1999

Synchronisation of composite operations in multithreaded object-oriented programs

Michael Lawler
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.
Synchronisation of Composite Operations in Multithreaded Object-Oriented Programs

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 Lawler

Bachelor of Computer Science, Honours Class 1, (Wollongong)

School of Information Technology and Computer Science

1999
Dedicated to Robert and Lola
Declaration

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

Michael Lawler
Abstract

This thesis examines the conflicts between the design ideals of object-orientation and the forms of synchronisation necessary for coordinating concurrency within multithreaded object-oriented programs, and presents an automatic runtime scheduling mechanism that resolves these conflicts.

In the context of composite operations involving multiple objects, existing approaches to providing synchronisation either violate the encapsulation of the participant components, or fail to address important issues in the areas of atomicity and isolation. This thesis provides a detailed description of a meta-layer synchronisation scheme which specifically addresses these conflicts. Invocations on shared objects are managed dynamically at run-time by a generic meta-level scheduling system. Meta-information describing operations is used by the generic scheduler to reason about object interaction within invocations, allowing additional control over the evaluation, scheduling and execution of concurrent operations.

In contrast with more static approaches to synchronisation, the generic meta-layer scheme allows the exploitation of object-oriented principles such as encapsulation, inheritance, and generalisation in the design of shared components for concurrent systems. It adapts to the flexibility and dynamism possible in object-oriented systems, while implicitly providing safety for both simple and composite operations on constellations of shared objects.
Acknowledgements

Firstly, I would like to express deep gratitude to my supervisor Associate Professor Neil A. B. Gray for his guidance and support throughout the period of both this thesis and my prior studies at the University of Wollongong. My sincere thanks are also extended to Sergio Giason for his cheerful willingness to be engaged in exhausting bouts of technical discussion well beyond the call of good friendship. I am also grateful to Associate Professor Greg Doherty for supervision in Neil's absence. My thanks to all of the people mentioned above for reading and commenting on various drafts of this thesis.

Additionally, I would like to express my appreciation of the support and assistance provided by the Intelligent Systems Research Group, and the administrative and support staff in the School of IT and Computer Science.

Thanks to friends and family for putting up with me, especially during the last few months when I have been so absorbed in the writing of this thesis.

Finally, I am indebted to my mother and father who have been a constant source of encouragement and support during the difficult times in this work.
2.5.2.2 Atomicity ........................................................................................................... 23
2.5.2.3 Isolation ............................................................................................................ 23
2.5.2.4 Atomicity for Composite Operations ................................................................. 23
2.5.2.5 Root Operation ................................................................................................. 24
2.5.3 Access Synchronisation in Composite Operations ........................................... 24
  2.5.3.1 Access Synchronisation Managed by Target ............................................... 26
  2.5.3.2 Access Synchronisation Managed by Caller ............................................... 29
  2.5.3.3 Caller vs. Target vs. Mediator ...................................................................... 30
2.5.4 Providing Atomicity and Isolation for Composite Operations ....................... 31
  2.5.4.1 Concurrency Control Mechanisms in Database Transactions .................... 31
  2.5.4.2 Two-Phase Locking ....................................................................................... 32
  2.5.4.3 Conflict Resolution ....................................................................................... 32
  2.5.4.4 Conflict Prevention ....................................................................................... 33
2.6 A Meta-Level Approach ....................................................................................... 33
  2.6.1 Rationale .......................................................................................................... 33
    2.6.1.1 O-O ............................................................................................................. 33
    2.6.1.2 Synchronisation ......................................................................................... 34
    2.6.1.3 Mechanics of Serialisation ....................................................................... 35
    2.6.1.4 Solution ....................................................................................................... 36
  2.6.2 Goals ................................................................................................................ 36
    2.6.2.1 Safety ......................................................................................................... 36
    2.6.2.2 Compatibility with O-O Principles ............................................................. 36
    2.6.2.3 Composite Operations .............................................................................. 37
    2.6.2.4 Liveness ...................................................................................................... 37
2.7 Related Work ......................................................................................................... 37
  2.7.1 Active Object .................................................................................................... 37
  2.7.2 Structured Transactions .................................................................................... 38
    2.7.2.1 Standardised Transaction Protocol ............................................................. 38
    2.7.2.2 Transaction Control Policy ....................................................................... 39
  2.7.3 SCOOP ............................................................................................................... 39
    2.7.3.1 Animation Model ...................................................................................... 40
    2.7.3.2 Dual Call Semantics ................................................................................. 40
    2.7.3.3 'Separate' Concept and Validity Rules ....................................................... 40
    2.7.3.4 Separate Call Rule and Object Reservation Mechanism .......................... 41
  2.7.4 SOS Paradigm .................................................................................................. 41
    2.7.4.1 Guard-Based Approach ......................................................................... 42
    2.7.4.2 Unsafe Access to Instance Variables ...................................................... 42
3 A Meta-Level Scheduler ......................................................................................... 43
  3.1 Introduction ........................................................................................................... 43
  3.2 Overview .............................................................................................................. 44
  3.3 Shared Object Model ......................................................................................... 47
    3.3.1 Uniformity of Access .................................................................................... 48
    3.3.2 Methods as Objects ..................................................................................... 48
    3.3.3 Operation Meta-Information Declarations ............................................... 49
      3.3.3.1 Target Access Mode ............................................................................... 50
      3.3.3.2 Shared Object Interaction ................................................................... 50
      3.3.3.3 Shared Object Reference Field Declarations ....................................... 51
      3.3.3.4 SharedObject Action Declarations ....................................................... 53
    3.3.4 MethodMetaInformation Objects ................................................................. 55
3.8.5 Application Condition-Based Deadlock

3.9 Discussion
3.9.1 Main Features
3.9.2 Benefits
3.9.2.1 Preserving Encapsulation of Shared Components
3.9.2.2 Increased Robustness of System Access Synchronisation
3.9.2.3 Increased Robustness of System Condition Synchronisation
3.9.2.4 Compatibility with Polymorphism
3.9.2.5 Compatibility with Inheritance
3.9.2.6 Increased Maintainability and Reusability
3.9.2.7 Pervasive Readers/Writer Access Policy
3.9.2.8 Avoidance of Potential Liveness Failure
3.9.3 Costs
3.9.3.1 Memory Overhead
3.9.3.2 Processing Overhead
3.9.3.3 Object Model Heterogeneity
3.9.4 Issues
3.9.4.1 Control Over Granularity of Atomic Actions
3.9.4.2 Liveness and Availability

4 Implementation Issues in Proof-of-Concept

4.1 Introduction

4.2 Shared Object Model
4.2.1 SharedObject Base Class
4.2.2 Method Definition
4.2.2.1 Methods are Inner Classes
4.2.2.3 Method Implementation Function
4.2.2.4 Meta-Information Declaration Function
4.2.3 Condition Definition
4.2.3.2 Condition Implementation Function
4.2.4 Method Meta-Information
4.2.4.1 Shared Object Reference Field Declarations
4.2.4.2 Shared Object Action Declarations

4.3 Invocation Mechanism
4.3.1 Invocation Syntax
4.3.1.1 Method Invocation
4.3.1.2 Condition Test Invocation
4.3.1.3 Conditional Invocation
4.3.1.4 Conditional Invocation with Consequent Method Invocation
4.3.2 Invocation Reification
4.3.2.1 Method Invocation Reification
4.3.2.2 Conditional Invocation Reification
4.3.3 Execution
4.3.4 Return Value
4.3.5 Type Checking
4.3.6 Dynamic Method Lookup and Inheritance

4.4 Invocation Scheduling Mechanism
4.4.1 Scheduler Architecture
Contents

4.4.1.1 Architecture Overview ................................................................. 121
4.4.2 Method Invocation Processing.......................................................... 123
4.4.2.1 Method Invocation Sequence......................................................... 124
4.4.3 Conditional Invocation Processing .................................................... 129
4.4.3.1 Conditional Invocation Sequence ..................................................... 129
4.4.4 Intra-Scheduler Concurrency ............................................................ 133

5 Demonstration Examples ................................................................. 135
5.1 The Dining Philosophers Problem ...................................................... 137
5.1.1 Problem Description ......................................................................... 137
5.1.2 Analysis ......................................................................................... 138
5.1.3 Example Solution ............................................................................ 138
5.1.3.1 Overview ..................................................................................... 138
5.1.3.2 Dining Philosophers Source Code ................................................... 139
5.1.3.3 Execution Output ......................................................................... 144
5.1.4 Discussion ....................................................................................... 144
5.2 The Cigarette Smokers Problem ............................................................ 147
5.2.1 Problem Description ......................................................................... 147
5.2.2 Analysis ......................................................................................... 148
5.2.3 Example Solution ............................................................................ 148
5.2.3.1 Overview ..................................................................................... 148
5.2.3.2 Smokers Problem Source Code ...................................................... 149
5.2.3.3 Execution Output ......................................................................... 156
5.2.4 Discussion ....................................................................................... 157
5.3 The Sleeping Barber ............................................................................. 159
5.3.1 Problem Description ......................................................................... 159
5.3.2 Analysis ......................................................................................... 160
5.3.3 Example Solution ............................................................................ 162
5.3.3.1 Overview ..................................................................................... 162
5.3.3.2 Sleeping Barber Source Code ......................................................... 163
5.3.3.3 Execution Output ......................................................................... 172
5.3.4 Discussion ....................................................................................... 173
5.3.4.1 Customer Thread ......................................................................... 173
5.3.4.2 Barber Thread ............................................................................... 175
5.4 Adding Additional Active Participants
Cigarette Smokers Revisited ................................................................. 176
5.4.1 Extension Description ....................................................................... 176
5.4.2 Analysis ......................................................................................... 176
5.4.3 Implementation ............................................................................... 177
5.4.3.1 Source Code ............................................................................... 177
5.4.3.2 Execution Output ......................................................................... 178
5.4.4 Discussion ....................................................................................... 178
5.5 Polymorphism, Inheritance and New Behaviour
Dining Philosophers Revisited ................................................................. 180
5.5.1 Extension Description ....................................................................... 180
5.5.2 Analysis ......................................................................................... 180
5.5.3 Implementation ............................................................................... 181
5.5.3.1 Source Code ............................................................................... 181
5.5.3.2 Execution Output ......................................................................... 183
5.5.3 Discussion ....................................................................................... 184

x
6 Alternative Implementations

6.1 Optimistic Locking Approach
   6.1.1 Context
   6.1.2 Alternative Design
      6.1.2.1 Conflict Detection
      6.1.2.2 Conflict Resolution and Scheduling
      6.1.2.3 Roll-Back Mechanism

6.2 Caching Access Decomposition Results
   6.2.1 Rationale
      6.2.1.1 Computational Costs of a Dynamic Approach
      6.2.1.2 Static Relationships Between Shared Objects
      6.2.1.3 Declaring Member Field References as Nonvolatile Associations
      6.2.1.4 Reusable Static Call Chain Nodes
      6.2.1.4 Characteristics of Static Call Chain Nodes
   6.2.2 Design Extensions
      6.2.2.1 Initialisation of Class MetaInformation
      6.2.2.2 Initialisation of Method MetaInformation
      6.2.2.3 Decouple Call Chain Nodes from Invocation Call Chain Graphs
      6.2.2.4 Cache Static CallChainNode Objects per Method
      6.2.2.5 Revise CallChainNode Factory of Method Class
   6.2.3 Discussion

6.3 Asynchronous Invocation
   6.3.1 Implementation Extensions for Asynchronous Semantics
   6.3.2 'Future' Style Return Values

6.4 Alternative Invocation Ordering Schemes
   6.4.1 Invocation Priority

6.5 Encapsulation of Method Guards

6.6 Invocation Time-Outs
   6.6.1 Invocation Access Conflict Time-Out
   6.6.2 Conditional Invocation Wait Time-Out

7 Conclusion

7.1 Summary

7.2 Future Research

A Framework Class Overview

A.1 Framework Class Hierarchy

A.2 Framework Class Relationships
   A.2.1 SharedObject Class Relationships
   A.2.2 Method Meta-information Class Relationships
   A.2.3 Invocation Class Relationships
   A.2.4 Access Decomposition Class Relationships

B Object Reference Management

B.1 Object Reference Management Classes

B.2 SmartSharedObjectSet
   B.2.1 SmartSharedObjectSet Interface
   B.2.2 SmartSharedObjectSet Object Lifecycle

B.3 SmartSharedObjectTable
B.3.1 SmartSharedObjectTable Interface ................................................. 217
B.3.2 SmartSharedObjectTable Object Lifecycle ...................................... 218
B.4 MemberFieldValueTable ..................................................................... 219
  B.4.1 Synchronising Access to Member Fields Between Competing Invocations ......................................................... 221
  B.4.2 Sharing Potential Values Between Multiple Observer Nodes .......... 222
  B.4.3 MemberFieldValueTable Object Lifecycle ...................................... 222
C Publications From This Thesis ............................................................. 224
Bibliography .......................................................................................... 237
List of Figures

2.1 Single Threaded Process ................................................... 10
2.2 Multithreaded Process ..................................................... 10
2.3 An example Composite Operation ....................................... 25
2.4 Atomicity/Isolation Failure in Composite Operations ............... 27
2.5 Opportunities for Deadlock in Composite Operations ................. 27

3.1 General program architecture showing heterogeneous object model . . 44
3.2 High-Level Components of the Synchronisation Meta-Layer ............ 45
3.3 Invocation Progress Through Synchronisation Meta-Layer ............... 47
3.4 Shared Object Model Component of Meta-Layer Scheme ............... 47
3.5 Comparison of Normal and Shared Object Models ...................... 49
3.6 Structure of Method Meta-Information .................................. 49
3.7 Target Access Modes of Method Objects ................................ 50
3.8 Class Relationship Diagram for SharedObjRefFLds .................... 51
3.9 Invocation Mechanism Component of Meta-Layer Scheme ............... 57
3.10 High Level Comparison of Invocation Mechanisms .................... 58
3.11 Invocation Class Relationship Diagram (Simplified) ................. 58
3.12 Invocation Mechanism Interaction Diagram ............................ 60
3.13 Access Decomposition Mechanism Component of Meta-Layer Scheme ... 61
3.14 Simple Operation "scoreHand()" ........................................ 63
3.15 Composite Operation "turnAttack()" .................................... 63
3.16 CallChainNode Dependencies ............................................ 69
3.17 Generic Invocation Scheduler Component of Meta-Layer Scheme ....... 77
3.18 Conditional Invocation Class Relationship Diagram ................. 90

4.1 Scheduler Architecture Overview ........................................ 123
4.2 Intra-Scheduler Concurrency Scheme ................................... 134

5.1 Dining Philosophers Problem ............................................. 137
5.2 Structure of Example Solution to Dining Philosophers Problem ....... 139
5.3 Philosopher.java Source Code Listing .................................. 141
5.4 Fork.java Source Code Listing ............................................. 142
5.5 PhilosopherThread.java Source Code Listing ......................... 142
5.6 DinersApp.java Source Code Listing ..................................... 143
5.7 DinersApp Execution Output .............................................. 144
5.8 Cigarette Smokers Problem ............................................... 146
5.9 Structure of Example Solution to Cigarette Smokers Problem ......... 148
List of Figures

5.10 Table.java Source Code Listing .......................................................... 150
5.11 Agent.java Source Code Listing .......................................................... 151
5.12 AgentThread.java Source Code Listing ................................................. 151
5.13 Smoker.java Source Code Listing ........................................................ 153
5.14 SmokerThread.java Source Code Listing .............................................. 154
5.15 SmokersApp.java Source Code Listing ................................................... 155
5.16 DinersApp Execution Output ............................................................... 155
5.17 Sleeping Barber Problem ...................................................................... 158
5.18 Class Relationships in Sleeping Barber Solution ................................... 161
5.19 Structure of Example Solution to Sleeping Barber Problem .................. 162
5.20 Chair.java Source Code Listing ............................................................. 163
5.21 Customer.java Source Code Listing ....................................................... 166
5.22 BarberThread.java Source Code Listing ................................................ 167
5.23 CustomerThread.java Source Code Listing ......................................... 169
5.24 BarberApp.java Source Code Listing .................................................... 170
5.25 DinersApp Execution Output ................................................................. 171
5.26 Cigarette Smokers problem with additional participants ..................... 175
5.27 Code Additions to SmokersApp.java ..................................................... 175
5.28 Extended SmokersApp Execution Output .............................................. 176
5.29 Source Code Listing for Dexterous.java .............................................. 180
5.30 Code Modifications to DinersApp.java ................................................ 180
5.31 Extended DinersApp Execution Output ................................................ 181

A.1 Framework Class Inheritance Diagram .................................................. 208
A.2 SharedObject Class Relationship Diagram ............................................ 209
A.3 MethodMetaInfo Class Relationship Diagram ........................................ 210
A.4 Invocation Class Relationship Diagram ............................................... 211
A.5 CallChainNode Class Relationship Diagram .......................................... 212

B.1 SmartSharedObjectSet Class Diagram .................................................. 214
B.2 SmartSharedObjectSet Class Diagram .................................................. 216
B.3 MemberFieldValueTable Class Diagram .............................................. 219
List of Tables

3.1 Creating SharedObject reference fields for use in MethodMetaInformation Declarations ...................................................... 53
3.2 Declaring Shared Object Interaction ....................................................... 54
3.3 Access Table for an Invocation of Simple Operation "scoreHand()" ........ 63
3.4 Access Table for an Invocation of a Composite Operation ................ 64
3.5 Method Invocation State Lifecycle ...................................................... 86
3.6 Conditional Invocation State Lifecycle .............................................. 95

5.1 Access Table for Invocation Philosopher.eat ........................................ 145
5.2 Access Table for Invocation placeRandomIngredients ....................... 156
5.3 Access Table for Invocation takeRollSmoke ......................................... 156
5.4 Access Table for Invocation findAChair ............................................. 172
5.5 Access Table for Invocation swapChairs ............................................. 172
Chapter 1

Introduction

This thesis examines the conflicts between the design ideals of object-orientation and the forms of synchronisation necessary for coordinating concurrency within multithreaded object-oriented programs. A scheme is presented which resolves these conflicts via a dynamic runtime scheduling mechanism.

Object-orientation is a paradigm that allows programmers to design and organise software as a collection of discrete objects that encapsulate both data structure and behaviour. Object-orientation provides useful rules and abstractions which allow the programmer to manage the complexity of modern software systems. The object-oriented approach includes powerful features such as encapsulation and information-hiding, polymorphism and client generalisation, and reuse through inheritance and composition. These features enable the design and construction of complex software that is highly decoupled, whose elements are independent of each other, and therefore highly flexible, maintainable, and reusable.

Multithreaded software architectures have become increasingly more important within the last decade. Multithreaded applications exploit multiple threads of control running concurrently (or pseudo-concurrently) within a single program to obtain benefits in such areas as throughput, communication, program structure, hardware parallelism, availability and use of system resources. However, multithreaded applications can be difficult to design due to the complexities arising from interdependencies between concurrent activities. Due to the shared memory and process state, and the ease of access to shared resources, programming with multiple threads requires more care and discipline than does single-threaded programming. Various forms of synchronisation are required for reliable execution in order to ensure safe access to shared resources by concurrent threads.
Synchronisation, the coordination of intra-program concurrency, however, poses different concerns to the encapsulation focus of object-orientation. In many cases, the operations of one thread are inherently dependent on the state of components belonging to another thread within the program. For example, in a producer-consumer system, if the consumer components are unable to process or buffer any more data, the producer's operations must be suspended; thus, correct operation of the producer requires knowledge of the state of the consumer. Similarly, when shared data must be updated by separate threads, these threads must coordinate their activities so as to avoid races leading to inconsistent updates, and must schedule their operations to avoid conditions such as deadlock.

The integration of multithreading with the object-oriented approach is hindered by conflicts between object-oriented design ideals and the mechanics of synchronisation required for reliable computation in a concurrent environment. The underlying source of conflict stems from their inconsistent views of the part/whole relationships of component object behaviour within a system. The object-oriented approach leads to a flexible and abstract view of part/whole relationships. Firstly, the principle of encapsulation allows the private implementation of a component’s behaviour to be considered independently of the other software elements within a system. Secondly, the principle of generalisation enables a high degree of flexibility in combining discrete objects in different ways when building composite functionality involving multiple components. The modularity of the object-oriented approach leads to flexible designs in which the parts of a system cooperate seamlessly without explicit knowledge of the whole.

In contrast, the mechanics of reliable synchronisation require an accurate, concrete view of the part/whole relationships between various aspects of a system, as it deals with the coordination of such aspects to prevent interference. For example, in order to coordinate the progress of a thread running in one object, a synchronisation scheme may need knowledge of the progress of other threads within the system. In another situation, a synchronisation mechanism might need to violate the encapsulation of some data manager object in order to identify and lock all structures that might be affected by an update operation. Thus, for reliability, synchronisation schemes often require explicit knowledge of the parts within a whole.

Execution issues such as atomicity and isolation become complicated when considered in the context of composite operations involving multiple objects. In addition, shared data objects may involve dynamic and temporal relationships which can only be determined at runtime. It is often difficult to find the appropriate structure for and
placing of the synchronisation code needed in such situations. Synchronisation code devised for one case may prove inappropriate for related situations, resulting in a loss of the reusability often claimed for object-oriented approaches.

The combination of the object-oriented approach and concurrency complicate the programmer's task. But there are strong forces driving program development in the direction of a combined multithreaded object-oriented approach. Symmetric multiprocessor CPU systems are becoming more common and providing direct hardware support for such software architectures. Already, the effective operation of many programs such as network browsers depends on the workings of multiple threads to provide interactive response while sustaining multiple network connections. This form of software architecture will become more common, with the overall operations of a program being realised through multiple threads handling subtasks. These threads may be partially independent, working with private constellations of objects, but will interact through some subset of shared data. The scheduling of thread access to shared data is a problem that programmers will encounter more frequently, and as such, automated mechanisms for resolving the synchronisation problems in an OO system would greatly benefit programmers.

This thesis argues that existing approaches to managing the synchronisation of composite operations in object-oriented programs fail to address many significant issues. This thesis introduces a meta-layer scheduling scheme for managing synchronisation in the general contexts of multiple independent tasks concurrently invoking composite read/write operations on dynamic constellations of shared objects. The meta-level scheme controls access synchronisation at the operation invocation level, such that access synchronisation becomes implicit instead of explicit. The novel aspects of the meta-layer scheme can be summarised as follows:

- a shared object model with new abstractions for describing the access requirements of object operations.
- a dynamic invocation mechanism which automatically provides atomicity of access synchronisation for both simple and composite operations on shared objects.
- a generalised scheduling mechanism which ensures serialisation of operations on shared objects via a static two-phase locking approach, avoiding liveness failures such as deadlock and starvation.
Chapter 1. Introduction

- a new approach to condition synchronisation compatible with the automatic access synchronisation implicitly provided for all composite operations on shared objects.

- reconciliation of the modularity, flexibility and dynamism of the object-oriented approach with the requisite mechanics of safe and reliable synchronisation for composite operations over multiple shared objects.

Some preliminary results of this research were presented by the author at the Third International Conference on Object-Oriented Technology (WOON’98 – July 1998, St. Petersburgh, Russia). Appendix C provides a full reproduction of the paper appearing in the proceedings of this conference.

The remainder of this thesis is structured as follows. Chapter 2 provides an overview of the field of synchronisation in multithreaded object-oriented programming. It defines the context for this thesis and identifies particular issues in the synchronization of composite operations in object-oriented programs that are to be addressed. Related work is discussed in association with the previously identified issues. Chapter 3 presents a high-level description of a proposed meta-level synchronisation scheme that addresses these issues. Chapter 4 discusses the issues in a proof-of-concept implementation of the proposed scheme which was implemented in the Java™ programming language. Chapter 5 works through demonstration examples involving both access and condition synchronisation in the scheme using some classical concurrency problems. In addition, the examples show the benefits of the approach in the areas of reuse, dynamism, inheritance and maintenance. Chapter 6 addresses issues in possible alternative implementations of, and design extensions to, the high-level definition of the proposed meta-layer synchronisation scheme. Chapter 7 briefly summarises the conclusions of this thesis and outlines possible areas for further research. Appendix A provides an overview of the classes of the scheme and class relationship diagrams for the major components. Appendix B describes some specialised collection classes used for managing multiplicities of object reference values.
Chapter 2

Synchronisation in Multithreaded Object-Oriented Programs

This chapter provides an overview of multithreaded object-oriented programming, and discusses the issues which are the source of conflict between the goals of object-orientation and the necessities of synchronising intra-program concurrency. Section 2.1 provides a brief introduction to object-orientation, focusing on features relevant to later discussion. Section 2.2 introduces multithreading as a form of concurrent execution. Section 2.3 presents an overview of different forms of synchronisation necessary for controlling intra-program concurrency. Section 2.4 provides brief descriptions of a range of synchronisation mechanisms. Section 2.5 discusses issues in the synchronisation of multithreaded object-oriented programs and describes the conflicts between object-orientation and synchronisation schemes that form the basis for the work presented in this thesis. Section 2.6 introduces the concept of a generalised meta-layer scheduler for synchronising invocations of operations on shared objects. Section 2.7 discusses existing work that relates to the synchronisation of composite operations in concurrent object-oriented environments.

