks, via a meta message, may be beneficial.

7.1.2 Object Orientation

Because Hermes/ST is an object-oriented as well as a distributed programming environment, attention has been paid to factors such as incremental development, reuse and ease of specification. A common theme that presents itself is one of the decoupling of the association between concurrency control definition and object definition.

Programmable locks decouple the explicit concurrency control specification from methods. This allows incremental development by allowing the application developer to start with simple, well tested, locking modes such as two phase read write locking and, if necessary to extend the lock types to more application specific locking modes such as those of the AccountWriteLock class. Reuse is implied in such a strategy. A method can be reused with a different types of locking and different locks can be reused by separate methods.

Minimal locking and its extension to multi-granular concurrency control provide incremental development, reuse and ease of specification. Again, the decoupling of concurrency control definition from object definition is evident. The same hierarchy can exhibit different concurrency control granularities by simply changing the position of highest and lowest markers in the hierarchy. An incremental development strategy is offered in two ways. Firstly, a fixed cost static hierarchy can be incrementally developed by reinstantiating the highest and lowest markers in a hierarchy. Secondly, a variable cost hierarchy can be developed by adding in trigger classes. As was the case with programmable locking reuse is implicit in such a strategy. A hierarchy (such as a binary tree) can be reused by different applications with different levels of concurrency control granularity, and triggers can be reused by different hierarchies.

7.1.3 Limitations of Multi-Granular Concurrency Control

Section 6.2 contains an evaluation of multi-granular concurrency control. This section enumerates several limitations of the results of this experiment and of this thesis.

1. The deposit experiment is a simple application that is meant to highlight the potential usefulness of both static and dynamic multi-granular concurrency control. As noted in section 2.7, the electronic bank modelled in this thesis is a simplified distributed bank. The development of a real banking application would entail a much more detailed analysis and implementation. For example, in a commercial implementation of a distributed bank, it is unlikely that the storage of accounts at branches would be implemented as a binary tree. A binary tree was chosen for this thesis because it afforded a simple mechanism to highlight the benefits of multi-granular concurrency control.

2. The size of the deposit experiment was deliberately constrained to show the crossover between minimally locked and guarded implementations of the distributed bank's branch data structure. This experiment could have been repeated over a
varying types of accounts data structures. Such a detailed analysis of the effects of dynamic multi-granular concurrency control on a large set of data structures is outside the scope of this thesis.

3. Hermes/ST pins all objects in memory during a transaction. This assumption is becoming more realistic as the size of real and virtual memory is growing on modern computer systems [GR93]. Object-oriented and relational databases do not necessarily make this assumption. Systems that do not pin data in memory during a transaction need to log changes to data through some form of undo-log (refer section 2.3.2.1). Given the limited time to create the Hermes/ST prototype, this simplification was considered acceptable. It is worthwhile noting that other distributed systems such as Argus [Lis88] and Arjuna [Shr92] have also taken the decision to pin all data in memory during a transaction.

4. Relational and object-oriented database researchers often take objection to the relatively simple physical data models that distributed programming environments such as Argus [Lis88], Avalon [EME91], Arjuna [Shr92] and Clouds [GRJL92] and Hermes/ST use. It is important to realise that distributed programming environments are typically investigating research into other aspects of distributed systems. For example, this thesis has concentrated its development effort on the implementation and linguistic support of multi-granular concurrency control. Aspects of transactional systems that were not central to this research were implemented using simple (but still correct) algorithms if necessary, or omitted if unnecessary. For example, Hermes/ST’s recovery is necessary to support transactions but not central to this research. Therefore, it is a simple implementation of the standard two-phase commit algorithm. As another example, performance optimisations such as composite object clustering [Cat91] were not central to the research and not necessary to support transactions. Such performance optimisations have not been implemented in Hermes/ST.

7.2 Future Research

This section is presented on two different levels. Firstly, proposals for the extension of the idea to decouple various components of the Hermes/ST system are proposed. Secondly, extensions to the version of Hermes/ST used in this thesis are proposed.