2.1 Object-Oriented Programming

As the computational power of computing hardware has increased, so too has the size, functionality and complexity of application software being written for it. Object-orientation is a general purpose programming paradigm that offers opportunities for improved software productivity by allowing the programmer to manage the inherent complexity of software systems.
Object-oriented programming views a program as a collection of discrete objects. An object is an encapsulation of state (data values) and behaviour (operations). The behaviour of an object is dictated by the object's class.

The major benefits in an object-oriented approach derive from the features of encapsulation, inheritance and generalisation.

2.1.1 Encapsulation

Programming languages have evolved to provide expressive and safe abstractions for manipulating data. The evolution of such abstractions starts at the assembly language programmer's access to storage locations such as bytes and registers. Next, early high level languages offered the programmer built-in types, like integers and floating point numbers, which provided greater ease in organising data and greater safety in manipulating it. The introduction of user defined types allowed the programmer to define application specific types. This allowed the programmer to organise related data items into records and then treat the resultant data structure as a unit. The development of abstract data types (ADTs) extended this concept in two important areas. Firstly, it allowed the type specification to encompass a set of operations that can be performed on a particular instance of the structure. Secondly, it provided the ability to separate a type's public interface from its hidden or private implementation. The combination of these two features enabled the encapsulation of abstract data types. Finally, object-orientation extended the concept of ADTs by combining the principle of encapsulation with the principle of inheritance.

The encapsulation of data structures and their operations is a key concept of the object-oriented approach [GOP90]. The public aspects of a class declaration specify the services provided by an instance, while access to the data elements and the implementation of the operations that manipulate them are kept private to prevent unwanted alteration. If a class is well designed, a client cannot determine internal details of an instance and cannot become dependent on such details. In effect, the implementation details of a class are immaterial to the client which sees merely a 'black box' object that can respond to varied requests for services.

Encapsulation is an important feature of object-orientation because it limits the possibilities of coupling between software components to that of their public interfaces. This aids in constructing modular programs and also ensures that parts of a program that access a resource are protected from future possible changes to the implementation of the resource.
Chapter 2. Synchronisation in Multithreaded Object-Oriented Programs

2.1.2 Inheritance

Object-orientation makes an important addition to the concept of ADTs by allowing the incremental specification of related classes. Inheritance is the sharing of attributes and operations among classes based on a hierarchical relationship. Instead of defining interface and implementation code monolithically for each individual class, the programmer can build on existing class definitions by inheriting data and operations and then tailoring behaviour through incremental modifications. The newly defined class is called a subclass of its parent or super classes. A subclass can introduce new operations or selectively reimplement some operations inherited from its parent class. Even if a subclass re-implements an operation, it still has the ability to invoke the superclasses corresponding operation via a "super" call.

The ability to factor out common properties of several classes into a common superclass and to inherit the properties from the superclass can greatly reduce repetition within designs and programs, and is one of the main advantages of an object-oriented approach.

2.1.3 Generalisation

The use of inheritance to construct hierarchies of related classes introduces a form of polymorphism into the class system. An instance of a subclass is considered type equivalent to an instance of the superclass, because it provides at least the same set of services. If a number of specialised subclasses inherit from the same base class, then an instance of any of the subclasses may be substituted for an instance of the base class. This means that a field defined to hold an instance of a particular class may alternatively hold an instance of some specialised subclass.

Dynamic or late binding is an important feature of object-oriented languages that defers resolution of the actual method implementation for an operation on an object held in a field until the point of method invocation at runtime. The language runtime determines and executes the correct operation based on the signature of the operation and the actual class of the target instance of the invocation.

Dynamic binding promotes generalisation in programs by allowing the creation of abstractions which cover a variety of special cases. Client code which requests operations on polymorphic classes of objects, can invoke a particular operation on an instance without needing to know the particular subclass to which the instance belongs. New subclasses which manage additional special cases may be added to the
system without requiring changes in any of the existing code which performs the general requests.

2.1.4 Association

Although classes are defined individually as separate entities, computation usually proceeds via interaction between associated object instances. When handling requests for its services, an object may often rely on nested interaction with other objects by invoking suboperations. The high-level functionality of an object-oriented application is often the result of complex nested interaction between the component objects of the system. Two forms of association that occur frequently in object-oriented designs are collaboration and aggregation.

Collaboration or delegation is a form of association between independently existing objects. Collaboration may be symmetrical – where both objects interact with each other and hence are both aware of the association, or asymmetrical – where a client object requests operations on a server object; while the server object performs the operations without requiring any knowledge of the client making the requests.

Aggregation is a stronger form of association in which an aggregate object is composed of other component objects [RBP+91]. The components and the aggregate are bound by some form of a part/whole relationship. Component parts may or may not exist apart from the aggregate or may appear in multiple aggregates. Some operations applied on the aggregate object may inherently involve sub-operations on component objects.

Aggregation is distinguishable from collaboration by examining whether one object is responsible for the lifetime of the other object (aggregation), or whether the two objects are created, are utilised, and are disposed of independently (collaboration).

Associations between objects may often change over the lifetime of an executing system. Such relationships are called dynamic associations.

The associations between groups of objects often form directed graph structures, where the nodes of the graph are the objects themselves, and the edges represent interactions between associated objects. Often in dynamic situations, the details of the objects involved or the levels of nesting can only be determined at runtime. These situations arise naturally from the use of the object-oriented principles of encapsulation and generalisation.
2.1.5 Benefits of Object-Oriented Programming

The use of an object-oriented design approach offers many important advantages in software productivity and in managing the complexity of large software systems.

2.1.5.1 Reuse

The themes of abstraction, encapsulation and generalisation allow programmers to create software components that are reusable in different applications. Inheritance enables the reuse of data structures and behaviours and allows tailoring through incremental modifications.

2.1.5.2 Maintenance

Encapsulation and generalisation enable a reduction in the interdependency among software components. This allows the construction of complex software that is highly decoupled, whose elements are independent of each other, and therefore highly flexible and maintainable. The software is easier to maintain because the internal implementations of classes may be modified or redesigned without affecting other components in the system. The loose coupling between components enables them to be maintained and tested in isolation from the complete system.

2.2 Multithreaded Programming

2.2.1 Concurrency

Concurrency is employed in many forms in modern computing systems. Operating systems support concurrent processes by time-sharing, which allows a single machine to execute several processes at once. Concurrency involving multiple processes has an established history in UNIX environments. A process is an operating system (O/S) execution abstraction in which each process maps to a unique address space. O/S resources are attached to single process. Processes are heavyweight, comparatively expensive to create, and have their execution scheduled by the operating system. Multiprocessing allows several separate applications to proceed in parallel on a single machine. Increasingly, however, the users of software desire concurrency within a particular application.

In contrast to processes, a thread is a lightweight abstraction for concurrency which executes within a process structure. A thread is a single, sequential flow of control within a program. Within each thread there is a single point of execution. Most
traditional programs consist of a single thread. However, multiple threads may execute concurrently within the same process address space, and all threads within a process share the same O/S resources. Figure 2.1 and Figure 2.2 illustrate the differences between a single-threaded process and a multi-threaded process. Threads are relatively inexpensive to create and their execution may be scheduled by application program code. Whereas multiprocessing is a mechanism for achieving concurrency between application programs, multithreading is a mechanism for achieving concurrency within application programs.

Concurrency offers many performance benefits (throughput, computational speed, responsiveness – or some combination). Programs involving expensive computations
that take a long time to run may exploit hardware parallelism to reduce execution time by dividing the work among separate tasks that can be executed concurrently on the different CPUs of a multiprocessor system. Threading can even speed up programs executing on single CPU systems, as the execution of computationally intensive threads may overlap with other i/o bound threads that are blocked waiting for slower synchronous i/o operations to complete and return.

Concurrency allows applications to maintain high availability of service by executing service requests in threads separate to the main handler. Further, when driving slow devices such as disks, networks, terminals and printers, a multithreaded program can perform other useful work while waiting for the device to produce its next event.

In addition, many types of applications exhibit concurrency naturally, such as reactive programs, in which an application performs different activities as reactive responses to different types of (possibly concurrent) input. Implementing these types of applications in sequential languages is often complicated and error prone; design and implementation is much easier using threads.

2.2.2 Concurrent Object Models

Different concurrent object-oriented languages use different object models to support a variety of features and enforce certain constraints. In a survey of concurrent object-oriented languages [KL95], Kafura and Lavender discuss an object animation model that describes the possible relationships between threads and objects. The major aspect of the animation model describes the extent to which threads of control respect object boundaries. Kafura and Lavender describe two opposing approaches: related and unrelated.

In the 'unrelated' approach, threads and objects are orthogonal concepts. This is also called a passive object model. Threads of control do not respect object boundaries and execute instructions independently of the object model concept. Through method invocation, an executing thread is free to cross object boundaries at will, similar to stack-based subroutine invocations in conventional sequential programming languages. The combination of a threads facility and an object-oriented language allows concurrent O-O with the unrelated model. SunSoft's Java programming language [AG97] and many general purpose concurrent implementations of C++ employ an unrelated approach to the object animation model.

In the 'related' approach to the animation model there exists a deeper integration between concurrency and the object model concept. This is also called an active object
or per-object-concurrency model [Lea97]. Under this scheme, a thread of control is encapsulated within a particular object instance and may not operate outside the bounds of its host object. This introduces a significant change in the way objects interact: as well as each object being a supplier of services to other objects, each object has its own active agenda. This model requires synchronisation mechanisms to control communication between individual objects. The active object model can clash with the inheritance concept [MY93, Mey97]. One well known example of this approach is the 'Actor' model of concurrent computation [Hew77, Agh86].

2.2.3 Concurrency Issues

The multiple threads executing within a particular program use the same memory address space, and hence have the potential to concurrently execute operations on the same object instances. If uncontrolled, threads executing concurrently over common interacting objects may interfere with each other. There are two issues related to preventing interference in concurrent programs: safety and liveness.

2.2.3.1 Safety

Safety is concerned with ensuring that interference between activities can not occur. There are three common strategies for preserving the safety of shared objects: immutability, synchronisation and containment.

Immutability is an approach that guarantees safety because it does not allow any changes to the state of shared object instances. Such classes do not define any mutative operations. Once an immutable object is created, its internal state can never be modified, thus limiting all forms of access to 'read' style or inspective access. Hence, concurrent access can never result in any form of interference. Selective use of immutability can be an effective tool for some situations.

A synchronisation approach is required when the objects to be shared define operations that may involve direct changes to internal state. A synchronisation policy is implemented to ensure that concurrent requests for such operations are performed only in consistent ways.

The containment approach provides safety by encapsulating shared component objects structurally within others, as private entities never exposed to other objects, so that the components can only ever be accessed directly by the parent container. This simplifies the task of preserving the safety of such components because the component
implementation itself does not need to bother with synchronisation issues. All access can be coordinated by a synchronisation policy implemented at the container level.

2.2.3.2 Liveness

Liveness is concerned with ensuring that the progress of concurrent activities is not halted unnecessarily for some reason. Any use of synchronisation can possibly lead to liveness issues. Two reasons for liveness failure are starvation and deadlock.

Starvation occurs when a thread is logically runnable but fails to proceed because of contention for resources from other competing threads. Starvation is often a scheduling issue, resolvable by ensuring fair selection between threads competing for access to a common resource.

Deadlock occurs when two or more threads block each other in such a way that none of them can possibly proceed. The occurrence of access deadlock is usually due to ordering conflicts between threads incrementally acquiring separate synchronisation locks.

2.3 Synchronisation

The purpose of synchronisation is to control and coordinate the execution of concurrent activities in order to prevent interference and thus ensure safe, reliable computations. There are two major forms of synchronisation: access synchronisation and condition synchronisation [And91]. Access synchronisation is resource-centric and is concerned with ensuring safe access to resources in order to preserve data integrity. In contrast, condition (or state-based) synchronisation is thread-centric and is concerned with the communication between separate threads of control – for example, delaying one process until another has finished its activities, or signaling to another process that a given condition is true.

The coordination of concurrent activities can be achieved through the explicit use of synchronisation primitives. However, in many concurrent object-oriented programming languages, synchronisation is often associated with the method invocation mechanism.

2.3.1 Access Synchronisation

Uncontrolled access to objects shared by concurrent threads is sometimes unsafe, resulting in interference. For example, if one thread attempts to examine an object while another thread is concurrently updating it, the first thread may see the object in an
intermediate or possibly inconsistent state. In the same way, if two threads attempt to update the same object at the same time, the end result may be an inconsistent meld of both operations.

To prevent interference problems, a system must support a means of controlling access to shared resources.

The particular synchronisation policy used when accessing an object often varies depending on the class the object belongs to. The correct choice of policy depends on the possibilities for intra-object concurrency between the operations provided by the class.

The simplest form of access policy is mutual exclusion. This type of policy is suitable when only a single thread is allowed to interact with the particular object at any one time. That is, no two operation requests may be executed concurrently on the one object.

In many situations, multiple threads may share a data structure in a manner that entails read access for the majority of cases, and write access only occasionally. If the data structure is large, or the access operations involve a relatively large amount of processing, then using a mutual exclusion policy to protect the resource may be inefficient - reader threads compete for exclusive access when they would not interfere with each other anyway. In such cases, the readers/writer policy provides greater opportunity for parallelism. In this policy, all access operations are characterised as either 'read' or 'write' requests. The policy allows requests for 'read' access by multiple independent threads to proceed concurrently, while ensuring that all 'write' access operations proceed with exclusive access.

Access synchronisation for operations on shared data may be achieved explicitly or implicitly. The explicit use of synchronisation primitives within operation implementations allows manual control over access to shared data structures. Alternatively, a suitable conditional method invocation interface provides implicit access synchronisation for operations.

2.3.2 Condition Synchronisation

Condition synchronisation is concerned with communication between separate threads of control. It allows a thread to wait, if necessary, until a given condition is true. For example, in a producer-consumer system, the interaction between producer and consumer often occurs through a shared buffer. The producer adds items to the buffer while the consumer removes items. If the buffer becomes full, then the producer must
temporarily suspend production until the consumer has cleared some space in the buffer. In this case the producer’s implementation may employ condition synchronisation to wait on the condition that the buffer is not full, before attempting to insert additional items into the buffer.

As with access synchronisation, condition synchronisation for operations may be achieved explicitly or implicitly. Explicit use of condition synchronisation primitives in a sequence of instructions enables one thread to wait for, or signal, a condition to be true. Alternatively, via a conditional method invocation interface, state-based conditions can be associated with operations in order to provide implicit condition synchronisation for invocations of such operations.

2.3.3 Invocation Control

Kafura & Lavender [KL95] discuss a synchronisation model, that considers what, if any, controls are imposed upon the execution of concurrent invocation requests on an object. Invocation control is an important synchronisation issue because it allows the management of concurrent interaction, at the operation level, to preserve the integrity of an object’s state. Invocation control in multithreaded object-oriented languages may be unconditional or conditional.

An unconditional interface provides no control over invocations; operations are executed without regard to the state of the target object. In these cases, designs require the explicit use of synchronisation primitives within method implementations to control and coordinate the progress of concurrent requests. Such solutions increase complexity, are error-prone and detract from reusability.

 Conditional interfaces allow for control over invocation requests – an invocation is subject to postponement until particular conditions related to the state of the target object are satisfied. There are different approaches to building conditional interfaces. Concrete approaches such as guarded accepts and guarded ports in CSP and Ada encapsulate the details of the conditional interface within the object. More abstract mechanisms allow for separating the details of the conditional interface from the implementation of the object itself.

2.3.4 The Inheritance Anomaly

It has been identified that the use of inheritance in concurrent object-oriented programming can be problematic, hindering code reuse and leading to the violation of
Chapter 2. Synchronisation in Multithreaded Object-Oriented Programs

encapsulation [KL89] [MY93] [MW94] [McH94]. This problem is termed the inheritance anomaly.

In the object-oriented approach it is desirable to extend a class’ behaviour through incremental, additive modifications – in order to maximise reuse of the super class’ code. However, the synchronisation scheme associated with a class definition is often highly coupled to the sequential instructions of its operation's implementations. Such coupling can hinder the incremental extension of individual functionality. Problems can occur in a number of ways.

2.3.4.1 Integration of Subclass Synchronisation Behaviour

Firstly, it may be difficult to integrate the additional synchronisation behaviour required for the specialised operation behaviour of the subclass, with the synchronisation scheme already defined by the superclass.

In many synchronisation approaches, some form of conditional invocation control is employed in class definitions to explicitly specify for each particular operation, which of the other operations defined by the class may be executed concurrently. This approach is problematic when subclasses define additional operations, because the invocation control scheme defined for the superclass can not take into account these additional operations and their synchronisation requirements. This often leads to considerable respecification of the synchronisation scheme within the subclass.

In other synchronisation approaches for concurrent object-oriented languages (e.g. POOL-T [Ame87]), concurrency control is centralised in a single monolithic 'body' routine which describes potential concurrency between all the operations defined for the class. Individual aspects of the centralised control policy cannot be inherited or reused by subclasses without total respecification of the policy in a new 'body' routine. Hence, this approach often requires a complete respecification of the synchronisation scheme within each subclass to take into account the additional or overridden behaviour.

These situations and their workarounds violate encapsulation as they require implementation-specific information from the superclass to be duplicated within the subclass.

2.3.4.2 Interference from Superclass Synchronisation Behaviour

If an overridden operation in the subclass, with its own synchronisation requirements, invokes an operation in the superclass, the superclass operation may be associated
Chapter 2. Synchronisation in Multithreaded Object-Oriented Programs

with synchronisation behaviour that is either embedded within its implementation or unavoidable in some other way. If the superclass' unavoidable synchronisation behaviour is incompatible with the operation of the subclass, then this may require that the superclass operation's same sequential instructions be reimplemented in the subclass with new synchronisation behaviour. Reuse through inheritance may be difficult to achieve because of the static nature of the superclasses synchronisation scheme.

These problems and their workarounds result in both a loss of code reuse, and violation of the encapsulation of the superclass.

### 2.3.5 Separation of Concerns

The sequential instructions and synchronisation instructions of a class serve different purposes: the former implements the services or operations provided by a class, while the latter is employed to control and coordinate concurrent access to a an instance of the class. Because they serve different purposes, its is often advantageous to separate the synchronisation code and sequential code of a class from one and another [Blo79].

Keeping the two types of instructions separate retains modularity which results in greater reusability and extensibility. It simplifies the incremental extension of functionality through inheritance – subclasses that provide overridden operation implementations which invoke the original superclass' implementations can avoid undesirable interference from the superclass' synchronisation instructions. If the synchronisation instructions are mixed in the operation's implementation then they are unavoidable and may hinder such forms of reuse. Further, separating the two types of instructions allows greater flexibility in defining extended synchronisation policies when existing components are combined in new ways that are not satisfied by the policies effected by each individual component's synchronisation instructions.

### 2.4 Synchronisation Mechanisms

Many types of synchronisation mechanisms, at varying levels of abstraction, have been developed to aid in controlling access to objects. At the low level, simple synchronisation primitives enable explicit manual implementations of access synchronisation policies. More elaborate mechanisms provide a higher level of abstraction allowing specification and control of concurrency at the operation level, simplifying the task of ensuring safe access to shared objects.
2.4.1 Synchronisation Primitives

A semaphore [Dij68] is a classical synchronisation primitive, introduced by Dijkstra. It provides a basic signaling mechanism via two atomic operations, reserve and free. Associating a semaphore with access to a resource can ensure mutually exclusive access to the resource. Although they are commonly used inside more elegant mechanisms, semaphores by themselves are usually considered too low-level for synchronising large complex systems.

A mutex (mutual exclusion object) [LB96] is another simple synchronisation primitive that can be used by multiple threads to ensure the integrity of a shared resource that they access, by allowing only one thread to access it at a time. A mutex has two states, locked and unlocked, and only one thread can lock a particular mutex at any time. A mutex is associated with each shared resource. Each thread in a program locks the mutex before it accesses its associated shared resource, and then unlocks the mutex when it is finished accessing the resource. If the mutex is locked by another thread, then the thread requesting the lock waits for the mutex to be unlocked. In this way, a mutex can be used to protect access to shared resources or sequences of operations through mutual exclusion.

In the same way, a readers/writer lock [LB96] can be used to provide a multiple readers / single writer access protocol. A readers/writer lock provides four operations, lock_reader, unlock_reader, lock_writer and unlock_writer which are used to protect access to a shared resource. The readers/writer protocol allows multiple read-mode accesses to occur concurrently, while ensuring that write-mode accesses execute in mutual exclusion. This allows concurrent execution to increase the liveness and availability of a system.

A condition variable [LB96] is a synchronisation object used in conjunction with a mutex in order to provide a safe environment for testing a condition predicate. A condition variable allows a thread to block its own execution until some shared data reaches a particular state. A thread locks a mutex for some shared data and then tests the relevant condition predicate. If the predicate tests false, the thread waits on the condition variable associated with the predicate. Waiting on the condition variable automatically unlocks the mutex. When another thread that acquires the mutex puts the data in the appropriate state, it wakes the waiting thread by signaling the condition variable. A waiting thread comes out of its wait state with the mutex locked, while any other threads waiting on the same condition remain blocked.
2.4.2 Simple Access Abstractions

Access abstractions have been introduced to languages which hide the details of the primitive synchronisation mechanisms they are based on. As a result, these abstractions are more reliable and easier to use.

Some languages allow the use of an abstraction called a critical region to protect specified sequences of instructions. The mechanism ensures that the critical region of code is executed atomically, and cannot be interrupted or pre-empted by another thread. A runtime mechanism guards the execution of such regions by allowing only one thread to execute through any particular critical region at a time. This is usually achieved by the implicit association of a mutex lock with each critical region defined. A thread must successfully acquire the associated mutex lock in order to begin execution, and the mutex is released automatically upon exiting the critical region.

A conditional critical region [And91] extends the critical region concept by allowing a condition variable to be associated with a section of code. In addition to the mutual exclusion and atomicity provided by the critical region, the predicate associated with the condition variable must also be in the right state for execution to proceed.

The monitor approach [Hoa74] combines critical regions with the modular structure of modern programming languages. A program module may be defined as a monitor. The mechanism enforces mutual exclusion at the routine level, such that only a single thread may execute any routine defined by the monitor at once. The Java programming language uses the synchronized keyword for this functionality at the object level.

2.4.3 Conditional Access Abstractions

Conditional access synchronisation mechanisms operate at a higher level of abstraction by allowing control over the progress of invocation requests via a conditional interface.

In a guard based approach, guarding conditions may be explicitly associated with each operation defined by a class. The runtime system ensures that the execution of any requested operation must wait until the associated guard condition becomes true.

Many guard-based synchronisation mechanisms are based on synchronisation counters. Synchronisation counters were independently developed by Robert and Verjus [RV77] and Gerber [Ger77]. These are a set of variables that provide details of the progress of invocation requests per operation for each object. The counters maintained typically include a count of how many requests have currently arrived at the object, how many
Chapter 2. Synchronisation in Multithreaded Object-Oriented Programs

have started execution, and the total number of invocations that have terminated or finished execution.

Path Expressions [CH73] allow a class to specify constraints on the order of operations executed on an object. A path expression is a loop that specifies the allowable sequence of operations on an instance via a regular expression of operation names.

Enabled Sets is based on behavioural abstraction [KL89] [TS89], another concept incorporated in many synchronisation mechanisms. A class specifies all of the synchronisation states which it may occupy. For each of these states, a set of enabled operations is defined. The implementation body of operations can use a become operation to specify the next state that the object enters – which then dictates the new set of enabled operations.

In general, these existing conditional access abstractions do not consider the synchronisation requirements of composite operations. Instead, they focus on specifying constraints on the execution progress of requests as they relate to the initial or root target of a composite operation, without considering access issues relating to other objects which may be subsequently involved through nested interaction.

2.5 Synchronisation Issues in Composite Operations

The complexity of object interaction resulting from the execution of an initial operation on a single target object may vary tremendously. Simple operations may be defined as those which contain no nested invocations of suboperations, and hence are limited solely to the examination and/or update of the internal state of the target object of the invocation. In contrast, composite operations may be defined as those which contain further nested invocation, perhaps on objects other than the initial invocation target.

In general, existing approaches succeed in meeting the synchronisation requirements of simple operations. However, the object-oriented approach permits software designers a high degree of flexibility in combining different classes together to build composite functionality. The flexibility of the object-oriented approach can conflict with the mechanics required for correct synchronisation of composite operations.

2.5.1 Context

There are many issues that arise in the synchronisation of multithreaded object-oriented programs. However, a detailed analysis is often only applicable in the context of the choice of object model used in a particular environment. In addition, some
synchronisation issues relate to the styles of interaction allowed between concurrent activities in programs, and some issues result from particular trade-offs between opposing forces in program design.

2.5.1.1 Passive Object Model

This analysis assumes the 'unrelated' or 'passive' object animation model, in which threads of control, via method invocation, may jump from one object to another, operating independently of object boundaries. This is the same model used in sequential environments, and is also used in many multithreaded object-oriented environments.

2.5.1.2 Unconstrained Interaction Between Components

Many specific patterns for managing concurrency work largely by limiting the allowable forms of interaction between components. For instance, the unidirectional flow pattern [Lea97] (or pipeline concurrency) achieves simplicity and generality by restricting interaction to the connections between the sequential stages of a directed pipeline. Another well-established pattern is the divide and conquer (or boss/worker) approach, in which a problem is initially split up (by some boss or coordinator) into relatively independent subproblems. The subproblems are then solved concurrently by independent worker threads executing in isolation. When all of the worker threads have finished executing, the results may then be merged or combined as necessary by the boss or coordinator. These models for concurrent computation achieve safety through the enforcement of a structure which constrains the locations of potential interaction between concurrent threads, hence simplifying the implementation of suitable synchronisation instructions.

In contrast to such constrained or well-defined patterns of interaction, this thesis examines the coordination of more general designs, where arbitrarily complex interactions are free to occur between any of the shared components in a system. In accordance with object-oriented design, this approach retains the benefits of flexibility and generalisation. Individual components of designs produced in this way may be extended or modified in ways that introduce new forms of interaction between shared components – without holistic or systematic analysis to determine whether these changes will contribute to a violation of some prior established interaction constraint.
2.5.1.3 Concurrency Structure Orthogonal to Component Structure

Another design approach to controlling concurrency involves structuring the components of a system into separate units of concurrent activity. Systems are defined such that the objects within each separate unit are specifically attached to the one particular thread of control exclusive to that unit. Safety is achieved through the combination of the containment principle and a mechanism for controlling interaction between separate units.

However, the static nature of such an approach introduces limitations on the flexibility of designs, and on the reusability and genericity of the components of such designs. Object-oriented systems often involve associations and interactions between objects that are of a dynamic nature with respect to the objects or the levels of nesting involved. It can be difficult to devise a suitable unit structure which is compatible with the dynamism of the associations between components – for if all of the components are forced to belong to the one unit then there can be no concurrent interaction upon the components individually. In addition there are often dynamic approaches to concurrency. For example, an application may dynamically create threads to handle events or perform tasks on existing objects as required. In such circumstances, the notion of any objects permanently belonging to a particular thread or unit may be inapplicable. Further, such an approach again results in fragile designs which make it difficult to introduce new forms of interaction between component objects in different units.

In contrast, this analysis considers the general case of a dynamic set of threads orthogonal to, and executing over, a dynamic set of shared objects which may maintain dynamic relationships. However, in the context of providing atomicity for concurrent actions upon shared objects, this analysis does not consider the creation of new threads within such atomic actions. New threads of control are assumed to be created outside of atomic actions on shared objects.

2.5.1.4 Aim of Analysis

In the context established above, this analysis examines the access synchronisation issues which arise in the coordination of composite operations involving multiple shared objects. In particular, in this analysis there is a desire to reconcile the goals of object-oriented design;

- encapsulating components to achieve decoupling, flexibility and maintainability,
Chapter 2. Synchronisation in Multithreaded Object-Oriented Programs

- employing generalisation within program and component design, with the necessities of coordinating concurrency;
- preventing interference to ensure safe and reliable computations,
- preventing liveness failure such as starvation or deadlock situations.

2.5.2 Serialising Concurrent Operations

2.5.2.1 Serialisation

Serialisation [Tan92] [Jan85] [Kum96] [Chr96] involves controlling the execution order of concurrent operations in such a way that the results are equivalent to the case in which the operations are executed serially, or one at a time. The serialisation of operations is an issue that has been well researched in the field of transaction processing for concurrent database systems. Serialisation enables safe and reliable computations in concurrent environments by guaranteeing two related properties of execution: atomicity and isolation.

2.5.2.2 Atomicity

Atomicity [Kum96] is the property of execution such that a sequence of instructions executes or appears to execute indivisibly. In relation to the execution of an operation, atomicity guarantees that, from an external point of view, the execution of any requested operation is either waiting to start, or has completely finished; there are no visible intermediate stages. Atomicity guarantees that no other concurrent activities can view the partial effects on the system of an atomic operation.

2.5.2.3 Isolation

Isolation [Kum96] is the property of execution such that an operation executes as if no other operations are executing concurrently, and thus its execution results are equivalent to those obtained by executing operations serially. Ensuring isolation is equivalent to preventing any form of interference.

2.5.2.4 Atomicity for Composite Operations

In the design of safe concurrent interaction it is often necessary to achieve atomicity and isolation properties that extend over composite operations as a whole. If a composite operation involving a sequence of suboperations is to be atomic, then the whole sequence of suboperations must be executed indivisibly as a single monolithic
operation. Competing concurrent threads must not be allowed to access any of the components involved in the composite operation until the whole sequence of operations is completed.

For example, consider a classical banking system problem where one thread is transferring money from account A to account B, while another thread is computing the sum of A and B. The threads should control their access to A and B through some locking mechanism, but such access control must encompass the complete transaction, not just protect individual accesses. Suppose that the first thread locks A while subtracting the nominal amount from it, then releases it, then locks B while adding the amount to it. Further imagine that the second thread locks A while retrieving its balance, then releases it, then locks B for the same purpose. It may occur that the second thread reads both A and B after the first thread has subtracted from A but before adding to B. Thus the second thread has missed the amount involved in the transfer operation being carried out by the first thread. This scenario is called a race condition and is not an acceptable execution quality in many applications.

If execution of the two operations is serialised then such a scenario cannot occur because the execution of the concurrent operations is controlled to preserve the illusion that they execute atomically in isolation, with a consistent view of the system of objects they act upon.

2.5.2.5 Root Operation

In the case of such composite operations, the root operation is defined as the operation where the granularity of requisite atomicity begins. All of the subsequent nested operations (via subinvocation) must be executed on their respective targets in isolation. The atomicity and isolation properties must be preserved over the entire duration of the resultant call chain, until control returns and the initial root operation finishes execution. This point then closes the granularity of the atomic action. This thesis does not consider actions with explicitly nested atomicity or composite granularity.

2.5.3 Access Synchronisation in Composite Operations

The access synchronisation requirements for each operation defined by a class vary with the complexity of nested object interaction within each operation's implementation. Simple operations involve access or manipulation of the target object's internal state, without any nested interaction with other objects. In contrast, composite operations involve nested interaction through calls to operations on other objects;
these may be independent collaborators or other components within an aggregate object.

```
A
void compositeOp()

B b
```

```
void operationA()
void operationB()
void operationC()
bool functionD()
```

Figure 2.3: An example Composite Operation

Many existing synchronisation mechanisms employ a conditional invocation interface which focuses on satisfying constraints on the permitted states of the initial target of each invocation. However, as shown in Figure 2.3, composite operations involve interaction with multiple objects. As a result, it is often necessary to widen such a focus when planning synchronisation mechanisms, so as to take account of the complete set of objects involved in the execution of a composite operation.

Synchronised access patterns must often be planned for all objects involved in an operation so as to provide the atomicity and isolation execution properties and ensure serialisability. The requirements for atomicity and isolation must be considered with respect to all of the objects involved in the entire resultant call chain structure of the initial root operation.

For simple operations, a monitor style lock suffices to provide both atomicity and isolation. However in an analysis of composite operations, synchronisation schemes that work in conjunction with traditional method invocation mechanisms are generally unsatisfactory because they either fail to fully address the issues of atomicity and isolation described, or they solve the problems in a manner that violates object-oriented encapsulation.
Most existing synchronisation approaches may be divided into two categories. In the first, the target of an invoked operation is responsible for ensuring synchronisation. In the second category, the caller of an operation ensures synchronisation.

2.5.3.1 Access Synchronisation Managed by Target

A common approach to synchronisation allocates responsibility for access control on the target object of each individual invocation. Access control may be achieved by either directly incorporating synchronisation code in the target method implementation code, by protecting the operation with a monitor style lock, or by a separate description of a constraint on method execution (via the use of a conditional invocation language feature such as guarded methods or behaviour sets). Regardless of implementation, this approach attempts to free the caller from the burden of synchronisation responsibility by letting the target of each operation manage the synchronisation issues at each individual invocation point. The advantage of this approach is that it allows both the encapsulation of access synchronisation within component definitions, and the use of polymorphism to promote client generalisation, and hence contributes to flexibility, reusability and maintainability.

This approach can provide isolation for simple operations when each is considered individually. However in composite operations, this approach can be problematic in two areas: atomicity/isolation failure and deadlock opportunities.

The synchronisation mechanism protects the target of each individual invocation for the duration of each individual operation. For example, in the composite operation shown in Figure 2.3, the simple suboperations operationA(), operationB() etc., will individually execute atomically and in isolation. The resultant synchronisation effort produced during execution of a composite multi-object operation equates to a progressive series of individual synchronisation points that correspond to the execution path of the call chain. However, the scope of access control provided by each individual synchronisation point is non-associative and does not encompass sequences of operations. Access locks are acquired, released and then reacquired through the progressive execution of subinvocations within the call chain. This allows interleaved access to shared objects by competing concurrent threads. Figure 2.4 shows an interaction diagram of an example of interference between two threads using the classes introduced in Figure 2.3. The atomicity of composite operations can not be guaranteed because concurrent threads can access and interact with objects which have seen only the partial effect of a composite operation. So while the approach...
provides atomicity and isolation during the execution of simple operations, it fails to extend these properties over composite operations.

In addition to atomicity/isolation failure, this approach is inherently prone to access deadlock during execution. As a thread executes through methods of objects, it retains the locks for all objects corresponding to the stack frames in its call chain, only releasing a lock when it returns from a method and pops the frame off its call stack. Research in operating systems and database concurrency control [Tan92] [Kum96] has shown that such a progressively incremental/decremental approach to lock acquisition is prone to deadlock opportunities. Figure 2.5(a) shows a simple association between two classes C and Z and Figure 2.5(b) shows the interaction diagram of an example of deadlock, arising from concurrent threads invoking the respective composite operations of each class at or near the same time.
Detection of such deadlock situations is possible but requires access to the synchronisation state of all of the other threads within the system. However, recovery from such deadlock situations can be a difficult task once they have occurred. At the point of deadlock, the threads involved may have acquired ownership of multiple objects, each of which may be at various transient or incomplete stages of update. Without a universal scheme to coordinate restoration of the objects involved to a
consistent state, the situation may be unrecoverable. Further, some operations on objects may involve side-effects that are not reversible.

Thus, synchronisation schemes encapsulated within the target objects of operations, which operate transparently via progressive execution through the call chain, are unsatisfactory in composite operation situations because of their failure to successfully address atomicity and isolation issues, and their susceptibility to unrecoverable deadlock situations.

2.5.3.2 Access Synchronisation Managed by Caller

In the alternative approach, the caller of an operation is charged with the responsibility of ensuring safe access, by explicitly arranging synchronisation first, and then invoking the operation. Manual access control, through the use of synchronisation primitives, enables strategies which ensure that atomicity and isolation properties encompass composite operations completely. This may be achieved through the construction of a singular monolithic synchronisation event that is applied over the whole composite operation. However, this approach is problematic again, but for different reasons.

Firstly, it is harder to guarantee isolation when separate mechanisms are used for the access control and method invocation. Synchronisation is not automatic or enforceable, and so it is possible that errant or devious code executed by another thread may omit the synchronisation step and invoke an operation directly – leading to a breakdown of the synchronisation scheme and a loss of system reliability.

Secondly, the coding of the client becomes more complicated because the programmer must foresee the ramifications of a planned sequence of operations and so determine the locks that must be acquired. Identification of the affected objects may necessitate that the client be aware of the private implementation details of the target object. However, the encapsulation of such an object is violated if synchronisation instructions specific to its private implementation are distributed throughout client code that requests operations on it. Further, in order to employ access policies more advanced than simple mutual exclusion (for example, a readers/writer policy), details on the characteristics of access must also be gathered. These various problems are typically addressed in an ad hoc fashion leading to complexity, unreliability, and lack of reuse. Collaborating components become highly coupled and implementation dependent resulting in fragile systems with a significantly lower degree of maintainability.
Although this style of approach allows the manual implementation of access control to be performed prior to an operation, it is not always easy or indeed possible to construct a scheme that can handle the dynamic nature of interaction within an object-oriented environment. Any explicitly coded strategy is dependent on the implementation of the operation for which it is being arranged, hence this approach is unsuitable for designs which exploit polymorphism and dynamic binding for generalisation. Additionally, when attempting to identify the set of objects possibly affected by a composite operation, dynamic associations between objects may result in levels of nesting which are too difficult to unravel.

Finally, when using this approach to provide access synchronisation, the task of ensuring atomicity and isolation for a composite operation is resolved only in the context of one usage. If an operation containing such 'hard-coded' synchronisation instructions is then used as a functional component within a larger composite operation, then progressive, incremental synchronisation steps will still occur during execution of the larger composite operation, resulting in possible atomicity and isolation concerns identified earlier in Section 2.5.3.1.

Thus, schemes which shift the synchronisation responsibility to the caller are unsatisfactory because of the violation of encapsulation, unsuitability to complex & dynamic cases, and the resultant code fragility introduced.

2.5.3.3 Caller vs. Target vs. Mediator

As discussed, it is problematic to assign the synchronisation responsibility to the caller of a composite operation because the initial caller must break the encapsulation of subsequent nested target objects in order to determine the synchronisation requirements. Schemes which rely on violating encapsulation lead to unsafe, tightly-coupled, fragile solutions.

Conversely, it is problematic to assign synchronisation responsibility to the individual target objects within a composite operation because of atomicity, isolation and deadlock concerns. To prevent these problems, a holistic approach is required which unites the synchronisation requirements into a monolithic atomic entity, rather than a piece-meal approach which follows execution through the call chain.

One solution to the problem of synchronisation responsibility in composite operations is to introduce a mediator between the caller and the target to provide the synchronisation mechanism. Such a mediator would need access to the synchronisation requirements of each of the component operations within a composite operation.
Without violating encapsulation, it is possible for each of the component classes to make elements of this information available, so that they can be combined by the mediator to determine the total synchronisation requirements for the root operation. In addition, both the mediator and the synchronisation elements must be compatible with the flexibility offered by the object-oriented approach, such as the use of generalisation and polymorphism.

Such a mediator approach forms the basis for the meta-layer synchronisation scheme described in this thesis.

2.5.4 Providing Atomicity and Isolation for Composite Operations

Concurrency issues such as atomicity and isolation are also prevalent in the areas of computer operating systems and concurrent database transactions. Related research in these fields has shown that an approach which involves progressively alternating resource acquisition and release to be problematic. It is well-known that the locking and unlocking of access to shared resources must be done in a two-phase manner to maintain serialisability [Kum96] [Jan85]. Although the synchronisation of multiple threads in an object-oriented system shares some similarities with approaches to controlling concurrent database transactions, the transactional approach employs conflict detection and resolution strategies which are not applicable to general object-oriented program execution.

2.5.4.1 Concurrency Control Mechanisms in Database Transactions

Concurrency Control Mechanisms (CCMs) are used in database systems to serialise the execution of concurrent transactions. At a high level, CCMs can be classified [Kum96] into two approaches: pessimistic and optimistic.

Pessimistic approaches assume that conflicts among transactions are inevitable and lead to undesirable situations, such as data inconsistencies and deadlock. Pessimistic approaches take preventative measures to avoid interference occurring, and take action against conflicts as they occur.

In contrast, optimistic approaches allow transactions complete freedom of access to data until a final commit point, where checks are performed to detect any conflict or interference.
Two-Phase Locking

Two-Phase Locking [Kum96] is a general locking protocol which can be used to ensure serialisability for concurrent operations. The protocol involves reordering access control within composite operations into two distinct phases: a single acquisition phase and a single release phase. During the acquisition phase, access locks for all the desired objects are acquired. Only after this first phase has ended, and no more access locks will be requested, can the release phase start, in which the access locks held can start to be released. The two-phase locking approach makes the entire locking phase mutually exclusive to the unlocking phase, which guarantees serialisation because any objects accessed within an operation remain locked for the complete duration of use within the operation. This ensures a consistent view of the objects used in an operation.

Two-phase locking protocol prevents access conflict by ensuring serialisability, however it does not provide protection from liveness failures such as starvation or deadlock.

Conflict Resolution

Competing transactions often conflict over access to data items. One of the main roles of a CCM is to resolve these conflicts. In the context of database transactions, conflict resolution entails either blocking the execution of a transaction until a data item becomes free, or alternatively aborting then restarting a transaction. In the case of access deadlock between two transactions, one of the conflicting transactions must be selected and aborted. Aborting or rolling back the transaction involves resetting the state of the data items involved back to their original states before the transaction started.

Database records, the resources database transactions operate on, are static representations of information in some persistent store. It is a simple matter to roll back database operations by reverting the stored information to a copy of an earlier version. In general, database CCMs exploit this ability by employing a reactive theme of conflict detection followed by the selection of a victim transaction for abortion and restart.

Although objects do provide a convenient level of granularity for applying a roll-back mechanism, application software, however, often involves operations with real-world or temporal side effects which cannot be reversed. For example, displaying information on a screen, playing audio through a speaker, or sending messages over a network are
all examples of operations with side-effects which may not be rolled back or reversed. Accordingly, synchronisation schemes for these categories of multithreaded object-oriented programs must focus on a pro-active approach of ensuring conflict prevention, rather than relying on conflict resolution.

### 2.5.4.4 Conflict Prevention

There is a specialised variant of the two-phase locking protocol in which deadlock cannot occur. Static locking or strict two-phase locking [Kum96] is the strongest way of enforcing mutual exclusion between the acquisition and release phases of the protocol. In this scheme, the entire locking phase must precede commencement of the execution phase, which must entirely precede commencement of the unlocking phase. After identification of all of the required resources, the concurrency control mechanism blocks the execution phase of the operation until it can acquire all of the corresponding locks at once. Once all the locks have been acquired, the execution phase can safely run to completion without any synchronisation points, then the locks involved are finally released.

The drawback in this approach is that all of the resources accessed during the operation must be identified before any execution takes place.

### 2.6 A Meta-Level Approach

Existing approaches to synchronisation for composite operations have drawbacks and limitations because of their failure to address the issues outlined above in section 2.5.3. This thesis proposes a meta-level approach that addresses these issues.

#### 2.6.1 Rationale

The underlying source of conflict between object-oriented design goals and the necessities of synchronisation is the inconsistency between their differing fundamental views of the part/whole relationship of entities within a system.

##### 2.6.1.1 O-O : Abstract, Flexible View of the Part/Whole

The object-oriented approach aims for complete encapsulation, where the component classes of a system declare public interfaces, but allow their internal workings to be designed and implemented in isolation to the rest of the system. The O-O themes of encapsulation and generalisation are used to achieve a reduction in the coupling or dependencies between class implementations which in turn allows greater versatility in
the ways in which they can be combined. The use of encapsulation allows systems in which the parts have no explicit knowledge of the whole, and dynamic cases where the parts of the whole may change over time. This modular, flexible and decentralised view of the part/whole relationship operates at many levels:

• At the instance level, objects can be combined through association (aggregation or collaboration), even though they are not designed with explicit knowledge of each other's concrete implementation behaviour.

• At the operation level, inheritance allows operations from various classes within a hierarchy to be combined into one concrete subclass definition.

Elements of such systems (objects in a system, operations in a class) may be composed seamlessly, without additional work to adapt the elements to each other. Further, the private implementations of component behaviour may be modified without requiring redesign of the system as a whole.

2.6.1.2 Synchronisation: Concrete View of the Part/Whole

In contrast however, the mechanics of synchronisation often requires a centralised, concrete view of the part/whole relationship. The aim of synchronisation is to control and coordinate the interaction between concurrent behaviour, and consequently this requires identification and knowledge of the entities involved in the whole. A synchronisation mechanism often requires concrete views of the part/whole relationships between various levels of entities:

• Objects - when providing atomicity for composite operations over multiple objects, a synchronisation mechanism requires concrete identification of the set of target objects involved in each particular operation.

• Operations - when providing safety for intra-object concurrency at the operation level, an access synchronisation mechanism requires knowledge of the synchronisation compatibility of all possible operations on a particular target. In the case of a derived subclass, a synchronisation mechanism must be able to compare the synchronisation details of operations defined by the subclass with those of operations inherited from its various superclasses.

Static synchronisation schemes may be implemented based on particular views of such part/whole relationships that are captured within the component classes of the system. However, such schemes are fragile and violate the O-O themes of encapsulation and generalisation. Further, they limit the flexibility of further
composability and reuse, and may be unsuitable for systems with dynamic part/whole relationships.

A synchronisation mechanism that is to be compatible with the flexibility of the object-oriented approach must not attempt to explicitly encapsulate a complete view of a system's part/whole relationship within a single class definition. Such an approach cannot take into account the context of an element's use within a composite system. Rather, a class should only make available details of the relationships within its immediate context – the complete view must be compiled by some other entity that unifies the relationships to form the larger view of the system.

2.6.1.3 Mechanics of Serialisation

To protect composite multi-object operations from various forms of interference it is necessary to use some form of two-phase locking to achieve serialisation. Since the principle of a rollback mechanism is inappropriate for many types of operations upon objects\(^1\), the strict form of two-phase locking is required. This necessitates the identification of all objects involved in a composite operation, before any actual execution processing is started.

The set of all objects which may potentially be involved in a composite operation depends on the behaviour (and hence implementations) of the suboperations invoked. Hence, identification of this set is a complex and difficult task to achieve without violating the encapsulation of the target objects of subinvocations. Unraveling dynamic levels of associations between polymorphic classes requires a generalised approach. In addition, the complete set of objects is often based on associations stored within the various target objects' private internal state. The process of inspecting a target object's internal state to identify further potential objects must also be synchronised against other concurrent update operations on such objects.

Further, due to the flexibility in composing systems from reusable objects, synchronisation for composite operations must be considered in context. A composite operation provided by some class may be a root invocation from a client task in one context, but be a component subinvocation of a larger composite operation requiring atomicity in another context.

\(^1\)See section 2.5.4.3.
Chapter 2. Synchronisation in Multithreaded Object-Oriented Programs

2.6.1.4 Solution: A Generalised Mediator

One way of resolving this conflict is an approach which allows classes to provide elements of synchronisation meta-information to a generalised mediator, whilst still maintaining complete encapsulation of their private implementation details. The elements of synchronisation meta-information are utilised at runtime by some adaptable generalised coordination scheme, in combination with the state of the objects within the system to produce a holistic view of the nature of object interaction within each composite operation. The generalised coordination scheme subsequently enforces an invocation-time synchronisation policy that is dynamically constructed based on the synchronisation meta-information elements of the actual classes instantiated within the system, and the current state of associations between instances involved in the operation.

2.6.2 Goals

This thesis proposes an adaptable, flexible and reusable approach to synchronisation in multi-threaded object-oriented programs based on the following characteristics:

- safety – automatic enforcement of access synchronisation,
- full compatibility with object-oriented design principles,
- atomicity and isolation execution properties for composite operations,
- liveness – preventing the possibility of starvation and deadlock.

2.6.2.1 Safety

The aim is to largely automate the processes required for a client to organize synchronisation and method invocation. Only when access synchronisation becomes implicit to operation execution can operations be reliably isolated. Automatic enforcement of access synchronisation for operations requires transparent coupling of the access synchronisation mechanism and the operation invocation mechanism. Access synchronisation becomes a precondition to operation execution. Removing explicit synchronisation instructions from operation implementations reduces complexity and allows more flexibility, reusability and maintainability.

2.6.2.2 Compatibility with O-O Principles

It is desirable for the synchronisation approach to retain full compatibility with the basic principles of object-oriented design: encapsulation, generalisation and inheritance. This enables software designers to realise the benefits of an object-oriented
approach in managing complexity in the design and maintenance of systems of shared components. In particular, the ability to successfully encapsulate and reuse component classes, and the use of polymorphism and dynamic binding to achieve generalisation from the caller's perspective, are both key techniques for managing software complexity.

2.6.2.3 Composite Operations

The scheme should work transparently with the principle of functional composition, such as aggregation or collaboration between objects. The execution qualities of atomicity and isolation protecting simple operations individually must be implicitly extended to encompass composite operations as a whole.

2.6.2.4 Liveness

An automated scheme for managing the execution of concurrent operations removes scheduling control from the participants. Such a scheme must provide fair and reliable scheduling qualities. It must also be free of undesirable liveness failures such as the starvation of operation requests or access deadlock between concurrent operations.

2.7 Related Work

This section discusses some existing work that relates to the ideas presented in this thesis:

- Active Object Design Pattern
- Structured Transactions
- SCOOP extensions to Eiffel.
- SOS Paradigm

2.7.1 Active Object

Lavender and Schmidt [LS96] describe the Active Object pattern which decouples method execution from method invocation in order to simplify synchronised access to an object by independent threads. The Active Object pattern describes a way to implement the 'related' animation model of the Actor model [Hew77] [Agh86], on top of a concurrent object-oriented environment based on a passive or unrelated animation model.
The Active Object pattern shares some similar concepts with those of the scheme proposed in this thesis:

- the reification of operation requests into Method\textsuperscript{2} objects.
- operation invocation via the client interface implicitly triggers the construction and queuing of a Method object.
- a prioritised activation queue for storing the Method objects of pending method invocations.
- a scheduler meta-object for managing the activation queue and controlling method execution.

However, unlike the focus of the scheme proposed in this paper, the Active Object pattern does not consider composite operations which span multiple active objects. Rather, it examines the scheduling aspects of operations involving a single target object. Hence, the core issues of this thesis – atomicity, isolation and deadlock in composite operations – are not addressed.