7.2.1 Extending Decoupling

In section 2.5.4 the idea that an object (or a group of objects) is a suitable unit for many of the features a distributed system was introduced. Concurrency control in Hermes/ST is an implementation of this idea. However, by introducing a decoupling between concurrency control and object hierarchies, Hermes/ST allows the concurrency control granularity to vary with respect to object hierarchies. This orthogonality of concurrency control granularity and object structure can be exploited at an application’s instantiation or during the application’s usage.

This decoupling process could be applied to other distribution features. For example, under some circumstances, it may be beneficial to vary the granularity of persistence handling of objects. Currently, Hermes/ST supports fine grained persistence handling via logging. A fine granularity of persistence handling trades off the amount of information that must be written to the disk against the cost of finding that information on the disk. Applications that rarely change their object structure could benefit by clustering
this information on stable storage. Naturally such a variable granularity of persistence handling would have to comply with the restrictions of section 4.4.3.

As another example, the semantics of object interaction could also benefit from the decoupling process. Currently, Hermes/ST passes constant objects by value, whereas transactional and volatile objects are passed by reference. Therefore, a distributed application that accesses remote objects has to perform an expensive network access each time it dereferences a remote object. By decoupling object interaction semantics from the various kinds of objects in Hermes/ST, performance gains could be made. For example, it could be worthwhile passing a deep copy of a heavily used remote transactional object, whereas a lightly used remote object should be passed in the existing pass-by-reference manner. Importantly, because different objects are used in different ways by different applications, no fixed interaction semantics is optimal in all circumstances.

7.2.2 Extensions to Mechanisms of this Thesis

7.2.2.1 Dynamic Multi-Granular Concurrency Control

One extension to dynamic multi-granular concurrency control is to remove the implicitCC restriction of section 5.3.1.1. Thus, explicitly concurrency controlled hierarchies could also perform concurrency control movement. One reason for this restriction is so that Hermes/ST can reason about the number of threads in a guarded hierarchy. However, if explicit guard objects were allowed, then it would be the responsibility of the application developer to ensure that threads that are active during a concurrency controller movement did not defy noCC serialisability. Another reason for this restriction, is that there is currently no provision for the expression of two lock specifications for an object in hierarchy. Two lock specifications would be needed because one behaviour is required when the object is as-specified, and another separate locking behaviour is required when the object is an explicit guard object.

Another extension to dynamic multi-granular concurrency control is to remove the local object restriction of section 5.3.1.2. As mentioned in section 5.3.1.2 this would entail ensuring that the transition between “before” and “after” states of concurrency control movements remained atomic in the presence of partial failures. As also mentioned in section 5.3.1.2, an earlier version of Hermes/ST implemented this functionality. However, this approach would have to be extended to deal with the access rule defying object invocations introduced in this thesis.

A final possible extension to dynamic multi-granular concurrency would be to make concurrency controller movements persistent. Technically, this would entail ensuring that the after state of concurrency control movements are logged to stable storage. Conceptually, there is some argument about whether concurrency control algorithms should be made persistent. If concurrency controller movements were made persistent then they would survive node crashes. However, each concurrency controller movement would be made more expensive by introducing the need for them to write to the log. Perhaps a suitable middle ground would be to allow the application developer to specify whether or not a concurrency controller movement should be persistent.

7.2.2.2 Nested Encapsulation

The implementation of nested encapsulation in this thesis could be extended. Hermes/ST assumes that nested encapsulation hierarchies are static structures. An object is created in a hierarchy and remains in that hierarchy until it is removed. An object or group of objects can be made to move hierarchies by deleting them from one hierarchy and adding
them to a new hierarchy. There is currently no system support for such movement of objects amongst hierarchies. The Hermes/ST system could implement object migration more efficiently than the current move-by-delete-and-add strategy. At the system level, the migration of object hierarchies can be performed by adding the moving object's Hermes/ST pointer into the destination hierarchy and removing it from the source hierarchy. Furthermore, the Hermes/ST system would have to ensure that objects migrating from one hierarchy to another comply with the order rule. Naturally, Hermes/ST should add syntactic sugar to allow object migration to be expressed. Gow [Gow94] has proposed some primitives for such movement.
Bibliography


BIBLIOGRAPHY


[Gow94] Brendon Gowing. A concurrent composite object model supporting mutation. Obtained via E-mail Correspondence with bgowing@dsg.cs.tcd.ie, Jan 1994.


BIBLIOGRAPHY