2.7.2 Structured Transactions

Lea [Lea97] describes transaction-based techniques that can be applied in general purpose concurrent programming contexts to achieve interference-free execution for multi-object operations. The approach is based on a passive object model with no explicit invocation control.

2.7.2.1 Standardised Transaction Protocol

In the structured transaction approach, each class supports a standardized transaction protocol which propagates control through the objects successively involved in each composite operation. In addition to normal arguments, each method requires a transaction control argument (for example, a unique transaction key). Each participant method is then responsible for using this key to manage and isolate actions in accordance with a given policy. Participants give up their local autonomy and instead rely on the transaction control mechanism to tell them when to perform actions and/or commit to their effects.

\textsuperscript{2}The Active Object pattern [LS96] refers to operation requests as 'Method' objects. The scheme described in this thesis refers to operation requests as 'Invocation' objects, while the operations themselves are defined as 'Method' objects.
Chapter 2. Synchronisation in Multithreaded Object-Oriented Programs

2.7.2.2 Transaction Control Policy

At a high level, there are two categories of transaction control policy: optimistic and pessimistic.

In optimistic schemes, each sub-operation attempts to perform the activities associated with its part of the larger composite action, but does not commit permanently to these effects until all participants have each confirmed their success. If interference is detected, then each sub-operation is required to perform a rollback to its initial state.

Alternatively, in pessimistic schemes, sub-operations employ some form of locking to guarantee transaction safety as control progresses through the composite operation. This avoids interference, but can result in deadlock. Recovery from deadlock requires another mechanism to detect such situations, in addition to a roll back mechanism for each component.

The approach provides atomicity and isolation for arbitrary composite operations by uniting the component suboperations together via the transaction control mechanism. However, because it does not employ strict two-phase locking, it requires a rollback mechanism when interference (in the optimistic case) or deadlock (in the pessimistic case) is detected.

In addition, the approach impacts the method signatures and complicates the implementations of all of the classes involved – as each operation must be designed to support the transaction mechanism.

2.7.3 SCOOP

SCOOP ("Simple Concurrent Object-Oriented Programming") is an alternate approach to concurrency [Mey97] that can provide atomicity for composite operations on multiple objects. The SCOOP approach represents minimal extensions to the sequential object-oriented language Eiffel [Mey88], via a single notational extension, some validity constraints, and additional call semantics. The scheme enforces synchronisation at the invocation level, obviating the usage of lower level synchronisation primitives. However, the scheme suffers from an overly simplistic approach which suffers from the same issues for composite operations detailed in section 2.5.3.
Chapter 2. Synchronisation in Multithreaded Object-Oriented Programs

2.7.3.1 Animation Model

The SCOOP approach is based on a constrained animation model where separate 'processors' (generalised processes/autonomous threads of control) are dedicated to 'handling' the sequential execution of instructions on one or more objects. At any point, each object within a system belongs exclusively to only one processor/handler. Hence, safety is guaranteed because each processor has exclusive access to the objects it operates upon. However, this constraint precludes intra-object concurrency and removes the possibility of readers/writer style concurrency within a single object or upon a set of objects handled by the one processor.

2.7.3.2 Dual Call Semantics

The invocation mechanism used in the SCOOP approach has dual semantics:

- If both the client and target objects reside on the same processor then the call is synchronous.
- If the client and target objects reside on different processors then the call is asynchronous, and said to be a 'separate' call.

2.7.3.3 'Separate' Concept and Validity Rules

The SCOOP approach requires that the software text indicates whether or not entities belong to the same processor/handler. It involves the use of the separate keyword to identify fields (parameters, variables, etc.) that refer to objects that belong to another processor.

In addition, there are four validity rules (Separate Consistency Rules) governing the use of separate objects which enforce the consistency of fields which contain object references. They ensure that through assignments, function calls, return values, and expanded types, a reference to a separate object can never be assigned to a field which itself is not declared separate.

The consistency rules increase coupling between individual components of a system because the components must be designed and coded in terms of their relative handlers or processors. This places constraints on the structure and granularity of concurrency within an application, as well as detracting from the flexibility and reusability of components, and the ability of clients to generalise in their use of components.

40
2.7.3.4 Separate Call Rule and Object Reservation Mechanism

The approach introduces another rule (Separate Call Rule) which is used both as an object reservation mechanism and to provide atomicity over sequences of calls to separate objects:

- The target of a separate call must be a formal argument of the routine in which the call appears.

This rule forms the basis of the synchronisation mechanism of the approach, as the runtime system blocks calls containing separate formal arguments, until the objects attached to those arguments are available i.e. reserved for use by the calling processor. Although the scheme assigns the role of acquiring locks on the resources to an implicit built-in runtime mechanism, it forces identification of the objects to be reserved on the caller of the operation, as in section 2.5.3.2.

This requires components to expose their implementations which breaks the principle of encapsulation. For example, an operation Foo on component A may involve private delegation to a separate component B. But the separate call rule requires that B must be a formal argument to operation Foo, exposing details of its private implementation. Further, in dynamic cases, Foo may involve delegation to a dynamic number of separate components – which may be difficult to design with a static argument list. Clearly, using an operation’s formal argument list as the basis of an object reservation mechanism is overly simplistic.

One way of hiding such details from callers is to define a wrapper around such operations involving separate calls. Callers can then invoke the wrapper method which in turn invokes the real operation with formal parameters according to the separate call rule. However, as discussed in Section 2.5.3.1, such a target-based approach often proves unsatisfactory in composite situations because the atomicity achieved is provided only over suboperations individually – the object reservation mechanism does not extend over the initial root operation’s entire call chain. In addition, the incremental and progressive nature of synchronisation points is prone to deadlock opportunities.

2.7.4 SOS Paradigm

McHale [McH94] discusses aspects of synchronisation in concurrent, object-oriented languages, such as expressive power, unsafe access to instance variables, genericity and inheritance.
2.7.4.1 Guard-Based Approach

The Service-Object Synchronisation (SOS) paradigm is a guard-based approach which uses synchronisation counters and scheduling predicates. The expressive power offered by the SOS paradigm allows synchronisation mechanisms to easily implement a wide range of particular access and condition synchronisation policies, at the operation level. The approach also proposes the provision of language support for generic synchronisation policies (Mutex, Readers/Writer, Bounded Buffer) so that such policies may be reused by a number of different classes.

However, the guard-based nature of this style of synchronisation approach focuses on the initial target of an operation, and consequently is often unsuitable for providing atomicity over composite operations\(^3\). In contrast, this thesis focuses on providing atomicity requirements for such operations in a way that retains the flexibility and modularity of the object-oriented approach.

2.7.4.2 Unsafe Access to Instance Variables

The synchronisation instructions related to an object often need access to the internal state or instance variables of an object in order to implement its synchronisation policy. Many synchronisation schemes allow such access but do not provide any means to ensure that this access is performed in a safe manner. McHale argues that a synchronisation mechanism can explicitly maintain its own copies of such variables and in doing so can implement synchronisation policies without having to access instance variables.

The synchronisation scheme proposed in this thesis does access the values contained in certain types of shared object instance variables. However, the synchronisation instructions which involve such accesses are executed exclusively by a special metalayer which controls all access to such shared objects. The scheme uses a pervasive readers/writer access policy and blocks all such accesses to internal state until it is safe to do so. Hence, safe access to instance variables by synchronisation instructions is ensured.

---

\(^3\)See section 2.5.3.1.
Chapter 3

A Meta-Level Scheduler

This chapter describes the workings of a meta-level approach to synchronisation in multithreaded object-oriented programs. Section 3.1 describes the intent of the synchronisation scheme. Section 3.2 provides an overview of the components of the scheme. Section 3.3 describes changes required to the traditional object model for classes of objects to be shared. Section 3.4 details the new invocation mechanism for operations on shared objects. Section 3.5 describes the process of access decomposition. Section 3.6 describes the operation of the general invocation scheduling mechanism. Section 3.7 discusses how condition synchronisation is achieved using the scheme. Section 3.8 revisits invocation scheduling in the context of conditional invocations. Section 3.9 discusses the benefits and costs involved in the use of the scheme.

3.1 Introduction

This synchronisation scheme introduces a high-level abstraction that implicitly provides automatic atomic access synchronisation for all operations on shared objects. The scheme decouples method execution from method invocation by passing control of the evaluation, scheduling and execution of invocations on shared objects to a generic synchronisation meta-layer. The scheme provides atomicity and isolation for both simple and composite operations, through the introduction of a dynamic pro-active pre-execution analysis stage and the subsequent use of a strict two-phase locking protocol to protect and isolate operation execution.

3.2 Overview

The synchronisation scheme operates in a heterogeneous object environment, in which objects can be classified into two forms:
Chapter 3. A Meta-Level Scheduler

- 'normal' objects
- 'shared' objects

A heterogeneous environment allows some components of a system to be designed and built within the context of traditional sequential object models. Normal objects are defined and their methods are invoked via the standard mechanisms of the underlying language. Shared objects require a special form and all interaction upon them is controlled by the meta-level synchronisation scheme.

Figure 3.1 General program architecture showing heterogeneous object model.
Figure 3.1 shows the general program architecture of the approach, involving a number of independently threaded tasks interacting with a set of common shared objects. Only those parts of a system that involve concurrent access to shared objects require adaptation to the new synchronisation scheme. Operations on 'shared' objects can only access their target's private internal state, thread-safe built-in data types and references to other 'shared' objects.

At the high level, the synchronisation scheme for shared objects is based on the following novel concepts:

- a new way of defining the member operations of shared object classes
- a new invocation mechanism for requests for operations on shared objects
- a generic supporting synchronisation meta-layer architecture containing
  * a general object access decomposition system
  * a general invocation scheduling system

Figure 3.2 illustrates the interaction between the four major components embodying the high level concepts of the scheme:

- Shared Object Model
- Invocation Mechanism
- Access Decomposition Mechanism
- General Invocation Scheduling Mechanism

A general synchronisation system is created as a meta-layer in the runtime architecture. This system is responsible for coordinating the concurrent execution of all
operations on shared objects. When independently threaded tasks invoke operations on shared objects, control of each request is handed to the meta-layer for evaluation, scheduling and execution. The synchronisation system takes a pro-active approach to conflict prevention through the introduction of an access decomposition stage which determines an access table. An access table holds the complete set of shared objects that may potentially be accessed within the execution of a particular operation request. This set of objects, which is determined before the actual operation execution stage, corresponds to a set of access locks that a general invocation scheduler uses, in accordance with a strict two-phase locking protocol, to guarantee atomicity and isolation for all operations on shared objects.

Shared objects must be instances of classes compatible with the meta-level scheduling system; the process of access decomposition requires the inspection of "meta-information" about shared object operations. In these classes, operations (member functions, 'methods') are bundled together with this descriptive meta-information, allowing the general access decomposition mechanism to reason about the potential shared object interaction involved in individual invocations. Further, to ensure safety, the operations of shared objects must be defined in such a way that necessarily reroutes all requests via the new invocation mechanism – preventing errant or devious circumvention of the synchronisation mechanism.

When a thread of control requests execution of an operation on a shared object, the usual method call mechanism is replaced with a more elaborate mechanism that provides the synchronisation meta-layer with control over the evaluation, scheduling and execution of the request. The first step in this new mechanism involves reification of the request – an "Invocation" object is created, its fields identify the target object and operation involved along with the parameters for the actual method call. Invocation objects are automatically passed to the synchronisation meta-layer for processing. The meta-layer's access decomposition mechanism then analyses the meta-information for the operation as provided by the target object referred by the Invocation object. This information is used both to identify the mode of access (read/non-exclusive or write/exclusive) required on an object, and in the case of composite operations, to explore potential call chains that might result from nested suboperations, in order to determine an access table for the invocation. After this evaluation stage, the invocation's progress is coordinated by the general invocation scheduling mechanism, which controls fair scheduling and then safe atomic execution of the operation. In accordance with a strict two-phase locking policy, the scheduler blocks an operation request until it can acquire simultaneously all of the appropriate
access locks for the complete set of objects involved. After this step, the complete operation can be performed atomically without requiring further access synchronisation for any nested actions. When the call chain returns, the scheduler releases the locks used and returns control and return value back to the calling thread.

Figure 3.3 illustrates the execution progress of multiple concurrent invocations as they move through the various stages of the synchronisation meta-layer.

### 3.3 Shared Object Model

Objects, that are to be shared, must be instances of classes designed to work with the meta-layer synchronisation scheme. Figure 3.4 provides an illustration of the *shared object model* as it relates to the other high level concepts of the scheme.
Diagrams illustrating the class relationships between various low-level aspects of the scheme are provided in Section A.2.

3.3.1 Uniformity of Access

In order to guarantee protection from interference, all access to shared objects must be controlled by the synchronisation scheme. Hence, shared object classes cannot permit any public access to data members, nor can they define any conventional publicly accessible member functions which would allow a program to circumvent the synchronisation scheme. Instead, all publicly accessible functions of a class are defined using "Method" objects.

3.3.2 Methods as Objects

A Method object combines the traditional aspects of an operation (name, argument list, return type, body of implementation) with an additional meta-information declaration function which describes object access within the implementation. Each operation defined for a shared object is defined as a specialised Method object within the parent shared object class definition. An implementation of the scheme provides a Method base class, with some built-in functionality, from which all operations are derived from. Figure 3.5 shows a comparison between the normal object model and the shared object model.

The architectural shift from defining an operation directly as a member function of its owner's class definition, to defining it as a function within a Method object, is required for the approach's advanced manipulation and control of operations. Firstly, it enables a direct association between an operation implementation and its corresponding meta-information declaration; they are both encapsulated within the
same object. Additionally, the implementation functions of Method objects are not directly accessible and so the approach provides protection against errant or devious circumvention of the synchronisation scheme. The Method base class also provides a mechanism for invocation reification, part of the shared object invocation mechanism, described in Section 3.4.

![Comparison of Normal and Shared Object Models](image)

Figure 3.5 Comparison of Normal and Shared Object Models.

### 3.3.3 Operation Meta-Information Declarations

Each Method operation provided by a shared object requires a function which declares *meta-information* about the operation's implementation. The meta-layer uses this function at initialisation time to create the meta-information for each class of Method. Figure 3.6 shows the structure of such meta-information declarations. The functions consist of two main components: target access mode and SharedObject interaction details. The *target access mode* component describes the mode of access the operation requires on a target instance. The *shared object interaction* component describes aspects of the operation that must be considered by the access decomposition mechanism (see Section 3.5, below) to determine, prior to actual execution, all the objects potentially accessed within a composite operation. The description of SharedObject interaction consists of two components: declaration of the SharedObject reference fields involved, and the actions potentially executed upon or involving those reference fields.
Chapter 3. A Meta-Level Scheduler

3.3.3.1 Target Access Mode

Each meta-information declaration function must declare the mode of access its corresponding operation implementation requires with respect to the internal state of its target instance. Figure 3.7 illustrates the Method object representations of both a read/non-exclusive and a write/exclusive operation.

3.3.3.2 Shared Object Interaction

In the synchronisation of composite operations, the access decomposition mechanism must identify all of the potential objects involved and the corresponding modes of access. As shared objects can only be accessed through operation invocation, this requires identifying the details of all potential invocations upon shared objects throughout all levels of the root operation's call chain. The access decomposition mechanism achieves this by using method meta-information to dynamically construct a
model of the potential call chains resulting from the root operation. To this end, the method meta-information function for each Method operation describes details of:

- potential subinvocations (identifying target field, operation name, shared object parameter variables)
- creation of new SharedObject instances
- assignments of values to those member fields of the target shared object that reference other shared objects
- references to other shared objects returned as results of calls

The declaration of this information consists of two parts:

- **SharedObject Reference Field Declarations** – any fields used within the operation's implementation (parameters, member fields, local new instances, results of subinvocations) which hold references to SharedObjects are declared.

- **SharedObject Action Declarations** – the potential actions (subinvocations, member assignments, returned fields) upon or involving SharedObject reference fields are declared.

### 3.3.3.3 Shared Object Reference Field Declarations

In order to describe and reason about the flow of potential values of references to shared objects within call chains, it is necessary to introduce an abstraction to represent those fields which store such values. Figure 3.8 shows the abstraction SharedObjRefFld which has four concrete subclasses: SharedObjParameter, SharedObjMemberFld, SharedObjSubResult and SharedObjNewInstance.

![Class Relationship Diagram for SharedObjRefFlds](image)

Instances of these concrete classes are used in method meta-information action declarations to describe the use and flow of references to shared objects within an operation's implementation:

---

51
- **SharedObjParameter** – This abstraction is used in describing the usage of a formal parameter to the operation which represents a reference to another SharedObject.

- **SharedObjMemberFld** – This abstraction is used in describing the usage of a member field of the target object which represents a reference to another SharedObject.

- **SharedObjSubResult** – This abstraction is used in describing the usage of the result of a subinvocation which returns a reference to another SharedObject.

- **SharedObjNewInstance** – This abstraction is used in describing the usage of a new concrete SharedObject instance which may be instantiated within this operation.

The SharedObjRefFld abstraction is not concerned with any fields which hold primitive values or references to 'normal' objects – these types of fields do not enter the consideration of the meta-layer synchronisation scheme. Hence interaction involving any objects (referred to in parameters, member fields, subinvocation results, or new instance fields) which are not descendants of the SharedObject base class are not coordinated explicitly by the meta-layer.

Method meta-information functions use field declaration functions, shown in Table 3.1, as required, to declare the various SharedObject reference fields used in the operation implementation. The functions return various concrete instances of SharedObjRefFld objects which may then be used in action declaration functions.
### Field Declaration Function

<table>
<thead>
<tr>
<th>Function</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>parameter(p)</td>
<td>returns a SharedObjParameter object associated with the formal parameter variable of name p in this Method's implementation function</td>
</tr>
<tr>
<td>.memberFld(m)</td>
<td>returns a SharedObjMemberFld object associated with the member field of name m of the target object of this Method operation</td>
</tr>
<tr>
<td>newInstance(c)</td>
<td>returns a SharedObjNewInstance object which represents a new SharedObject instance of class c that may be instantiated by this Method's implementation function.</td>
</tr>
<tr>
<td>self()</td>
<td>returns a special case of SharedObjMemberFld object which is associated with the target of the Method operation itself</td>
</tr>
<tr>
<td>subInvocationResult(c,sid)</td>
<td>returns a SharedObjSubResult instance associated with the result of the subinvocation described by the SubInvocationDescriptor(^1) sid of class type c</td>
</tr>
</tbody>
</table>

Table 3.1 Creating SharedObject reference fields for use in MethodMetaInformation Declarations

### 3.3.3.4 SharedObject Action Declarations

Method meta-information declarations use *action declaration functions*, shown in Table 3.2, as required, to describe shared object interactions involving the SharedObject reference fields declared.

---

\(^1\)SubInvocationDescriptor objects describe subinvocations and are created via Shared Object Action Declarations. See Section 3.3.3.4, next.
Chapter 3. A Meta-Level Scheduler

<table>
<thead>
<tr>
<th>Action Declaration Function</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>subInvocation(t,o)</td>
<td>declares a potential subinvocation of the operation ( o ) on target ( t ).</td>
</tr>
<tr>
<td></td>
<td>( o ) is a string holding the name of the operation</td>
</tr>
<tr>
<td>subInvocation(t,o,(p1,...))</td>
<td>declares a potential subinvocation of the operation ( o ) on target ( t ).</td>
</tr>
<tr>
<td></td>
<td>In addition (p1,...) are SharedObjRefFld objects which represent fields referencing shared objects passed as parameters to the subinvocation.</td>
</tr>
<tr>
<td>sid = subInvocation(t,o)</td>
<td>as above, but for the case where the invocation ( o ) on ( t ) may return a reference to a shared object. ( sid ) is a SubInvocatonDescriptor which can be used to create a SharedObjSubResult object for use in further declarations which describe further interaction based on this result.</td>
</tr>
<tr>
<td>assignMember(source, dest)</td>
<td>declares the potential assignment of SharedObjRefFld ( dest ) to the member field represented by SharedObjMemberFld ( source ).</td>
</tr>
<tr>
<td>returnSharedObject(f)</td>
<td>declares the potential use of SharedObjRefFld ( f ) as a return value to the caller of this operation.</td>
</tr>
</tbody>
</table>

Table 3.2 Declaring Shared Object Interaction.

For example, in a simulation of the Dining Philosophers problem, we may envisage a Philosopher class 'Eat' operation which invokes suboperations on associated left and right Chopstick objects.

```java
void eat()
{
    leftChopstick.pickup(this);
    rightChopstick.pickup(this);

    // simulate eating by sleeping for a while
    ...

    leftChopstick.putDown();
    rightChopstick.putDown();
}
```

54
Chapter 3. A Meta-Level Scheduler

The equivalent Method object representation of this operation may be implemented as

```java
class Eat extends Method {
    void meta()
    {
        targetAccessMode(write);
        SharedObjMemberFld left = memberFld(leftChopstick);
        SharedObjMemberFld right = memberFld(rightChopstick);
        subinvocation(left, pickup, self());
        subinvocation(right, pickup, self());
        subinvocation(left, putDown);
        subinvocation(right, putDown);
    }

    void impl()
    {
        leftChopstick.pickUp(this);
        rightChopstick.pickUp(this);

        // simulate eating by sleeping for a while
        ...
        leftChopstick.putDown();
        rightChopstick.putDown();
    }
}
```

3.3.4 MethodMetaInformation Objects

At Method initialisation time, the meta-layer processes the meta-information declaration function of each Method class to build the meta-information for each type of operation. This data is stored as a `MethodMetaInformation` object. Each Method object is associated with a corresponding MethodMetaInformation object.

The MethodMetaInformation object is an indexed representation of the details declared by the operation’s meta-information function, and is used by the meta-layer during the access decomposition process. A MethodMetaInformation object encapsulates:

- the target access mode of the operation
- a table of shared object parameter fields (instances of SharedObjParameter)
- a table of shared object member fields accessed (instances of SharedObjMemberFld)
- a table of potential subinvocations (instances of SubInvocationDescriptor)
Chapter 3. A Meta-Level Scheduler

- a table of potential subinvocation results used (instances of \texttt{SharedObjSubResult})
- a table of potential shared object member field assignments
- a table of potential shared object reference fields returned to caller
- a table of dependencies between sources of shared object reference fields and their usage (FieldAction Dependency Index)

An explanation of the use of method meta-information is presented in Section 3.5, below, where the access decomposition mechanism is described in detail.

3.3.5 Collaboration

The meta-layer’s access decomposition mechanism introduces constraints on the allowable forms of collaboration between shared objects and other normal objects within a running application system. The access decomposition mechanism requires meta-information describing the subinvocations and respective target objects potentially involved in each suboperation, in order to determine the complete access requirements of the root invocation.

However, building object-oriented application programs usually involves reusing standard classes from a framework or class library. These classes do not conform to the shared object model prescribed by the meta-layer approach, and hence the access decomposition mechanism cannot identify the access synchronisation requirements for nested operations on these types of objects.

To ensure safety from interference, shared objects may only invoke suboperations on normal objects if both of the following conditions are satisfiable:

- The operation on the target normal object will not subsequently involve any nested requests for operations on shared objects. This ensures that the access decomposition mechanism can ignore the interaction with the normal object but still produce an accurate access table. A correct access table is necessary for the integrity of the meta-layer’s conflict-based scheduling algorithm.

and

- The operation will not block for a synchronisation event related to another thread executing under the scheduling control of the meta-layer. This prevents the external synchronisation event and the meta-layer’s synchronisation scheme from composing a deadlock situation.
Chapter 3. A Meta-Level Scheduler

If the normal object supports concurrent operations, and it meets the previous conditions then operations on it may be freely invoked by shared objects. However, the meta-layer scheduler does not provide any atomicity or isolation over operations invoked in this manner, and concurrent operations may interleave their access to the normal object.

If atomicity and isolation are required over operations on normal objects invoked by shared objects in this manner, then structural containment techniques can be used to achieve these properties. A 'wrapper' shared class can be defined to encapsulate the normal object, and provide atomic and isolated access according to the schemes readers/writer access policy.

3.4 Invocation Mechanism

The synchronisation meta-layer enforces a new invocation mechanism for operations on shared objects. Figure 3.9 provides an illustration of the invocation mechanism as it relates to the other high level concepts of the scheme.

3.4.1 Overview

At a high level, the new invocation mechanism introduces two extra stages to the traditional mechanism. As shown in Figure 3.10, the first stage entails identifying the access requirements of the operation, while the second stage is concerned with scheduling the invocation against other concurrent invocations to ensure access atomicity and isolation.
Chapter 3. A Meta-Level Scheduler

3.4.2 Invocations as Objects

The synchronisation meta-layer uses Invocation objects to explicitly represent the requests for operations on shared objects that it manages. Figure 3.11 shows a simplified class diagram for the Invocation class. Each Invocation object associates the following information:

- the SharedObject which is the target of the invocation
- the Method object which is the requested operation
- the parameters to the requested operation
- the access table of shared objects and access modes determined by the access decomposition mechanism.

Figure 3.11  Invocation Class Relationship Diagram (Simplified)
3.4.3 Invocation Sequence

Figure 3.12 illustrates the major steps involved in the invocation of an operation on a shared object:

1. **Invocation Reification**: a request for an operation on a shared object is invoked via one of its Method objects; this results in the creation of an Invocation object.

2. **Meta-Layer Interception**: the Invocation object registers with the meta-layer scheduling system, the invoking thread executes the meta-layer's control code of steps 3 to 5 before finally performing the desired operation, step 6, and then exiting from the meta-layer, steps 7, 8 and 9.

3. **Access Decomposition**: the meta-layer uses the method meta-information to determine an access table containing the set of potential objects and access modes required. This step may involve suspending or blocking the request temporarily if the shared objects involved in the access decomposition process are scheduled for, or are currently being updated.

4. **Scheduling**: The Invocation is scheduled against other concurrent Invocations to prevent access conflict. This step may also involve blocking the request until scheduled or currently executing invocations have completed.

5. **Logical Access Lock**: The meta-layer logically acquires all of the required access locks on the Invocation's behalf.

6. **Atomic Execution**: The entire operation is performed atomically and in isolation without any further synchronisation points.

7. **Logical Access Unlock**: The meta-layer logically releases the access locks.

8. **Rescheduling**: Competing concurrent threads that are blocked within the meta-layer because of access conflict with this invocations access table are considered for rescheduling.

9. **Return**: Control and return value are returned to the caller.
Figure 3.12 Invocation Mechanism Interaction Diagram

3.4.4 Nested Invocations

A root invocation occurs when an active thread executing within 'normal' objects, external to the shared object space, invokes an operation on a shared object. At this point control is intercepted by the meta-layer. Execution of a root operation may subsequently involve further nested method invocations on shared objects, however it is the level of the root invocation at which the synchronisation scheme works to guarantee atomic and isolated execution.

In the case of any subsequent nested method invocations, requested during execution of an operation which has already been scheduled by the meta-layer, invocation analysis and control are not required. Access locks for all shared objects that could potentially be accessed have already been acquired by the meta-layer during the synchronisation processing of the root invocation. Accordingly, the meta-layer identifies such nested invocations and allows operation execution to proceed immediately without any synchronisation points.
Thus, during the execution of a composite operation involving nested invocations, both the access decomposition phase, and the invocation scheduling phase, are carried out only once – at the root level.

### 3.5 Access Decomposition Mechanism

Access decomposition plays a central role in the synchronisation scheme's proactive approach to access conflict. Figure 3.13 provides an illustration of the *access decomposition mechanism* as it relates to the other high level concepts of the scheme.

Section 3.5.1 provides an overview of the process of access decomposition. Section 3.5.2 introduces the *access table* – the aim or the required output of the mechanism. Section 3.5.3 describes the architecture of the data structures used by the mechanism to model call chains. Section 3.5.4 explains the process of building a call chain model. Section 3.5.5 describes producing an access table from a complete call chain model.

#### 3.5.1 Overview

At runtime, as each invocation enters control of the meta-layer, it is analysed dynamically to determine the set of objects that may be potentially accessed within the operation's entire execution call chain. The aim of this access decomposition process is to determine (before actual execution of the operation) an access table listing the set of objects and corresponding access modes potentially involved in operation execution. The access table is then used as the basis for scheduling the invocation to ensure access atomicity and isolation during execution of the operation. The mechanism used in the access decomposition process requires the introduction to the shared object model of implementation-specific method meta-information per
operation. The generic decomposition mechanism leverages this meta-information to model the potential call chains involved in an invocation.

The mechanism builds a representation of the potential call chain, taking into account all possible combinations of subinvocations and target objects that could potentially result from the initial root invocation. The mechanism simulates execution, building nodes that correspond to the stack frames that may potentially occur during actual execution. Each node represents the invocation(s) of a particular Method operation on a particular shared object. A method's meta-information describes the types of suboperations that it may invoke; however, in some cases, the potential targets of these operations must be determined dynamically. The potential targets are also declared by the method's meta-information, and correspond to various concrete types of SharedObjRefFld\(^2\) (shared object references within member fields of the target object, shared object references passed as parameters, shared object references obtained from the return values of suboperations, or shared objects newly instantiated within the operation). The meta-layer constructs a set which pessimistically represents the potential values for each SharedObjRefFld used. For example, if an operation calls a subinvocation to select an item from a collection, then the scheduler will associate all objects in the collection with the reference representing the result of that selection operation. If the scheduler determines that an operation involves a reference variable that might be changed by a currently executing, or a previously decomposed but not yet executing operation, then the decomposition process is suspended until such previously scheduled activities have completed. Suboperations invoked are explored recursively for each potential target.

### 3.5.2 Aim

The required output of the access decomposition process is an access table, containing a list of access claims describing the locking requirements of all potential nested object interaction within an operation. Each access claim identifies a particular shared object instance and the associated access mode required on it.

The specific access claims required for any particular method invocation are dependent on the nature of object interaction within the operation's implementation code. In dynamic cases, they may depend on parameters of the invocation, the internal state of the target and other shared objects in the system. The access decomposition mechanism must be able to handle this dynamism.

\(^2\)See Section 3.3.3.3.
Chapter 3. A Meta-Level Scheduler

As shown in Figure 3.14 and Table 3.3, for simple operations which do not contain any nested subinvocations, there can only be one access claim: on the target object itself.

![Figure 3.14 Simple Operation "scoreHand()"

<table>
<thead>
<tr>
<th>Shared Object</th>
<th>Access Mode</th>
</tr>
</thead>
<tbody>
<tr>
<td>CardHand &quot;Ben&quot;</td>
<td>read</td>
</tr>
</tbody>
</table>

Table 3.3 Access Table for an Invocation of Simple Operation "scoreHand()".

However, in the case of composite operations, as shown in Figure 3.15 and Table 3.4, access tables contain multiple access claims.

![Figure 3.15 Composite Operation "turnAttack()"

63
Chapter 3. A Meta-Level Scheduler

<table>
<thead>
<tr>
<th>Shared Object</th>
<th>Access Mode</th>
</tr>
</thead>
<tbody>
<tr>
<td>Player &quot;Adalita&quot;</td>
<td>write</td>
</tr>
<tr>
<td>MagicItem &quot;Wand&quot;</td>
<td>read</td>
</tr>
<tr>
<td>Monster &quot;Tethayr&quot;</td>
<td>write</td>
</tr>
<tr>
<td>MapLocation &quot;D4&quot;</td>
<td>read</td>
</tr>
</tbody>
</table>

Table 3.4 Access Table for an Invocation of a Composite Operation.

The scheme's shared object model requires that all methods of shared objects specify via *method meta-information* the mode of access required on the target instance. The only form of interaction allowed on shared objects is through the invocation of a Method object, hence the set of access claims required for a composite operation is derivable from the set of potential Method subinvocations (at all levels) that could possibly result from the root operation. Accordingly, the mechanism firstly determines the complete set of potential subinvocations, then uses the target access mode within each suboperation's respective method meta-information, to build the table of access claims.

### 3.5.3 Call Chain Model

In designing a pessimistic algorithm to determine access claims before actual execution, there is a tradeoff between the degree of pessimism, and the amount of computation and modeling complexity required to produce a more accurate prediction. Although simple pessimistic algorithms may require less processing time (i.e. lock all accessible objects, or lock all objects of class Z), it may come at the expense of a larger access table which claims access locks on shared objects that may never be used, thus needlessly limiting the parallelism, availability and liveness of the system. In contrast, an algorithm which uses an excessively detailed model may prove too computationally expensive for actual use – the time taken to compute the access table to take advantage of any available parallelism may outweigh the time savings of executing the operations in parallel.

The mechanism described here uses a directed graph data structure to represent an operation's potential call chain. The potential call chain is a function of the potential operations to be invoked, and the potential shared objects that are used as invocation...
targets. While the mechanism aims to pessimistically model potential call chains, it does not attempt to predict the result of any computations involving primitive values. The shared object model's method meta-information provides a description of shared object interaction within each operation's implementation in terms of the suboperations that may be invoked, the shared object fields used as targets, and the potential flow of references to shared objects between fields (i.e. parameters, member fields, return results). As opposed to computations involving primitive values, call chain modeling only requires dynamic modeling of the flow of references to shared objects between the scopes of the potential call chain stack frames.

3.5.3.1 Modeling the Flow of Shared Object References Between Method Scopes

At any point in runtime, the potential shared objects that may be used as targets of subinvocations within the scope of a particular Method are limited to a finite set. The values in this set originate from the following sources:

- references to shared objects stored in the target objects member fields
- references to shared objects passed to the operation as parameter variables
- references to shared objects obtained as return values from suboperations
- newly constructed shared objects instantiated by the operation

Potential reference values may be dependent on the actions of other operations within the call chain. In the case of the target objects member fields, these may be modified by other operations on the same instance within the call chain, and so this source of reference values is dynamic and dependent on other operations within the call chain. In the same way, the potential reference values passed to the operation are dependent on the caller(s) of the operation, and the potential values returned to this scope are dependent on the suboperations invoked.

Hence, the process of determining the potential values of references to shared objects within the scope of a particular Method in a call chain requires dynamically modeling its dependencies on other operations in the call chain.

3.5.3.2 CallChainGraph

A CallChainGraph object is used to represent the potential call chain model for a particular invocation. The graph is composed of CallChainNode objects. A CallChainNode object representing the root operation is created initially as the root
Chapter 3. A Meta-Level Scheduler

node of the graph. The mechanism then works to expand the model incrementally via progressive sweeps through the graph, adding and updating nodes, until it has determined that the model has exhausted all possibilities for further potential subinvocations.

The aim of the access decomposition mechanism is to determine an access table – the participant objects and the corresponding modes of access. Accordingly, the potential call chain graph is modelled to the minimum level of detail which suffices to accurately determine this information. Hence, many aspects relating to the progress and paths of execution through the actual call chain do not require representation, or are summarised for simplicity within the model. In particular:

- each node is a summary of all potential calls to a particular operation on a particular instance.
- the graph does not contain any information relating to the order of execution of nodes, or iteration over one or a sequence of nodes.

3.5.3.3 CallChainNode

A CallChainNode is used to model the effects of all possible invocations of a particular Method operation on a particular shared object instance within the potential call chain. Even if an operation on a particular shared object instance may be called multiple times by different places within the call chain, the details of all such calls are summarised and represented by a single CallChainNode object.

A CallChainNode has a one to one relationship with a particular Method object. Each class of Method object in turn has a corresponding MethodMeta Information object which describes the shared object interaction in that Method's implementation. The MethodMeta Information object is used as the template for building each CallChainNode's interaction model, in conjunction with contextual details such as the parameter values and instance member field values.

Thus, a MethodMeta Information instance is a static object which stores meta-information representing a model of the potential interaction inherent in the implementation of a particular Method object. A CallChainNode instance is a more dynamic object which is built from a corresponding MethodMeta Information object, but incorporates contextual information such as parameter values and the target object's member field values to build an interaction model tailored for a particular invocation.
A CallChainNode encapsulates and maintains the following information:

- a reference to the target SharedObject
- a reference to the corresponding Method object
- a reference to the MethodMetaInformation object
- a list of parent CallChainNodes that call this node (caller nodes)
- a list of child CallChainNodes that are called as suboperations (called nodes)
- a table of the potential values of parameter variables to this node which refer to shared objects (parameter values)
- a reference to a table of the potential values of those of the target object’s member fields which refer to other shared objects (member field values)
- a table of the potential values of references to shared objects which may be returned by this node to parent/caller nodes (return values)
- a table of the potential values of references to shared objects which may be returned to this node by child/called nodes (subinvocation results)

The information stored is a summary or the superset of the information related to all potential calls to this particular node. For example, in the case where a Method takes a shared object parameter, and this operation could be called multiple times perhaps by different callers within the call chain with different parameter values, then the parameter values table will contain the superset of all such potential shared object reference values.

A CallChainNode’s responsibility is to use the method meta-information to model the effect that its corresponding operation might have on the potential call chain. This involves:

- creating child CallChainNodes representing potential subinvocations (identifying target, method, shared object parameter variables)
- creating a model for the flow of references to shared objects to and from the scope of this node (parameters to and return values from child nodes, return values to parent)
- updating the model by acting on the dependencies between the sources of input for references to shared objects, and their usage, when new potential values are introduced for such fields.
3.5.3.4 Dependencies Between Call Chain Nodes

Due to the flow of shared object references (as parameters and return results) through the structure of call chains, and the mutability of shared object references within shared object member fields, the interaction models built by nodes may be highly dependent on each other. During building of the call chain graph, the construction of later nodes may introduce additional potential shared object reference values for previously constructed nodes. The previously constructed nodes will require revision processing. These dependencies are illustrated in Figure 3.16 and may occur in several forms:

- **Shared Object Parameter Variables** – the interaction model constructed by a node may be dependent on the values of shared object references passed as parameters to the operation. This introduces a dependency on all of the caller nodes which call this node.

- **Shared Object Member Fields** – the interaction model constructed by a node may be dependent on the values of shared object references contained in member fields of the target of the operation. The call chain model is a summarisation which does not incorporate the order of node execution and so this introduces a dependency on all those nodes representing operations on the same target instance which may assign new potential shared object references to those member fields.

- **Shared Object Subinvocation Results** – the interaction model constructed by a node may be dependent on the values of shared object references returned from subinvocations called by the operation. This introduces a dependency on all of these child nodes which return potential shared object reference values to this node.
A node revision process is required as a result of the dependencies on shared object references provided by other nodes. Any node can be dependent on the input of a number of other nodes for a number of different dependent inputs. When revising the model for a node the scheme must determine which inputs have actually changed, examine all of its value tables, and determine which parts of the model need revising. In addition, new nodes representing additional sources of input may be introduced that provide shared object reference values that dependent nodes have already processed/considered. To assist in managing this task, the scheme uses a 'smart'
container abstraction that knows whether a set of potential values has actually changed or not. The 'smart' container can also provide the subset of dirty values that have changed since the last time the container's owner observed the set of values.

The SmartSharedObjectSet abstraction, described in detail in Section B.2, manages a set of potential shared object reference values for a single field. It is used to manage the potential return values of a particular call chain node.

The SmartSharedObjectTable abstraction, described in detail in Section B.3, manages a table of potential shared object reference values for multiple fields. It is used to manage potential shared object reference values of the parameter variables, and potential shared object reference value results of subinvocations, for a particular call chain node.

### 3.5.3.5 Access to Shared Object Member Fields

The modeling process requires access to the member fields of the shared objects involved in the call chain. This raises two issues:

- access synchronisation between invocations concurrently under the control of the meta-layer which may involve interaction with the same shared object instances
- the sharing of potential shared object reference values, assigned to member fields in different operations on the same shared object instance, within the same call chain.

Access synchronisation is required internally within the meta-layer, to ensure that call chain models are not built using reference values within member fields of shared objects that may be modified before actual execution of the operation is scheduled, leading to an incorrect access table result. The access decomposition process for a particular invocation must be temporarily blocked if any member field access required conflicts with the access tables determined for other invocations either executing or being scheduled for execution. In accordance with the rest of the scheme, this access synchronisation is carried out under a readers/writer access policy.

In addition to the synchronisation requirement of member field access, a mechanism is also required for sharing and managing multiple changes to the potential values of individual member fields of shared objects, across multiple nodes within the same call chain graph.

The MemberFieldValueTable abstraction, described in detail in Section B.4, manages these requirements.
If required, a MemberFieldValueTable object may be created and associated with each SharedObject instance. Each call chain node that uses member field references collaborates with the corresponding MemberFieldValueTable instance, both for potential values of member fields, and for synchronising access to these values. During CallChainNode initialisation and revision, the node checks with the MemberFieldValueTable object to determine whether access requests are granted or denied, and to insert and retrieve potential member field values.

3.5.4 Graph Construction Process

The process of building the call chain graph starts with the initialisation of a root node representing the root Method operation on the initial target object. Using this node as the root, an attempt is made to build an exhaustive potential call chain model. If the root Method involves nested interaction, then further nodes must be constructed to represent the suboperations. A recursive approach to graph construction is unsuitable because of the possibly cyclic nature of dependencies between call chain nodes involving shared object references (parameters, member fields, suboperation results). Instead, the graph nodes are processed individually, and use a dirty flag to mark dependent nodes that require revision. Revision of dependencies is managed via iterative sweeps through the graph nodes. When the model is complete the call chain graph is finalised, and an access table can be determined.

3.5.4.1 Attempting Call Chain Finalisation

An attempt at callchain finalisation involves repeated sweeps through the call chain graph, processing each dirty call chain node individually in turn, until there are no more dirty nodes. At any point, an individual node can be classified into one of three states:

- **Clean** – the node has modeled all of its potential effect on the call chain.
- **Dirty** – the node currently requires processing (either initialisation or revision). A node is considered dirty if it has not been initialised, or if any of the potential values of shared object reference fields its model depends on (parameters, member fields, or suboperation results) have changed.
- **Blocked** – the node is denied access to its target instance's member fields, due to access conflict with another concurrent invocation within the meta-layer. This node's initialisation or revision processing can not be allowed to proceed until member field access is granted.
When there are no more dirty nodes left in the call chain graph, the meta-layer checks to see if any nodes are blocked.

The attempt at building the call chain graph returns failure if any nodes are blocked – the invocation was denied access to the member field values of some target shared object, hence the call chain model is incomplete, and so the invocation must be blocked. At some later point, another invocation will release access to the shared object of conflict, this invocation will be notified and may reattempt call chain finalisation.

If there are no blocked nodes then all potential interaction has been modeled. The attempt at call chain finalisation returns success, and an access table may be determined for the invocation based on the finalised call chain graph.

Each CallChainGraph maintains a list of all nodes which are flagged as *dirty*, as well as a list of all nodes which are flagged as *blocked*. The process of attempting call chain finalisation involves the steps shown in the following pseudocode:

```plaintext
while (dirty list not empty) {
  /* sweep graph */
  copy dirty list to sweep list
  reset dirty list to empty
  for each node in sweep list
    process node (initialisation or revision)
}
if (blocked list not empty)
  finalisation = failure
else
  finalisation = success
```

At each sweep of the graph, all of the currently dirty nodes are processed in turn. This may result in the introduction of new reference values to other existing nodes causing the affected nodes to become dirty and require revision as well.

Individual call chain nodes require two types of processing:

- *Initialisation* – in which the node uses its Method’s corresponding MethodMetaInformation to create and initialise the internal value tables required to model the shared object interaction associated with the operation it
Chapter 3. A Meta-Level Scheduler

represents. Additional child call chain nodes may be created to represent any required subinvocations.

- Revision – in which a dirty node acts on any changes made to its value tables as a result of the processing of other nodes within the graph. Additional child call chain nodes may be created to represent any new subinvocations.

To eliminate cyclic recursion, the processing for each individual node is performed in a manner that does not require any nested node processing (initialisation or revision) to occur. Each node performs its own model initialisation or revision processing completely, without interruption by nested processing. For example, if during the processing of some node, a new node must be created to represent a subinvocation, then a new CallChainNode instance is constructed, and flagged as dirty, but the new node’s initialisation processing is not performed yet. Its initialisation processing is performed after the current nodes processing has finished, possibly in the next sweep of dirty nodes. In the same way, if additional potential values for shared object references are introduced which must be passed, returned or assigned to fields accessible in the scope of other nodes, then the affected nodes are marked as dirty and are revised once the current node has finished processing. The sweeping approach ensures that the cyclic dependencies between call chain nodes do not result in infinite recursion.

3.5.4.2 Node Initialisation

A CallChainNode is initialised based on the interaction details stored in its associated MethodMetaInformation object. The MethodMetaInformation object describes the various details of the operation implementation. The sequence of steps for initialising a new node are:

1) **Ensure Member Field Access** – if the operation requires read or write access to the member fields of the target, then the node requests the corresponding access mode from the target’s MemberFieldValueTable. If the operation involves assigning new values to member shared object reference fields then write access is required on the table. If the operation uses member shared object reference fields in any other ways then read access is required on the table. The MemberFieldValueTable checks the access request against those of other invocations currently within the meta-layer according to a readers/writer policy. If access is granted, the node notes that the current values for the required member fields are marked as observed. If access is not granted at this
time, the node initialisation process is abandoned and the node is marked as blocked waiting for access.

2) **Build Parameter Value Table** – if the operation accepts shared object reference parameters, then a SmartSharedObjectTable is created and filled with all potential values for the shared object parameters to this node.

3) **Build Subresult Value Table** – if the operation uses shared object reference values returned from subinvocations, then a SmartSharedObjectTable is created to store all potential values of such fields for use in this node.

4) **Build Subinvocations** – if the operation involves subinvocations, then this node must be linked to corresponding subnodes within the call chain graph. Potential values of targets and parameters are taken from the MemberFieldValueTable, the Parameter Table and the SubResult Table as required. If corresponding subnodes already exist, then the parameters of the subinvocation are added to the Parameter Tables of these existing subnodes. If these ParameterTables become dirty then those subnodes are flagged as dirty. If the corresponding subnodes do not exist yet and need to be created, then CallChainNode objects are created, informed of potential parameter values, and flagged as dirty. Dirty subnodes are then initialised/revised in the next sweep of the graph.

5) **Process Member Assignments** – if the operation involves assigning new shared object reference values to the target's member fields, then the destination field in the MemberFieldValueTable is updated with the appropriate values according to the potential values of the source field of the assignment. If the MemberFieldValueTable becomes dirty then it determines any nodes which depend on its values and flags those nodes as dirty.

6) **Build Return Value Set** – if the operation returns a shared object reference value, then the node creates a SmartSharedObjectSet to hold all such potential return values. These values are then inserted into the SubResult Tables of all parent/caller nodes which call this node. If their SubResult Tables become dirty then those caller nodes are flagged as dirty.

### 3.5.4.3 Node Revision

The node revision process maintains the interaction model by examining which inputs to the dirty node have changed since their initialisation or last revision, and acting on the dependencies for those fields described by the MethodMetaInformation. The additional potential values for dirty parameter variables, member fields, or subresult
values may have to be taken into account. The MethodMetaInformation object provides a lookup table (Field Action Dependency Index) of the dependencies of all shared object reference fields used within the operation. A field action dependency describes one of the following potential actions for the value in the field:

- it is assigned to a member field of the operation's target object
- it is the target of a subinvocation
- it is passed as a parameter to a subinvocation
- it is used as the return value of the operation

The sequence of steps for revising a dirty node are:

1) **Ensure Member Field Access** – if the node requires read or write access to the member fields of the target, the node calls the MemberFieldValueTable to check whether access is still permitted. If access is denied, revision is aborted and the node is flagged as \textit{blocked} waiting for access. If access was previously allowed but is not permitted at this time, then the complete graph is invalidated and graph construction restarted to remove the effect on dependent nodes. (This invalidation effect may occur if another invocation with earlier invocation ordering priority was previously blocked in the graph construction process and now has been unblocked to continue its graph construction. Further progress in its graph construction may reveal a conflict which will invalidate the graph of the later conflicting invocation).

2) **Member Field Revision Action** – if the MemberFieldValueTable is dirty then the Field Action Dependency Index is examined to determine if any of those dirty member fields have dependent actions. If so, then those field action dependencies must be revised, based on the additional reference values for the member field. The node then marks the observable values of the MemberFieldValueTable as clean.

3) **Parameter Revision Action** – if the Parameter Table is dirty then the Field Action Dependency Index is examined to determine if any of those dirty parameter fields have dependent actions. If so, then those field action dependencies must be revised, based on the additional reference values for the parameter field. The node then marks the Parameter Table as clean.

4) **SubResult Revision Action** – if the SubResult Table is dirty then the Field Action Dependency Index is examined to determine if any of those dirty
subresult fields have dependent actions. If so, then those field action dependencies must be revised, based on the additional reference values for the subresult field. The node then marks the SubResult Table as clean.

5) **Cleanup** – The node itself is marked as clean.

Revising a field dependency is based on the same process involved in initialising a node. Depending on the type of action described by the dependency, one of the following actions must be carried out:

- **Member Field Assignment** – if the dirty field is assigned to a shared object member field, then the MemberFieldValueTable is updated with the additional reference values. Due to the pessimistic approach of the mechanism, none of the existing potential values for the member field are removed, the new value is simply added and considered as another potential value. If the MemberFieldValueTable becomes dirty then it itself will find any other nodes which depend on these values and flag those nodes as dirty to be revised in the next sweep of the graph.

- **Subinvocation Target** – if the dirty field is a potential subinvocation target, then this node must be linked to subnodes corresponding to the additional potential field reference values. If corresponding subnodes already exist, then any additional parameter values for the subinvocation are added to the Parameter Tables of the subnodes. If the corresponding subnodes have not yet been created, then CallChainNode objects are created, informed of potential parameter values and flagged as dirty. Dirty subnodes are then initialised/revised in the next sweep of the graph.

- **Subinvocation Parameter** – if the dirty field is used as a parameter in a potential subinvocation, then the dirty values are inserted into the Parameter Table of the corresponding subnodes. If the ParameterTable of the subnode becomes dirty then the subnode is flagged as dirty.

- **Return Value** – if the dirty field is used as a return value for the operation, then the dirty values are inserted into the SmartSharedObjectSet representing this node's potential return values. Any dirty values are then inserted into the SubResult Tables of all parent/caller nodes which call this node. If in turn their SubResult Tables become dirty then those parent/caller nodes are flagged as dirty.
3.5.5 Access Table Production

Each node within the call chain graph corresponds to a particular Method operation on a particular target shared object instance. Each Method operation has an associated MethodMetaInformation object which declares the operation's target access mode.

To produce the access table, the access decomposition mechanism simply enumerates through each node in the finalised call chain graph creating an access claim associating each call chain node's target instance with its required access mode.

The access table is produced as a superset of the access claims per shared object target within the call chain graph. For example, if some nodes require read access to a particular instance, and other nodes require write access to the same instance, then the access table stores a single write mode access claim for that target.

3.6 Generic Invocation Scheduling Mechanism

The meta-layer represents a generic invocation scheduling mechanism in which concurrent root invocation requests by independent threads are scheduled against each other to achieve serialised execution (atomicity and isolation). Figure 3.17 provides an illustration of the generic invocation scheduler as it relates to the other high level concepts of the scheme.

![Generic Invocation Scheduler Component of Meta-Layer Scheme](image)

The primary aspects involved in scheduling are conflict detection, and subsequently, conflict resolution. The scheduler takes a pro-active approach to resolving conflict, by determining any potential conflict before execution time and hence before any effects can occur. New invocation requests that do not conflict with any existing requests...
Chapter 3. A Meta-Level Scheduler

(being decomposed, scheduled or executing) will start execution immediately. Potential access conflict between competing invocations is resolved by the use of a rule to order the execution of invocations. One of the invocations is selected and its scheduling progress temporarily suspended until the other has finished execution and released access to the source of the conflict.

Due to the synchronisation required within the access decomposition process, two forms of invocation scheduling may occur within the meta-layer.

- An initial form of scheduling may occur during the access decomposition process – if any nodes within the call chain graphs of competing invocations require conflicting access to the member fields of their target instance. The meta-layer effectively schedules conflicting invocations by blocking the progress of access decomposition until member field access is granted.

This phase of scheduling ensures the reliability of the access decomposition process and the accuracy of the information in the access table determined for each invocation.

- The main scheduling phase occurs after access decomposition, when competing invocations are scheduled for execution based on comparisons between their access table requirements and those of other concurrent invocations.

This phase of scheduling ensures atomicity and isolation for the entire duration of each root operation’s execution.

3.6.1 Conflict Detection

The meta-layer uses a pervasive readers/writer policy for synchronising all access to shared objects.

3.6.1.1 Access Lock Types

There are two distinct categories of access locks used to prevent access conflict within the meta-layer.

- **MemberFieldValueTable Access Locks** – each MemberFieldValueTable object manages the requests for access to the potential values of a particular target shared object’s member reference fields. Synchronised read/write access to these potential values is required in order to safely and accurately build a call chain graph model for each invocation. The MemberFieldValueTable manages access according to a readers/writer policy. The MemberFieldValueTable
manages these locks, and hence grants access, on a per call chain graph basis (i.e. per invocation). If different nodes within the same call chain graph have different access modes then the call chain graph as a whole is considered to require the stronger form of access.

- **Execution Access Locks** – these are logical locks managed by the general invocation scheduler and are employed to ensure the atomicity and isolation properties over shared objects during actual execution of invoked operations. Each lock represents access to a particular shared object instance, and the locks are managed according to a readers/writer policy.

### 3.6.1.2 Lock Acquisition Attempt and Release Points

Access conflict is detected between competing concurrent invocations when one of the invocations attempts to acquire access locks held by another invocation. This may potentially occur during two separate phases of the invocation sequence:

- **Access Decomposition:**
  * **MemberFieldValueTable Lock Acquisition** – during the processing (initialisation or revision) of each individual node in a call chain graph, any nodes that require access to their target instance’s member reference fields ensure synchronised access via the target instance’s corresponding MemberFieldValueTable object. If an access conflict arises between competing concurrent invocations then the MemberFieldValueTable enforces an order on the two invocations, granting access to one invocation and temporarily denying access to the other. If access is denied during call chain node processing, then call chain finalisation fails and the access decomposition process is temporarily suspended while waiting for access. However, the invocation’s call chain graph continues to hold all the access locks successfully acquired on other targets. These locks are acquired incrementally as an invocation expands its call chain graph model during access decomposition, holding existing locks and blocking waiting for any required access which is currently unavailable. In resolving conflict between competing concurrent invocations, it is possible that access may be granted to an invocation at one time, but then revoked before the invocation gets to execute. In this case the lower order invocation will be removed from the scheduling queue and will reattempt access decomposition at a later point.
However, once an invocation has started the actual execution phase it cannot have this access revoked.

* MemberFieldValueTable Lock Release – after the scheduling and subsequent actual execution phases, the meta-layer performs post-execution processing. MemberFieldValueTable access locks are released at this point, allowing any blocked invocations to continue with access decomposition.

• Invocation Scheduling –

* Execution Lock Acquisition – once the call chain graph is finalised and the access decomposition mechanism has successfully produced an access table, the invocation scheduling mechanism checks for conflict between this table and the access tables of other concurrent invocations within the meta-layer. The scheduler attempts acquisition, on behalf of the invocation, of the locks corresponding to the access claims in the invocation's access table according to a strict two-phase locking protocol. If there are no conflicts then the invocation may begin the execution phase. If there are conflicts then the invocation scheduling mechanism enforces an order on the execution of the conflicting invocations. Once the meta-layer scheduler allows an invocation to begin the execution phase, all of the access claims within its access table are logically considered to be held for the duration of execution.

* Execution Lock Release – after the actual execution phase, the meta-layer performs post-execution processing. Execution access locks are logically released at this point, allowing any waiting/scheduled invocations to be considered for execution.

3.6.2 Conflict Resolution

Access conflicts between competing invocations must be resolved in a way that is fair and free from potential deadlock situations. Conflicts are resolved by using a rule to enforce an order on the progress of the conflicting invocations. The resulting order is used to determine which of the invocations will be granted access and its processing allowed to proceed, and which will be denied access and be blocked until such time at which the first invocation has finished execution and released access to the source of the conflict.
The rule used to enforce order on conflicting invocations affects the basic scheduling properties of the meta-layer.

The ordering rule has no effect on any conflict with an invocation that has already started the actual operation execution phase. Once execution has started, an invocation is guaranteed atomic and isolated access according to the claims in its access table. If other invocations of higher order subsequently determine access conflict, they must wait until the executing invocation has finished.

3.6.2.1 Invocation Ordering Rule

To ensure fairness for competing threads, the chronological order in which invocations are presented to the meta-layer is taken into account. As each new root Invocation object enters control of the meta-layer, it is assigned an invocation identifier corresponding to its chronological order.

When an access conflict occurs, the meta-layer uses the invocation identifiers of the conflicting invocations to enforce a First-In-First-Out (FIFO) order. A FIFO order for conflicting invocations ensures fairness and prevents starvation by guaranteeing that an invocation presented to the meta-layer cannot be suspended indefinitely due to any form of access conflict detected.

However, a FIFO ordering rule is not an essential requirement for safe operation of the meta-layer scheme. In some scenarios, such fairness between competing invocations is unnecessary or undesirable. The use of alternative ordering rules is discussed in Section 6.4.

3.6.2.2 Uniform Use of Ordering Rule

The meta-layer uses the same ordering rule to resolve every occurrence of potential conflict within and between both the access decomposition phase and the general invocation scheduling phase.

As newly requested invocations enter control of the meta-layer they are processed in turn and scheduled based on conflict with the other invocations concurrently within the meta-layer.

As some mutually exclusive insertion processing must be performed before any potential access conflict can be determined, the ordering rule is also applied to the order of individual invocation's insertion processing by the meta-layer.
When an invocation finishes execution and releases access locks, this may cause blocked invocations to be reconsidered for further processing. All such blocked invocations are considered in turn according to the invocation ordering rule.

### 3.6.3 Invocation Blocking During Access Decomposition

In the course of access decomposition, an attempt is made to finalise the call chain graph. This attempt will fail if any call chain nodes in the graph request, and are not granted access to, the MemberFieldValueTable managing the potential values of their target instance's member fields. In this case, the process of access decomposition for the invocation is temporarily suspended.

The invocation's call chain graph that holds the member field access lock on the contentious target will eventually finish execution and release that access. The blocked invocation is then woken up and the process of access decomposition continued with another attempt at finalisation. Again, if the attempt fails the access decomposition process is suspended.

When finalisation succeeds, the access decomposition mechanism will have determined the access table for the Invocation. Control then proceeds to the invocation scheduling mechanism proper.

However, under certain conditions during call chain finalisation, an invocation may be granted access to a shared object's potential member field values, then at a later stage have that same access revoked and be forced to wait for access again. This can occur in some cases when concurrent invocation requests, involving conflicting access, attempt access decomposition while the objects involved are already locked by other earlier invocation requests under control of the meta-layer. The access synchronisation enforced causes temporary suspensions in the graph construction process which allows other competing invocations a chance to attempt finalisation. Access that has been granted previously has to be revoked if the order of the initialisation of call chain nodes between competing invocations corresponds to the following pattern:

1. Invocation A on object 'x' enters meta-layer and begins access decomposition.
2. Invocation A blocks waiting for access to 'x'.
3. Invocation B on object 'z' enters meta-layer and begins access decomposition.
4. Invocation B requests and is granted access to 'z'.
5) Invocation B has not started actual execution yet and is either blocked in access decomposition waiting for access to some further object, or is in the scheduling queue waiting for execution.

6) Invocation A is granted access to 'x' and resumes access decomposition.

7) Invocation A requests write access to 'z' now,

- access conflict has occurred over object 'z'
- Invocation A is granted access according to the invocation ordering rule because it entered the meta-layer first

and hence,

8) Invocation B must release access on 'z' and then return to access decomposition. It must remove any of 'z's effects on its call chain graph and then wait for Invocation A to finish executing before reattempting call chain finalisation.

3.6.4 General Invocation Scheduling Algorithm

Once the access table describing the potential set of objects and respective access modes involved in a root invocation has been determined, execution of the invocation is then scheduled against other competing concurrent invocations.

An access conflict occurs between the access tables of two invocations if they require conflicting access to the same target object. For example, if one invocation requires write/exclusive access while the other requires read/non-exclusive access then there is a conflict – the two invocations must not be allowed to execute concurrently.

The generalised scheduler maintains a list of currently executing invocations, and a queue of waiting invocations.

The primary invariant that the scheduler enforces at all times is that no invocations start execution that would cause access conflict with the other invocations currently executing. By enforcing this rule, all operations execute atomically and in isolation, which results in successful serialisation of all operations.

In addition, the scheduler uses the waiting queue to impose an execution order on conflicting invocations in order to ensure fairness and prevent starvation.
However, if an independent thread requests an operation that does not conflict with any waiting or executing operations, then it may begin execution immediately.

3.6.4.1 Waiting Queue

Invocations that cannot be executed immediately due to access conflict are stored in the waiting queue. The waiting queue is maintained according to the order of execution that the scheduler will enforce.

In order to be fair, the general scheduling mechanism must guarantee that the execution of an invocation cannot be postponed indefinitely while invocations from other independent threads are executed in preference. This requires that when the scheduling mechanism processes new invocations, it takes into account not only access conflict with currently executing invocations, but also access conflict with other invocations waiting for execution.

When a new invocation is introduced, the scheduler compares the access table of the new invocation with that of the invocation at the tail of the waiting queue. If there are no access conflicts detected between the invocations, then the new invocations access table is compared with that of the next invocation in the waiting queue. This process continues until an access conflict is found, or the head of the queue is reached.

If an access conflict is found then the new invocation is inserted into the queue behind the first conflicting invocation encountered. The invocation will begin execution only after any conflicting invocations ahead of itself in the waiting queue have finished execution.

If the invocation does not conflict with any invocations in the waiting queue then it is considered for progression into the execution phase.

The waiting queue and the order it imposes on conflicting operations ensures fairness for concurrent operations. No thread can starve out another in violation of the ordering rule, because the rule is imposed on competing concurrent operations at every point of conflict. Conversely, the analysis of access sets allows the scheduler to execute non-conflicting operations in an optimally concurrent fashion according to a readers/writer policy.

3.6.4.2 Execution

When an invocation reaches the head of the waiting queue, the scheduler checks for access conflict between its access table and those of all the invocations currently
executing. If the invocation being scheduled does not conflict with any executing invocations, it is removed from the waiting queue, added to the list of executing invocations and actual execution of the operation starts.

If there is any access conflict between the invocation being scheduled and the currently executing invocations, then the invocation remains in the waiting queue.

The executing invocations and their respective access tables form a set of logical locks which guarantee the safety and isolation of concurrent operations. The meta-level scheduler alone manages the logical acquisition and release of these access locks for invocations via a strict two-phase locking style, and so guarantees atomicity while preventing deadlock.

3.6.4.3 Post Execution Rescheduling

When a root operation finishes execution, it is removed from the list of executing invocations and its access locks are logically released. As a result of freeing up those resources, the scheduler may then reconsider some of the blocked invocations for further processing.

Releasing all access locks involves two steps:

1) Release Execution Access Locks – Removing the invocation from the list of executing invocations logically releases the execution access locks held by the invocations access table. This may allow other blocked invocations to begin execution.

2) Release MemberFieldValueTable Access Locks – The CallChainGraph used to determine the access table for the invocation is now disposed of. Any access locks within MemberFieldValueTables held by the call chain nodes within the invocations call chain graph are also released. This may allow other blocked invocations to continue with access decomposition.

Blocked invocations fall into two categories; those waiting in access decomposition, and those waiting in the queue scheduled for execution. To ensure fairness between conflicting invocations, concurrently blocked invocations must be considered in turn according to the invocation ordering rule, regardless of which stage in the invocation sequence they are blocked at.

- Invocations Waiting for Execution – If a waiting invocation’s access table no longer contains an access conflict with any executing invocations, or any
invocations of higher order in the waiting queue, then it may move to the execution processing stage.

- **Invocations Waiting in Access Decomposition** – If an invocation waiting for MemberFieldValueTable access is now granted access, then it may resume access decomposition and make another attempt at call chain finalisation. If call chain finalisation succeeds, then the invocation progresses to the scheduling mechanism proper.

### 3.6.4.4 Invocation Lifecycle

An Invocation presented to the meta-layer is scheduled for execution as soon as possible. Only access conflict with other invocations of higher order can cause execution to be delayed.

At any point during the execution of a method invocation under the control of the meta-layer, its status can be classified into one of the categories listed in Table 3.5.

<table>
<thead>
<tr>
<th>Invocation Status</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>Initialised</td>
<td>The Invocation object has been constructed but not yet presented to the meta-layer for execution</td>
</tr>
<tr>
<td>Insertion</td>
<td>The Invocation object has been presented to the meta-layer but its processing has not been started yet. This occurs momentarily due to the meta-layer's internal synchronisation if many invocations are presented to the meta-layer simultaneously.</td>
</tr>
<tr>
<td>Finalisation</td>
<td>The Invocation is in the process of access decomposition.</td>
</tr>
<tr>
<td>Scheduling</td>
<td>The Invocation is scheduled in order within the waiting queue due to an access conflict with another invocation(s).</td>
</tr>
<tr>
<td>Execution</td>
<td>The Invocation is currently executing the implementation code of the requested operation.</td>
</tr>
<tr>
<td>Post-Execution</td>
<td>The operation has been executed and the meta-layer is currently performing dependent rescheduling.</td>
</tr>
<tr>
<td>Returned</td>
<td>The meta-layer has already returned control and operation return value to the calling thread.</td>
</tr>
</tbody>
</table>

Table 3.5 Method Invocation State Lifecycle
3.7 Condition Synchronisation

The automatic access synchronisation provided by the meta-layer scheme introduces a major effect on the way condition synchronisation can be used in concurrent applications.

However by constraining where and how conditions are tested, condition synchronisation can be employed in a manner that is compatible with the meta-layer's model for automatic access synchronisation.

3.7.1 Constraints on Condition Synchronisation

The meta-layer scheme removes the need for explicit access synchronisation instructions because the general invocation scheduling mechanism guarantees that all operations on shared objects execute (with respect to a readers/writer access policy) atomically and in isolation. These execution qualities also extend over composite operations on shared objects.

However, the scheme represents a trade-off between the atomicity focus of the automatic access synchronisation provided by the meta-layer, and certain forms of interaction such as manual lock acquisition and release, condition synchronisation within atomic actions, and the use of guarded methods.

3.7.1.1 Manual Lock Acquisition and Release Disallowed

Explicit synchronisation code embedded within operation implementations allows the manual acquisition and release of access to shared objects during the execution of composite operations. In contrast, the new scheme provides atomicity and isolation automatically by guaranteeing that all of the shared objects accessed during the execution of an operation are logically locked for the entire duration of the root operation. Safety of the mechanism precludes any manual locking or unlocking of access to shared objects.

The shift from manual style locking to an automatic scheme has an advantage. In composite operations, lock management is handled holistically by the general scheduling mechanism instead of separately by individual code fragments. The automatic scheme allows the use of strict two-phase locking to detect and resolve potential conflicts – such as deadlock and interference – between competing concurrent operations, and also, simplifies the implementations of the individual component's operations.
3.7.1.2 Condition Synchronisation within Atomic Actions Disallowed

The preclusion of manual locking/unlocking of access to shared objects removes the possibility of condition synchronisation within atomic actions. The scheduler's readers/writer access policy guarantees that, if a thread T is executing atomic operation Foo which involves read access to object A, then no other concurrent thread will be able to change the state of object A while thread T is executing operation Foo. This eliminates any viability of thread T blocking to wait for a condition based on the state of A – because the scheduler guarantees that no other thread will be able to execute any operation that could modify A.

For this reason, the scheme does not allow condition synchronisation to be performed within atomic operations on shared objects. Additionally, this eliminates the types of situation where a large grained action, holding access locks on a number of shared objects, blocks at some point midway through execution while waiting on a further condition. Such a style of synchronisation, in which threads perform incremental stages of condition synchronisation while retaining locks on resources, can be prone to deadlock.

3.7.1.3 Guarded Methods Disallowed

Method guards are an abstraction for encapsulating condition synchronisation at the operation level. The use of guarded methods in composite operations results in successive condition synchronisation points, and hence is incompatible with the serialisation/access atomicity focus of the meta-layer synchronisation scheme.

3.7.1.4 Single Condition per Atomic Action

One model for employing condition synchronisation that is compatible with the automatic access synchronisation provided by the meta-layer is a mechanism which allows a single condition synchronisation event to be associated with the consequent execution of some operation on a shared object. When the condition evaluates to true then the operation is subsequently executed according to the atomicity and isolation properties provided by the meta-layer. However, the consequent operation itself may perform no further condition synchronisation within any part of its call chain.

This model for allowing a single condition synchronisation event before an atomic action involves the following components and constraints:

- a \textit{condition} abstraction that allows shared objects to define functions that represent tests on their internal states.
Chapter 3. A Meta-Level Scheduler

- a *conditional invocation* abstraction that allows a client to create a compound invocation that binds a *condition* to a consequent method invocation for evaluation and scheduling by the meta-layer.

- such conditional invocations may not be invoked within shared object method operations – this would allow nested conditional synchronisation within shared object method execution. Instead, conditional invocations are only allowed as root invocations requested by normal objects external to the control of the meta-layer.

### 3.7.2 Condition Functions

Similar to Method definitions, *conditions* may be defined for shared object classes. Condition functions are a special subclass of Method operations. Conditions implement a boolean function which represents some condition relating to the internal state of a shared object. Conditions are constrained to simple read-only access of the shared object target they test. Conditions are not allowed to make nested calls\(^3\), although they may accept primitive or shared object parameters for use in determining the truth value of the condition.

Conditions can not be tested or executed directly by application code, they are only ever tested by the meta-layer scheduler, via conditional invocations or method invocations.

### 3.7.3 Condition Synchronisation via Conditional Invocation

In addition to the invocation mechanism introduced for managing the execution of operations on shared objects, the meta-layer is extended to manage another class of requests that involve condition synchronisation.

Requests for operations involving condition synchronisation take the form of a pre-condition and an optional consequent operation. The meta-layer scheduler delays execution of the consequent operation until the condition becomes true. Then while the

\(^3\)Allowing condition functions to invoke nested calls introduces complications in detecting the times when such conditions may be safely tested. If a condition function requires read access on only its single target, then it may be safely tested immediately following the completion of an atomic action which has updated that target, before considering other actions queued for execution. However, if a composite condition involves read access to more than one object, then the scheduler must ensure safe access to all objects involved before testing the condition. The availability of such objects may not coincide with the access locks released following the completion of atomic actions. Solutions to this problem necessarily involve unacceptable liveness failure or possible missed conditions.
condition is guaranteed to be true, the consequent operation is executed atomically and in isolation. No further condition synchronisation is allowed within the consequent operation, as this would interfere with its atomicity and isolation. Hence, conditional subinvocations are illegal within shared object Methods. Conditional invocations may only be requested as root invocations by threads executing in objects external to the shared objects under the control of the meta-layer.

Whereas a regular Invocation encapsulates the details of a request for an operation on a shared object target, a Conditional Invocation is composed of two components: a condition and a consequent regular method invocation. Figure 3.18 illustrates the relationships between the classes involved in a conditional invocation.

The target of the condition of a conditional invocation may be different to the target of the consequent invocation.

When the condition becomes true, the meta-layer attempts execution of the consequent invocation. If the consequent invocation is omitted, the conditional invocation returns from meta-layer control as soon as the condition becomes true, otherwise the meta-layer attempts to process the consequent invocation according to the normal scheduling rules based on access table conflicts.

3.7.4 Condition Testing via Method Invocation

The meta-layer allows a caller (from within an operation of either a 'normal' or shared object implementation) to test a condition on a shared object target and return the result as soon as possible, without waiting for the condition to become true. As
Condition functions are simply a special case of Method operations, this is achieved via the normal shared object method invocation mechanism. Execution of the test is performed safely and fairly according to the usual scheduling rules based on access conflict and invocation ordering. The condition test result is returned to the caller as a boolean value.

### 3.8 Conditional Invocation Scheduling

#### 3.8.1 Access Conflict and Resolution

##### 3.8.1.1 Access Conflict Involving Consequent Invocations

The consequent invocation is scheduled based on potential access conflict against any other concurrent method invocations within the meta-layer (access decomposition, waiting queue, execution list). Uniform use of the same invocation ordering rule used between plain invocations provides fairness and ensures starvation cannot occur between conditional invocations and method invocations.

When access decomposition of the consequent method invocation is achieved and the consequent invocation's access table does not conflict with those of any waiting or executing invocations, then the consequent operation can begin execution as normal. When execution is complete, the conditional invocation returns control and the return value of the consequent method invocation.

The execution of consequent invocations allows conflict resolution ordering to interrupt the schedulers established order of invocations blocked in either of the access decomposition or the waiting queue stages.

In some cases, due to fairness ordering, a consequent invocation may have to be blocked because of an access conflict with another invocation either in access decomposition, the waiting queue or already executing.

##### 3.8.1.2 Atomicity over Condition and Consequent

In addition to ensuring atomicity and isolation for a consequent operation, the meta-layer must also check that the condition which tested true does not become false before the consequent operation actually enters the execution phase. If the condition becomes false before the consequent can be executed, then the conditional invocation may 'miss' the condition. These situations can occur if a consequent method invocation of higher order interrupts and moves into the execution phase with write-mode access
on the original condition's target object. In this situation, the consequent method invocation of the interrupting conditional invocation consumes the event, and the original conditional invocation must wait until the condition becomes true again.

For example, in some situations, multiple independent conditional invocation requests may be waiting on the same condition. When the condition becomes true, the corresponding conditional invocations are considered in turn according to the meta-layer's invocation ordering rule. It is possible that the access table of the consequent operation for the first conditional invocation may involve write access on the condition target. If this occurs, the consequent operation may cause the condition to become false and so the other conditional invocations must wait for reevaluation after the first consequent operation has finished execution.

In a special case, where the condition of a conditional invocation has become true and the consequent invocation is either blocked in access decomposition, or in the waiting queue, the condition of another unrelated concurrent conditional invocation of higher order may become true. It may interrupt and perform a consequent operation that causes the original condition to become false again. The scheduler checks for this condition and makes the original conditional invocation block again on its condition which ensures that a consequent invocation is only executed while its antecedent condition is true.

A less strict form of condition synchronisation is often required where an independent thread simply wants to delay until a certain condition becomes true, but does not require its next operation to be executed atomically with the condition, or does not want to potentially 'miss' the condition. This can be achieved by splitting the conditional invocation into a consequenceless conditional invocation and a following separate method invocation. Because the consequent is omitted and thus can have no access conflicts, the conditional invocation will return as soon as the condition can be tested and found true.

3.8.2 Conditional Invocation Scheduling Algorithm

The execution of a conditional invocation is summarised by the following sequence of steps:

1) **Check Access on Condition Target** – the invocation requires read access in order to safely test the condition. If any other higher order invocations concurrently within the meta-layer involve write access on the condition target then the conditional invocation is blocked until access is available. The
conditional invocation does not make a permanent read access claim on the target however, because if the condition proves false then the read access claim will be surrendered to allow subsequent method invocations an opportunity to execute and possibly satisfy the condition.

2) **Test Condition** – the condition is tested and if it evaluates to true then the scheduler attempts to process the consequent invocation in step 3. If the condition is false then the conditional invocation must release read access on the condition target and block until it is time to test again. A condition which evaluates to false cannot possibly become true until another method invocation involving write access on the target has finished executing. The post-execution processing of such an invocation will arrange notification of the blocked conditional invocations which require condition reevaluation.

3) **Schedule Consequent Invocation** – when the condition becomes true the consequent invocation is scheduled according to the normal invocation sequence of access decomposition and access based conflict resolution. If the conditional invocation loses read access on the condition target due to an interruption by a conflicting invocation of higher order, then the execution attempt is abandoned to preserve the atomicity of the condition and the consequent, and progress of the conditional invocation returns to step 2. Otherwise, the conditional invocation waits for the consequent operation to be scheduled for execution.

4) **Return** – the meta-layer passes control and the return value of the consequent operation back to the calling thread.

### 3.8.3 Post-Execution Invocation Processing

If the condition of some conditional invocation evaluates to false, the scheduler must block the progress of the conditional invocation and arrange to reevaluate the condition at some appropriate later time.

The truth value of the condition can only change if its target object is updated by some other operation. Further, a condition evaluation can only be allowed at such times as the conditional invocation can be granted temporary read access on the condition target. Accordingly, the meta-layer uses the post-execution invocation processing as an opportunity to reconsider waiting conditional invocations for re-evaluation.

After releasing all access locks held by a method invocation, the post-execution processing sequence must consider waiting conditional invocations as well as other blocked method invocations. To ensure fairness between conflicting invocations, both
blocked method invocations (in either access decomposition or scheduling queue phases) and waiting conditional invocations must each be considered in turn according to the invocation ordering rule:

- **Waiting Conditional Invocation** – The access table of the freshly executed method invocation may contain a write-mode access claims on an object which corresponds to the condition target of a waiting conditional invocation. If so, the waiting conditional invocation must successfully apply for temporary read access on its target in order to evaluate its condition. The conditional invocation must compete against other method invocations in the waiting queue for access to the target. If access is granted and the condition evaluates to true then processing may begin for the conditional invocation’s consequent method invocation.

- **Method Invocation Blocked in Scheduling Queue** – The scheduler examines method invocations that were blocked in the scheduling queue and considers them for promotion to the execution phase.

- **Method Invocation Blocked in Access Decomposition** – The scheduler examines method invocations that were blocked in the access decomposition phase and considers them for another attempt at call chain finalisation. If call chain finalisation is successful, then scheduling processing continues with the method invocation either being inserted into the waiting queue, or moving straight to the execution phase.

### 3.8.4 Conditional Invocation Lifecycle

A conditional invocation presented to the meta-layer is always considered for evaluation as soon as possible, according to the invocation ordering rule. Only access conflict with other invocations of higher order can cause evaluation to be delayed. Once the condition becomes true, the consequent invocation in turn is scheduled for execution as soon as possible. Again, only access conflict with other method invocations of higher order can cause execution to be delayed.

At any point during the execution progress of a conditional invocation under the control of the meta-layer, its status can be classified into one of the categories listed in Table 3.6.
Chapter 3. A Meta-Level Scheduler

<table>
<thead>
<tr>
<th>Invocation Status</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>Initialised</td>
<td>The ConditionalInvocation object has been constructed but not yet presented to the meta-layer for execution</td>
</tr>
<tr>
<td>Insertion</td>
<td>The ConditionalInvocation object has been presented to the meta-layer but its processing has not been started yet. This occurs momentarily due to the meta-layer's internal synchronisation if many invocations are presented to the meta-layer simultaneously.</td>
</tr>
<tr>
<td>Waiting</td>
<td>The invocation is waiting for the condition to be true. In order to test the condition, the invocation must be granted read access to the target of the condition, which requires waiting behind other conflicting invocations of higher order which involve write access on the condition target.</td>
</tr>
<tr>
<td>Finalisation</td>
<td>1) The condition is true, and remains true while 2) The process of access decomposition is performed on the consequent MethodInvocation. If the first clause becomes false, due to an interruption, then progress moves back to the 'Waiting' stage.</td>
</tr>
<tr>
<td>Scheduling</td>
<td>1) The condition is true, and remains true while 2) The consequent MethodInvocation is scheduled in order within the waiting queue due to an access conflict with another invocation(s). If the first clause becomes false, due to an interruption, then progress moves back to the 'Waiting' stage.</td>
</tr>
<tr>
<td>Execution</td>
<td>1) The condition is true, and 2) The consequent MethodInvocation is currently executing the implementation code of the requested operation.</td>
</tr>
<tr>
<td>Post-Execution</td>
<td>The consequent operation has been executed and the meta-layer is currently performing post-execution rescheduling.</td>
</tr>
<tr>
<td>Returned</td>
<td>The meta-layer has already returned control and consequent operation's return value to the calling thread.</td>
</tr>
</tbody>
</table>

Table 3.6 Conditional Invocation State Lifecycle

3.8.5 Application Condition-Based Deadlock

The meta-layer itself ensures that no opportunities for access based deadlock can occur between or within the processing of method invocations or conditional invocations. Conditional invocations that are waiting for a false condition to become true hold no access locks and prevent no other invocations from executing.
Still, it is possible that application programs with concurrency designs involving conditional interaction may still reach condition-based deadlock. However, these types of deadlocks do not stem from the implementation of the synchronisation meta-layer, but are due to logical errors in the set of conditions protecting the actions in the system. These situations can be resolved by re-analysing the conditional interaction in the system.

3.9 Discussion

The meta-layer scheme removes the need for explicit access synchronisation because the general invocation scheduling mechanism guarantees that all operations on shared objects execute atomically and in isolation, with respect to a readers/writer access policy. These execution qualities also extend automatically over composite operations on shared objects.

In terms of access synchronisation, this scheme introduces a higher level of abstraction, because it directly associates operation execution with atomic object access. Equally important, the abstraction is designed to be compatible with composition – the meta-layer ensures that the execution properties of atomicity and isolation are extended over composite operations, whilst eliminating interference such as race conditions and access deadlock.

3.9.1 Main Features

The shared object model, invocation mechanism and scheduling meta-layer introduce a higher level of abstraction for defining, requesting and coordinating the execution of operations on shared objects in multithreaded object-oriented programs. The abstractions greatly simplify the programmer’s task of synchronising access to shared objects, especially in dynamic cases involving composite operations over multiple objects.

- access synchronisation - the scheme provides access synchronisation automatically for all requests for operations on shared objects according to a readers/writer policy.

- atomicity and isolation for composite operations - the scheme automatically provides the properties of atomicity and isolation which encompass the complete execution of each individual root invocation.
Chapter 3. A Meta-Level Scheduler

The meta-layer scheduler shields the application programmer from the considerable complexity of conflict detection and fair resolution required for ensuring access synchronisation for composite operations involving multiple objects.

3.9.2 Benefits

3.9.2.1 Preserving Encapsulation of Shared Components

In other approaches, atomicity for composite operations is achieved through the introduction of explicit synchronisation instructions within the implementations of collaborating classes. This introduces a form of coupling between such classes, and often involves implementation-dependent assumptions which violate the encapsulation of collaborator classes.

In contrast, under this scheme, access synchronisation is managed implicitly by the meta-layer, and so implementation code does not have to include any explicit access synchronisation instructions when invoking operations on other shared objects.

A shared object’s operation implementation code may freely invoke nested suboperations on other shared objects because the meta-layer manages access synchronisation via the use of the operation meta-information. These declarative elements of meta-information themselves do not violate the encapsulation of any classes either. They are simply private declarations of the interaction inherent within their associated operation's implementation, and refer only to the names of invoked operations and their potential targets. Hence, access synchronisation is achieved without introducing any additional coupling between collaborating classes. Any coupling between collaborating objects is limited to names of the operations defined in the public interfaces of the shared object components. Hence the scheme allows complete encapsulation of shared component classes.

The encapsulation of components is a key feature of object-oriented design. When component class implementations are independent of each other, systems can achieve higher flexibility, maintainability and reusability.

In contrast, schemes which violate encapsulation by using implementation-specific access synchronisation instructions within implementation code suffer from component fragility and limited reusability.
3.9.2.2 Increased Robustness of System Access Synchronisation

Many schemes operate through the explicit construction of a co-operative system synchronisation scheme which is realised through the implementation code of the participating objects. If the interaction of components within the system is modified, or if new participants or patterns of interaction are added, the synchronisation scheme must often be completely reworked to accommodate the changes. Even the synchronisation code of components whose role and function have not changed may often be reworked to ensure correct operation and safety from interference and deadlock issues. In this sense, such systems are extremely fragile and difficult to maintain.

In contrast, the access synchronisation provided by the meta-layer is generated dynamically at run time according to the operation meta-information declared by the participant shared classes. As a result, patterns of interaction may be changed or removed, new participants may be added, and the meta-layer will continue to schedule requests safely and fairly based on access conflict.

3.9.2.3 Increased Robustness of System Condition Synchronisation

The meta-layer scheme introduces a higher level abstraction for condition synchronisation which allows the use of a condition to represent a particular internal state of a shared object. The conditional behaviour of an independently threaded task can be specified in terms of the abstract condition, rather than being coupled to a lower level synchronisation primitive such as a semaphore or condition variable.

As a result, modifications or extensions may be made to the design and implementation of the object for which the condition represents a certain situation, or new conditional participants may be added, and the meta-layer will continue to schedule and resolve such conditional requests safely and fairly4.

3.9.2.4 Compatibility with Polymorphism

The access synchronisation scheme is generated dynamically by the meta-layer at runtime based on the operation meta-information. The access decomposition mechanism that models the potential call chains involved is fully compatible with

---

4Due to the nature of condition synchronisation, as discussed in Section 3.8.5, systems involving this type of synchronisation cannot be guaranteed free from side-effects when changing or adding participant interaction. Such systems inherently possess some degree of fragility due to the logical coupling between the interaction of conditional behaviour of associated concurrent activities.
polymorphism among the potential targets of operations, and retrieves operation meta-information based on runtime type identification of potential target instances. As a result, access synchronisation is always based the actual concrete type of the shared objects involved.

Hence, polymorphism remains effective in achieving client generalisation. New subclasses of shared components, which might involve different patterns of shared object interaction, may be introduced to a system without requiring changes to the clients that may interact with them.

3.9.2.5 Compatibility with Inheritance

As discussed in Section 2.3.4, static synchronisation schemes often conflict with the inheritance concept of the object-oriented paradigm. The concept of inheritance enables incremental specification of classes by adding or changing individual operation implementations, without breaking the encapsulation of the superclass. The problems with inheritance in concurrent environments arise when trying to incrementally specify the changes in synchronisation requirements for a new subclass, while ensuring that the existing static synchronisation model of the superclass does not interfere with the operation of synchronisation for the subclass. The conflict may stem from one of the following situations:

- Monolithic specification of a synchronisation scheme per class
- Synchronisation instructions embedded within operation implementations

The meta-layer scheme avoids both of these situations. Instead of attempting to specify a class' access synchronisation scheme statically at compile-time, it is generated at runtime per invocation based on meta-information declared per operation. A shared class may be extended incrementally with the addition or modification of operations. These operations are accompanied by new meta-information declarations, which are then considered dynamically when synchronising requests on the subclass. If an operation overridden by a subclass calls the original operation defined in the superclass, then the access synchronisation requirements of both operation implementations are merged automatically by the access decomposition mechanism.
3.9.2.6 Increased Maintainability and Reusability

Because the scheme maintains compatibility with the principle features of object-oriented design, these features can be exploited to achieve the benefits of increased reusability and maintainability.

The dynamic runtime-based nature of the synchronisation scheme and the ability to encapsulate components provides a distinct benefit, as it allows shared classes to be designed and maintained independently of each other, but still be combined together safely and seamlessly within multithreaded programs.

Shared components may be reused in a variety of different ways. The same component class can be reused without modification in separate programs containing different patterns of concurrent access. In addition to class reuse, inheritance can be used to achieve code reuse between related shared classes.

3.9.2.7 Pervasive Readers/Writer Access Policy

The pervasive use of a general readers/writer access policy within all aspects of the meta-layer's operation enables an optimal degree of concurrency over the shared objects within the system, while maintaining safe and isolated execution.

The complex task of detecting and appropriately resolving access conflict over multiple objects, however, is shielded from the application programmer by the meta-layer.

3.9.2.8 Avoidance of Potential Liveness Failure

The use of a Strict Two-Phase Locking approach for synchronising access to the set of objects involved within a composite operation avoids the potential opportunities for deadlock that are inherent in other approaches. The task of identifying this set of objects, however, is shielded from the application programmer and from the caller object, by the meta-layer's generic access decomposition mechanism.

The invocation ordering rule enforced on conflicting invocations by the meta-layer avoids the potential opportunities for starvation between conflicting concurrent tasks. The meta-layer scheduler overrides the host languages thread priority mechanism to provide fair, safe, and optimally concurrent invocation scheduling.
Chapter 3. A Meta-Level Scheduler

3.9.3 Costs

The synchronisation scheme introduces additional layers of abstraction for describing, invoking, evaluating and scheduling operations on shared objects. The power and flexibility offered by these abstractions come at the cost of increased resource utilisation.

3.9.3.1 Memory Overhead

The meta-layer scheme requires the following static memory resources related to shared object instances:

- per shared class:
  * MethodMetaInformation aggregate object per operation
- per shared object instance:
  * Method object for each operation
  * MemberFieldValueTable if required

The meta-layer scheme uses the following dynamic memory resources related to each scheduled request (root invocation):

- per root invocation:
  * Invocation object
  * CallChainGraph containing CallChainNode aggregate objects.
  * AccessTable containing AccessClaims

Subinvocation requests nested within composite invocations result in the creation of an Invocation object only – no meta-layer processing is required, the subinvocation executes immediately, and thus, call chain graphs and access tables are not required.

3.9.3.2 Processing Overhead

In comparison to the other schemes, the meta-layer approach represents a significant increase in the amount of processing involved in coordinating requests for operations on shared objects.

The flexibility offered by the meta-layer approach partly results from the way in which it determines the access requirements for each root operation requested at runtime dynamically. In contrast, many synchronisation schemes employ a more rigid
approach which involves specifying such access requirements explicitly and statically in class definitions.

For situations in which such a static approach satisfies access synchronisation requirements, the meta-layer’s dynamic approach represents a significant waste of processing resources – the access decomposition mechanism may expend processing resources for each individual invocation of the same operation only to continually arrive at the same access table result. Section 6.2 describes a caching optimisation extension to the meta-layer design, which recognises and greatly reduces the access decomposition processing required in such static situations.

Still, the flexibility and dynamism of the meta-layer approach comes at the cost of an invocation mechanism of higher processing overhead.

3.9.3.3 Object Model Heterogeneity

The mechanics of the meta-layer require changes to the object model for instances to be shared, which impacts the interoperability of components with the approach. Without special tools to automatically convert class definitions to the required form of object model, incompatibilities arise:

- Shared components built with the approach will most likely be unusable with different synchronisation approaches in other programs.
- In some cases, in order to employ existing 'normal' classes as shared components, reimplementation or some form of adaptation of may be required5.

However, the heterogeneous object model approach does allow the impact of the meta-layer scheme to be limited to only those areas within an application that share objects.

3.9.4 Issues

3.9.4.1 Control Over Granularity of Atomic Actions

The scope of atomicity provided by the synchronisation meta-layer extends over the entire duration of the resultant call chain of each root invocation. If the root operation involves nested calls to operations on other shared objects, then the scope of atomicity

5See Section 3.3.5.
encompasses those shared objects as well. In effect, all of the shared objects accessed during the execution of each composite root operation are logically locked for the entire duration of that root operation.

This constrains the granularity of atomic actions according to the structure of the root operations invoked at execution time. One arguable drawback is that the implementation code of the component object’s methods lose control over the granularity of atomic actions. This can be seen as a positive feature because it allows a more holistic approach to the synchronisation of composite operations.

Many current approaches to access synchronisation involve explicit synchronisation code embedded within component method implementations. These approaches, through manual access locking/unlocking strategies, allow individual operations to exercise control over the granularity of atomic actions. A drawback with this approach is that the explicit synchronisation instructions often interfere with further functional composition. Operations have no fixed "top" – any composite operation may itself (unbeknownst to the original programmer) be used as a functional component within a larger composite operation. In such situations, the synchronisation code within the suboperations may conflict with or fail to address the synchronisation requirements of the larger operation as a whole.

In contrast, the new scheme automatically provides atomicity and isolation qualities extending dynamically over all the objects involved in the call chain of each root operation. As a result, granularity of atomic actions is dictated dynamically by the structure of the root operation requests presented to the scheduling mechanism at runtime.

Effectively, the sequence of root operations invoked from the application task level code corresponds to an identical sequence of large grained atomic actions on shared objects. This shifts control over atomicity granularity from being specified by the individual component operations, to the structure of the call chains of the root operations requested at runtime.

3.9.4.2 Liveness and Availability

As with any synchronisation scheme, care needs to be taken to ensure that a system does not suffer from liveness or availability failure.

Because this scheme provides atomicity and isolation over the entire duration of each root operation, the system designer must ensure that application task level work is
broken up and implemented as a sequence of atomic root operations of appropriate granularity.

If a system is built which involves atomic actions that are too large grained, then concurrent threads may suffer a reduction in liveness, availability and parallelism as the scheduling mechanism ensures isolation between conflicting actions. This may result in the scenario where other concurrent tasks cannot proceed because they require conflicting access with shared objects held by an already executing or scheduled operation. It may be that the large grained operation only uses the contentious object initially, and then does not access that object again within the entire duration of the root operation. However, due to the atomicity and isolation properties enforced by the meta-layer, the contentious object remains locked for the entire duration of the large-grained root operation.

In contrast, smaller grained actions allow more potential for concurrency, but at a cost of more meta-layer processing and scheduling overhead.
Chapter 4

Implementation Issues in Proof-of-Concept

This chapter discusses the issues arising in a proof-of-concept implementation of the meta-layer synchronisation scheme using the Java™ programming language. Section 4.1 introduces the goals of the implementation and discusses the tradeoffs between introducing new language extensions or providing a framework of supporting classes. Section 4.2 explains where and how the additional elements of the shared object model required by the approach are incorporated into definitions of shared classes. Section 4.3 describes the operation of the invocation mechanism for operations on shared objects. Section 4.4 explains the implementation of the general invocation scheduler and the synchronisation scheme to control the progress of concurrent threads within the meta-layer.

4.1 Introduction

The meta-layer synchronisation approach described in Chapter 3 introduces new ways of describing, invoking and coordinating the execution of operations on objects. The approach requires associating information to and between software elements in ways that existing programming languages were not designed for.

An ideal implementation of the meta-layer synchronisation approach would involve extending a language to provide:

- built-in compiler language support for the new abstractions for describing and invoking operations on shared objects:
  - native shared object model
  - implicit auto-generation of operation meta-information
- native language support for invoking operations on shared objects

- built-in optimised runtime support for managing the execution of operations on shared objects

- native runtime invocation scheduling meta-layer

A language and runtime environment with such native support would cause minimal impact on the implementation of both the shared classes and the objects which interact with them.

The proof-of-concept implementation described in this chapter however, does not introduce any language extensions. The scheme is implemented as a reusable framework of classes for use with the Java™ programming language and virtual environment. The implementation makes use of newer language features such as reflection and inner classes, and as a result requires at least version 1.1 of the Java Platform Core API.

The use of a class framework, instead of language extensions, to implement the concepts of the new approach does have drawbacks. Language extensions allow the introduction of a special grammar and syntax which can be employed to describe and invoke operations on shared objects. Such language extensions can enable the semantic specification of elements within the scheme in a concise manner. Without language extensions, components must work within the syntactic and semantic confines of the host language. As a result, there is often unnecessary verbosity introduced into the shared classes which make use of the framework components. Further, compile-time type checking of parameters and return types under a general invocation mechanism can become complicated without introducing additional 'baggage' to shared classes.

Appendix A provides an overview and a class diagram of the component classes of the framework.

4.2 Shared Object Model

4.2.1 SharedObject Base Class

The SharedObject class is provided as an abstract base for all classes of objects to be shared.
4.2.2 Method Definition

The proof-of-concept framework replaces normal operation implementation functions with the Method abstraction. There are a number of constraints which impose on possible architectural choices for implementing the Method abstraction.

- A Method must encapsulate and associate an operation implementation function with a meta-information declaration function.
- Clients of the shared class require a mechanism to invoke an operation on a shared object via the scheduling meta-layer.
- The mechanism must have controls to prevent a client from circumventing the scheme by executing an operation implementation on a shared object directly.
- Methods (and their signatures) must be publicly defined elements of a shared class' interface.
  - Compile-time checking of operation name, parameter types and return value type.
  - Operations defined by shared classes which make subinvocations to operations on other shared classes must be able to refer to such operations in their own operation's meta-information declaration functions (for the purpose of access decomposition).
- The operation implementation code within a Method definition requires direct access to the member fields of its target object.

The proof-of-concept implementation provides an abstract Method class which is used as a base for defining all operations on shared classes.

4.2.2.1 Methods are Inner Classes

Operations, in the form of concrete subclasses of Method, are defined as inner classes [AG97] of the parent shared classes they belong to. This requires a Method object instance for each operation provided by each shared class instance. The inner class concept allows the member functions of inner classes to directly access all of the member fields of the enclosing/parent object.

4.2.2.2 Method Instances Accessed Via Member Fields

An additional public member field is defined per operation and initialised to refer to an instance of its corresponding Method object. As an arbitrary convention, the
Method object's name has its first letter capitalised, while the name of the member field referring to that Method has the same name except for a lowercase first letter.

For example, if a shared class SomeSharedClass defines operation 'Foo', then a member field 'foo' is define to hold a reference to an instance of the Foo Method. i.e.

```java
public class SomeSharedClass extends SharedObject {

    // declare Method 'Foo'

    public class Foo extends Method {

    }

    // declare and initialise member field 'foo'
    // to refer to an instance of the 'Foo' method

    public Foo foo = new Foo();
}
```

Thus, after instantiating the shared class

```java
SomeSharedClass instance = new SomeSharedClass();
```

then the Method object for operation Foo is accessible via

```java
instance.foo
```

### 4.2.2.3 Method Implementation Function

Operations provide an implementation function, within their concrete Method class definition, as a member function named impl().

The impl() function signature may be modified arbitrarily to accept parameters and return a value.

The impl() function has full access to the member fields of the parent object.

Functionality built into the base class Method object uses reflection to locate the implementation function by name ("impl") and hence to determine the parameter list and return type.

In this implementation, to simplify syntax and compile-time linkage, the parameter types are limited to objects. These may be immutable 'normal' objects (Integer,
Boolean, Double, String etc.) or subclasses of SharedObject. Thus, primitive parameter types (e.g. int, boolean, char) are excluded. A native or ideal implementation would not involve this constraint.

The return value of the method implementation may be of any type, including primitive types. Primitive type return values are automatically enclosed in a corresponding wrapper object.

4.2.2.4 Meta-Information Declaration Function

Operations may provide a meta-information declaration function, within their concrete Method class definition, as a member function with the following signature:

```java
public void meta()
```

Each `meta()` function is called once by the meta-layer during class initialisation to create a MethodMetaInfo object for each type of concrete Method operation.

The `meta()` function declares the operation's meta-information by calling appropriate functions in the Method base class to describe target access mode and shared object interaction. These functions are described in below in Section 4.2.4.

4.2.3 Condition Definition

Conditions are implemented as concrete subclasses of the Condition class, which is a special subclass of the Method class. Conditions are inner classes that are allowed read access to the member fields of the parent shared class, but are not allowed to update the values of those member fields, nor are they allowed to make any kind of subinvocation on other shared objects. Conditions are automatically associated with read-mode target access and therefore do not require an explicit meta-information declaration function.

4.2.2.2 Condition Implementation Function

Conditions provide an implementation function, within their concrete Condition class definition, as a boolean member function named `cond()`.

The `cond()` function signature can be modified to accept arbitrary parameters but must always return a boolean value.

---

1 Other types of 'normal' objects are also allowed under certain conditions. See Section 3.3.5.
Although the parameter fields may be of arbitrary type, the `cond()` function may not perform any subinvocations which would interfere with the operations of any other shared objects in the system\(^2\).

The `cond()` function has read access to the member fields of the parent object.

Functionality built into the base class `Method` object uses reflection to locate the condition function by name ("cond") and hence determine the parameter list.

### 4.2.4 Method Meta-Information

In an ideal implementation, operation meta-information could be determined by a compiler or pre-processor and would be implicit to the shared classes involved.

In the proof-of-concept implementation however, meta-information declaration functions for individual operations must be determined and explicitly coded by the programmer when required.

In the case of simple operations, where there is no shared object interaction, the meta-information required is trivial and pertains only to the target access mode of the operation. For convenience, the `Method` base class supplies read-mode access on the target as default behaviour. Operations which may involve write-mode access, or require mutual exclusion for some other reason, may also declare the convenient mixin interface `Mutative` which overrides the default to enforce write-mode access. Hence simple operations do not require an explicit meta-information function. For example, the following code illustrates two simple Method definitions which do not require explicit meta-information declarations. The first uses the default read-mode target access and the second enforces write-mode target access.

```java
public class GetTitle extends Method {

    public String impl() {
        return title;
    }
}

public class SetTitle extends Method implements Mutative {

    public void impl(String t) {
        title = t;
    }
}
```

\(^2\)See Section 3.3.5.
In the case of composite operations, or simple operations which may modify or return shared object references, an explicit meta-information function is required. The Method base class defines the meta() member function which must be overridden by concrete Method classes to declare the shared object interaction for these non-trivial cases.

The two aspects of the meta-information declaration which describe shared object interaction (shared object reference field declarations and shared object action declarations) correspond to calls to member functions of the Method base class. These functions contain instructions which build a MethodMetaInformation object associated with the concrete Method operation.

### 4.2.4.1 Shared Object Reference Field Declarations

The shared object reference field declaration functions provided by the Method base class correspond to the functions introduced in Table 3.1 in Section 3.3.3.3. These functions are as follows:

- **Parameterisation** –
  
  This initialisation call is required before attempting to obtain any SharedObjParameter objects representing a shared object parameters to the operation.

  ```java
  public void parameterise(String p)
  public void parameterise(String p1, String p2)
  public void parameterise(String pa[])
  ```

  where

  - **p, p1, p2** are arbitrary string literals which the meta-layer attaches sequentially to the actual parameters to the operation.
  - **pa** an array of arbitrary string literals which the meta-layer attaches sequentially to the actual parameters to the operation.

- **Declaring SharedObject Parameters** –

---

3 Described at a high-level in Section 3.3.2.
4 The Java™ 1.1 Reflection API does not provide any mechanism for referring to parameters by name. Parameter types are referenced by the order in which they appear in a method signature's parameter list. This initialisation call attaches name strings to the parameters by their order in the parameter list, and subsequently allows meta-information declarations to then obtain SharedObjParameter objects by such name strings.
This call returns a SharedObjParameter instance that represents a shared object reference parameter to the Method.

\[\text{public \ SharedObjParameter} \ \text{parameter(String name)}\]

where

\[\text{name} \quad \text{is the string literal use in the parameterise() call which corresponds to the desired parameter to the operation.}\]

- Declaring SharedObject Member Fields –

This call returns a SharedObjMemberFld instance that represents a shared object reference field of the operation's target instance.

\[\text{public \ SharedObjMemberFld} \ \text{memberFld(String name)}\]

where

\[\text{name} \quad \text{is a string literal corresponding to the name of the field in the Method's parent/enclosing shared object.}\]

- Declaring SharedObject Member Fields (self reference) –

This call returns a SharedObjMemberFld instance that represents a shared object reference to the operation's target instance itself.

\[\text{public \ SharedObjMemberFld} \ \text{self()}\]

- Declaring SharedObject Subinvocation Result Fields –

This call returns a SharedObjSubResult instance that represents a shared object reference field which containing the result of a subinvocation from this operation.

\[\text{public \ SharedObjSubResult} \ \text{subinvocationResult(}
\quad \text{Class} \ t,
\quad \text{SubinvocationDescriptor} \ \text{sid})\]

where

\[\text{t} \quad \text{is Class object representing the type returned by the subinvocation.}\]
is a SubinvocationDescriptor\textsuperscript{5} describing the subinvocation which is the source of the result.

4.2.4.2 Shared Object Action Declarations

The shared object action declaration functions provided by the Method base class correspond to the functions introduced in Table 3.2 in Section 3.3.3.4. These functions are as follows:

- Declaring Shared Object Subinvocations –
  
  This function declares the details of a potential subinvocation of a particular method operation on a shared object target contained in a particular shared object reference field. Regardless of the actual formal parameters to the subinvocation, the meta-layer is only concerned with those parameters which are shared objects – other arguments are omitted.

  The function returns a SubinvocationDescriptor object. If the subinvocation returns a shared object, then the subinvocation descriptor can be used in a subsequent shared object reference field declaration to obtain a SharedObjSubResult instance for use in further action declarations:

  \[
  \begin{align*}
  \text{public SubinvocationDescriptor} & \quad \text{subinvocation(} \\
  & \quad \text{SharedObjRefFld target,} \\
  & \quad \text{String methodName)}
  \end{align*}
  \]

  \[
  \begin{align*}
  \text{public SubinvocationDescriptor} & \quad \text{subinvocation(} \\
  & \quad \text{SharedObjRefFld target,} \\
  & \quad \text{String methodName,} \\
  & \quad \text{SharedObjRefFld pi)}
  \end{align*}
  \]

  \[
  \begin{align*}
  \text{public SubinvocationDescriptor} & \quad \text{subinvocation(} \\
  & \quad \text{SharedObjRefFld target,} \\
  & \quad \text{String methodName,} \\
  & \quad \text{SharedObjRefFld pi,} \\
  & \quad \text{SharedObjRefFld p2)}
  \end{align*}
  \]

  \[
  \begin{align*}
  \text{public SubinvocationDescriptor} & \quad \text{subinvocation(} \\
  & \quad \text{SharedObjRefFld target,} \\
  & \quad \text{String methodName,} \\
  & \quad \text{SharedObjRefFld[]} \quad \text{parameters)}
  \end{align*}
  \]

  where

\textsuperscript{5}See Section 4.2.4.2, next.
target is some concrete instance of SharedObjRefFld obtained via a shared object reference field declaration function representing the target of the invocation.

methodName is a string literal containing the name of the Method to be invoked.

p1,p2 are SharedObjRefFld instances representing shared object reference values that will be passed as parameters to the invocation.

parameters is an array of SharedObjRefFld instances representing shared object reference values that will be passed as parameters to the invocation.

- Declaring Shared Object Member Field Assignments –

This call declares a potential assignment of a reference to some shared object to a member field of the operation's target instance.

    public void assignMemberFld(SharedObjMemberFld dest, 
   SharedObjRefFld source)

where

    dest is SharedObjMemberFld representing the destination of the assignment.

    source is some concrete SharedObjRefFld representing the source of the assignment.

- Declaring Shared Object Return Value –

This call declares the potential use of a reference to some shared object as the operation's return value.

    public void returnSharedObject(SharedObjRefFld fld)

where

    fld is some concrete SharedObjRefFld representing a field returned by the operation.
4.2.4.3 Encapsulation and Meta-Layer Access to Member Fields

Both the meta-information declaration functions and the meta-layer's access decomposition mechanism require identification of, and access to, the member fields of shared objects that refer to other shared objects. The proof-of-concept implementation achieves this in a generalised way using the Java™ Reflection API to locate and retrieve values from these types of member fields.

The Reflection API supports access to only the public members of a class. Hence, the scheme implementation requires that member fields that refer to other shared objects must be declared with the public visibility modifier, as opposed to the private, protected, or default visibility options.

A native implementation of the meta-layer scheme would remove this limitation and allow complete encapsulation of shared class member fields.

4.2.4.4 Runtime Initialisation of Meta-Information

The meta-layer scheme processes the Method meta-information declaration functions for each shared class at initialisation time to create corresponding MethodMetaInfo objects. This initialisation process requires access to the member fields defined in the concrete subclasses of SharedObject. This mechanism cannot be built into the SharedObject base class constructor, because the member fields of the concrete class are not accessible via the Reflection API until after the concrete class' constructor has executed [AG97].

By convention, the proof-of-concept implementation requires a call back to the SharedObject base class function metaInitialise() in each concrete shared object's constructor, in order to perform initialisation of Method meta-information. In the case of class hierarchies that employ inheritance to reuse code among shared classes, the implementation correctly handles any multiple calls to metaInitialise() that occur during instance construction.

A native or ideal implementation of the meta-layer scheme would not require this convention.

4.3 Invocation Mechanism

There are two steps required to initiate processing of a request on a shared object. Firstly, the invocation must be reified – an Invocation object must be constructed and initialised appropriately – to provide a representation of the request within the meta-
layer. Secondly, control and the Invocation object must be passed to the meta-layer scheduler for processing.

Both steps are conveniently performed by a generalised 'execution' function provided by the Method and Condition base classes.

**4.3.1 Invocation Syntax**

**4.3.1.1 Method Invocation**

A client requests an operation on a shared object by using the following syntax:

\[
rv = target.method.exec();
\]

or

\[
rv = target.method.exec(p1,...);
\]

where

- `target` is the `SharedObject` target of the invocation
- `method` is the target's member field referring to the Method
- `p1...` are the parameter objects to the method.
- `rv` is the return value object of the operation.

**4.3.1.2 Condition Test Invocation**

A client tests a condition on a shared object by using the following syntax:

\[
rv = target.condition.test();
\]

or

\[
rv = target.condition.test(p1,...);
\]

where

- `target` is the `SharedObject` target of the invocation
- `condition` is the target's member field referring to the Condition
- `p1...` are the parameter objects to the condition.
- `rv` is the boolean truth value of the condition.
4.3.1.3 Conditional Invocation

A client may request to wait until a certain condition defined by a particular shared object becomes true by using the following syntax:

\[
\text{condTarget.condition.waitTrue();}
\]

or

\[
\text{condTarget.condition.waitTrue(p1,...);}
\]

where

- \(\text{condTarget}\) is the \(\text{SharedObject}\) target of the invocation
- \(\text{condition}\) is the target's member field referring to the \(\text{Condition}\)
- \(p1...\) are the parameter objects to the condition.

4.3.1.4 Conditional Invocation with Consequent Method Invocation

A client may request atomic processing of an operation on a shared object as consequence of a condition on a particular shared object becoming true by using the following syntax:

\[
\text{MethodInvocatio} n \ mi = target.method.makeInv(p1,...);
\]

then

\[
rv = \text{condTarget.cond.execTrue(mi)}
\]

or

\[
rv = \text{condTarget.cond.execTrue(p2,...,mi)}
\]

where

- \(\text{target}\) is the \(\text{SharedObject}\) target of the consequent invocation
- \(\text{method}\) is the consequent target's member field referring to the \(\text{Method}\)
- \(p1...\) are parameter objects to the consequent method
- \(\text{condTarget}\) is the \(\text{SharedObject}\) target of the conditional invocation
- \(\text{condition}\) is the condition target's member field referring to the \(\text{Condition}\)
- \(p2...\) are parameter objects to the condition
- \(rv\) is the return value object of the consequent operation.
Chapter 4. Implementation Issues in Proof-of-Concept

4.3.2 Invocation Reification

Each request on a shared object by a client results in the creation of an invocation object which will be scheduled by the meta-layer. There are two concrete subclasses derived from the abstract Invocation base class. MethodInvocation is used to schedule invocations of Method operation on shared objects, while ConditionalInvocation is used to manage the testing of a Condition on a shared object, and optionally coordinate the atomic execution of a consequent MethodInvocation.

4.3.2.1 MethodInvocation Reification

The Method class manages the creation of an Invocation object, on behalf of a caller's request, for a particular operation on a particular shared object instance. Each concrete Method instance corresponds to a particular operation on a particular target shared object. The Method base class provides the following functions which construct and initialise an Invocation object appropriately:

```java
public MethodInvocation makeInvoke()
public MethodInvocation makeInvoke(Object p1)
public MethodInvocation makeInvoke(Object p1, Object p2)
public MethodInvocation makeInvoke(Object[] pa)
```

where

- `p1...` are the parameter objects to the method invocation.
- `pa` is an array of parameter objects to the method invocation.

Note however, that the exec() and test() functions described above are provided as convenient wrappers around this functionality.

4.3.2.2 ConditionalInvocation Reification

The Condition class manages the creation of a ConditionalInvocation object, on behalf of a caller's request, for a particular condition on a particular shared object instance. Each concrete Condition instance corresponds to a particular condition on a particular target shared object. The Condition class provides the following functions which construct and initialise an ConditionalInvocation object appropriately:

```java
public ConditionalInvocation makeCondInvoke()
public ConditionalInvocation makeCondInvoke(Object p1)
public ConditionalInvocation makeCondInvoke(Object p1, Object p2)
public ConditionalInvocation makeCondInvoke(Object[] pa)
```
where

\[ p1... \]  are the parameter objects to the conditional invocation.

\[ pa. \]  is an array of parameter objects to the conditional invocation.

Note however, that the `execTrue()` and `waitTrue()` function described above are provided as convenient wrappers around this functionality.

### 4.3.3 Execution

Once the Invocation object is constructed and initialised, it must be passed to the meta-layer scheduler for processing.

The Invocation base class defines the abstract function

```java
public abstract ReturnValue synchSchedule()
```

which performs synchronous execution of the invocation via the scheduling meta-layer.

The MethodInvocation and ConditionalInvocation concrete classes override this function to call the appropriate entry point in the meta-layer scheduler's interface, passing a reference to themselves as a parameter. From this point, the meta-layer controls progress of the thread, before eventually executing the operation and returning control and return value.

### 4.3.4 Return Value

To allow return values of different types, the entry point to the scheduler returns a ReturnValue object which is a wrapper around the actual return value.

The default high level invocation mechanism (`Method.exec()` and `Condition.execTrue()`) unwrap this object to return the actual return value as an object.

If the actual return value is of primitive type, it is wrapped in an object corresponding to the primitive type. For example, a boolean return value is wrapped in a `Boolean` object.
4.3.5 Type Checking

The invocation mechanism of the proof-of-concept implementation is a generalised mechanism that exploits reflection to determine the actual operation's implementation function and perform parameter linkage.

Without a pre-processing stage or customised compiler, providing compile-time type checking for parameters and return values using the general invocation mechanism would require variants of the mechanism to handle each permutation of parameter list and return type. This is possible but increases the verbosity of the shared class code because it requires customising the Method class' invocation interface for the actual parameter list and return type of each operation.

In this implementation, compile-time type checking is sacrificed in order to simplify the meta-layer implementation and implementation of the demonstration examples. However, the meta-layer performs run-time type checking on parameters to MethodInvocations and ConditionalInvocations.

4.3.6 Dynamic Method Lookup and Inheritance

The scheme implementation's conventional invocation syntax involves a level of indirection via a member field holding an instance of the desired Method object.

In the case of inheritance hierarchies of shared objects, when a subclass overrides an operation defined by a superclass, it provides an overriding Method class definition and a corresponding member field to refer to an instance of the overridden Method. The corresponding member field in the subclass has the same name as the member field in the superclass, and is called a shadowed variable. However, generalised clients that interact with the polymorphic class are built to work with the general base class. The Java™ language [AG97] defines differing semantics for shadowed variables versus shadowed (or overridden) methods. Overridden methods are resolved via runtime dynamic method lookup, while shadowed variables are resolved statically at compile time. Hence, using the conventional invocation syntax, a generalised client will always invoke the base class' Method operation, and never the subclass' overridden version, interfering with polymorphism.

This problem is a simple low-level implementation detail which arises because the scheme framework implements a different object model and invocation mechanism on top of a host language, compiler and runtime environment.
Chapter 4. Implementation Issues in Proof-of-Concept

However, the problem is easily avoided by declaring a simple member wrapper function to be used for method invocation.

For example,

```java
public class SomeSharedClass extends SharedObject {
    ...

    // declare Method 'Foo'
    public class Foo extends Method {
        ...
    }

    // declare and initialise member field 'foo'
    // to hold instance of 'Foo' method
    public Foo _foo = new Foo();

    // declare wrapper function to be used
    // for method invocation
    public void foo() { _foo.exec(); }
}
```

Clients may then use the standard syntax

```java
so.foo();
```

for invoking the operation, which ensures that the methods are resolved dynamically at runtime.

To eliminate excess verbosity, such wrapper functions are omitted in example code, except where required for correct operation of polymorphic shared classes. A native or ideal implementation of the scheme would not involve the member field indirection and hence would not suffer from this side-effect.

4.4 Invocation Scheduling Mechanism

The meta-layer's generalised invocation scheduling mechanism is implemented by a global singleton Scheduler object.

4.4.1 Scheduler Architecture

4.4.1.1 Architecture Overview

Figure 4.1 shows a high-level overview of the main data structures involved in the Scheduler object.
• **Scheduling Queue** – this ordered list stores references to all Invocation objects which cannot progress any further within the invocation sequence due to access conflict. The list is ordered according to the invocation ID assigned at the meta-layer entry point, to ensure fairness between competing invocations. When an invocation’s access table contains no conflict with any higher order invocations in the scheduling queue, it can be considered for activation and removal from this queue.

• **Execution List** – this list stores references to all Invocation objects which are currently executing their Method implementation functions. An invocation cannot be added to this list (i.e. begin execution) if its access table conflicts with the access tables of any of the invocations already in this list.

• **Execution Thread List** – this list stores references to the Thread objects driving root invocations that are currently in the execution list. This list is checked as each new Invocation object is introduced to the Scheduler. If the requesting thread is a member of this list then the new Invocation is a subinvocation of an already scheduled root invocation, and hence may execute immediately without any meta-layer processing.

• **Active Condition List** – this list stores references to all ConditionalInvocation objects which have access to test the condition on their target shared object. A conditional invocation must first pass through the scheduling queue to ensure fairness against competing invocations.

• **Active Consequent List** – this ordered list stores references to all scheduled MethodInvocation objects which are consequents of conditional invocations. Until the consequent method invocation actually begins execution, the meta-layer must monitor atomicity over the condition and the consequent method invocation.

• **Attention List** – this ordered list stores references to invocation objects which were blocked but have since been flagged as potential candidates for further progress in the invocation sequence. The list is ordered according to the invocation ID assigned at the meta-layer entry point, to ensure fairness between competing invocations.
4.4.2 Method Invocation Processing

The meta-layer's processing of method invocations can be grouped into the following six stages:

- Insertion
- Access Decomposition
- Scheduling
- Activation
- Execution
- Post-Processing
All stages of processing for a method invocation are not necessarily executed by the thread which requested the invocation. Initially, both the insertion processing and an attempt at access synchronisation are always performed by the requesting thread. If call chain finalisation is unsuccessful, or the invocation involves access conflict with other scheduled or executing invocations, then the invocation object is inserted into the scheduling queue and the thread abandons scheduling and jumps directly to blocking for execution activation. In such cases, the intermediary stages such as access decomposition, scheduling and execution activation are performed by other thread(s) in the post-processing of other invocations. The original thread then executes the requested operation on the target object, performs the post-execution processing for the invocation, before exiting the meta-layer.

4.4.2.1 Method Invocation Sequence

Each method invocation is processed by the Scheduler according to the following sequence of steps:

**INSERTION**

1. *Check for Nested Invocation* – if the thread object which is requesting this operation is an element of the current execution thread list, then this method invocation must be a nested subinvocation request from within a previously scheduled method invocation that is currently in its execution phase. The root invocation has already been scheduled once and so no further synchronisation or scheduling is necessary. The Method implementation function for this subinvocation is executed immediately and control and return value are returned.

   If the current thread object is not an element of the current execution thread list then the method invocation is a new root invocation and control proceeds to step 2.

2. *Assign Invocation ID* – the invocation is assigned a unique serial number from an invocation insertion counter.

3. *Insert into Scheduling Queue* – the invocation is inserted into the scheduling queue according to order of invocation ID. If the invocation ID was just assigned, then this position is always the queue’s tail. However, if the method invocation is the consequent of a conditional invocation, then it shares the same ID as its antecedent conditional invocation. If such a conditional invocation had to wait for some time before its condition became true, then the consequent
method invocation may be inserted accordingly at some point up to even the
head of the scheduling queue, according to its invocation ID.

ACCESS DECOMPOSITION

4. **Build Call Chain Model** – the access decomposition mechanism attempts to
build a finalised call chain model for the invocation. The attempt may be
unsuccessful if any call chain nodes encounter access conflict over member field
values.

5. **Build Access Table** – an access table is produced from the call chain graph.

SCHEDULING

6. **Check Finalisation** – if the call chain model was unable to be finalised then
conflict must have occurred over access to shared object member field values.
Although an initial access table has been generated, the scheduler cannot
proceed at this point with any further processing for this invocation. The
invocation must wait until the higher order invocation executes and releases
access to the member field of the contentious shared object. The invocation
object's status is set to `blocked_in_finalisation` and control jumps to step 9.

   • Note: If the method invocation becomes `blocked_in_finalisation` then the
     remaining portion of the access decomposition process, followed by
     invocation scheduling and subsequent execution activation for this
     invocation will be carried out by the scheduler during the post-execution
     processing of other invocations under other threads of control.

7. **Check Access Conflict** – the invocation compares its own access table for
conflict with those of the other invocations in the scheduling queue, starting
from its own position in the queue. For newly inserted method invocations, this
is the queue tail. If there are no conflicts with any invocations in the scheduling
queue, then the invocations in the execution list are checked as well.

   • If any conflict occurs then a reference to this invocation is stored as a
     release dependency in the winner invocation object. The invocation must
     wait until this higher order invocation finishes execution and releases
     access to the conflicting shared object(s). This invocation object's status
     is set to `blocked_in_scheduling` and control jumps to step 9. As in 6
     above, the remaining invocation scheduling and subsequent execution
     activation processing for this invocation will be carried out by the
Chapter 4. Implementation Issues in Proof-of-Concept

scheduler, during the post-execution processing of other invocations, under other threads of control

- If no conflict occurs then control proceeds to step 8.

**ACTIVATION**

8. *Execution Activation* – the invocation is readied for execution.

- *Move to Execution List* – the invocation object is removed from the scheduling queue and added to the execution list.

- *Check for Conditional Deactivation* – If this invocation contains write-access claims on any shared objects then the scheduler examines the active conditional list to see if any conditions are active for these write-claimed targets. If such conditional invocations are currently active then they must be deactivated and demoted to the scheduling queue.

- This invocation object's status is set to `execution_ready`.

- *Check for Antecedent Removal* – if this method invocation is a consequent, then we can remove the antecedent conditional invocation from the active condition list.

9. *Block for Execution Activation* – if the invocation object status is `execution_ready` then control proceeds immediately to step 10. However, if the invocation has some other status then the thread uses a synchronisation primitive to block until the invocation object has been updated to the required state. In such cases, the synchronisation primitive will be signaled by the scheduler, during the post-execution processing of other invocations, under other threads of control.

If any prior stage of this invocation's processing resulted in blocking, then its subsequent processing will have been performed, during the post-execution processing of other invocations, by other threads of control. Hence, this stage represents a rendezvous point, where the original requesting thread becomes active once again, in order to actually perform execution of the operation.

10. *Insert Thread into Execution Thread List* – the current thread object is added to the execution thread list, to assist in detecting subinvocations from this operation.

**EXECUTION**
11. **Execute Method Implementation Function** – the operation is executed atomically without requiring any further synchronisation, through the following chain of interaction:

   (a) **Call Invocation to Execute Method** – The Invocation object calls a Method object operation (b) to execute the implementation. The return value is stored in the invocation's ReturnValue object.

   (b) **Call Method to Execute Implementation** – The Method object uses reflection to invoke the implementation function with the invocation parameter list, via a method pointer stored in the corresponding MethodMetaInfo object. The return value object from the call to the implementation function is returned to (a)

**POST PROCESSING**

12. **Remove Thread from Execution Thread List** – the current thread object is removed from the execution thread list.

13. **Release Access Locks** – the scheduler performs the following steps to release the access locks held by the completed invocation.

   - **Remove from Execution List** – the invocation object is removed from the execution list.
   - **Notify Release Dependencies** – any release dependencies (other lower order invocation objects which previously encountered access conflict with this invocation) must now be notified. All such invocations have their status set to schedule_queue_attendable and they are inserted into the attention list according to their invocation ID.
   - **Dispose Call Chain Graph** – the invocation's call chain graph and component nodes are disposed, releasing any write access claims on shared object MemberFieldValueTable objects. This will notify any invocations waiting in access decomposition for access to those tables. The notified invocations will have their status set to finalisation_attendable and will be inserted into the attention list according to their invocation ID.

14. **Process Attention List** – at this stage the attention list will contain, in order of priority, any invocations which may now be able to potentially proceed in their invocation sequence. There are three categories of blocked invocation:
• **Blocked in Access Decomposition** – if the invocation’s status is *finalisation_attendable* then the scheduler can reattempt call chain finalisation (as in Step 5.), in order to build an access table.

If finalisation is still unsuccessful, then the invocations status is reset to *finalisation_blocked* and processing moves to the next invocation in the attention list.

If finalisation is successful, then that invocation’s scheduling processing is continued according to step 7.

• **Blocked in Schedule Queue** – if the invocation’s status is *schedule_queue_attendable* then the scheduler can compares its access table for conflict with those of the other invocations in the scheduling queue, starting from its own position in the queue (as in Step 7.).

If any conflict occurs then a new *release dependency* is created in the winner invocation object as in Step 7. This invocation object’s status is reset to *blocked_in_scheduling* and processing moves to the next invocation in the attention list.

If no conflict occurs then the invocation can be activated for execution as in Step 8.

• **Inactive Conditional Invocation** – if a conditional invocation’s status is *condition_attendable* then a method invocation has just released write access on the shared object target of this conditional invocation’s condition. The scheduler then compares the conditional invocation’s access table for conflict with those of the other invocations in the scheduling queue, starting from its own position in the queue (as in Step 7.).

If any conflict occurs then a new *release dependency* is created in the winner invocation object as in Step 7. This invocation object’s status is reset to *blocked_in_scheduling* and processing moves to the next invocation in the attention list.

If no conflict occurs then the conditional invocation can be activated as described in Step 6 of Section 4.4.2.2

15. **Return** – The method invocation exits the meta-layer scheduler and control is returned to the MethodInvocation object. The return value of the operation is stored in the invocation’s member ReturnValue object. The invocation
object unwraps this value and returns control and return value back to the caller.

4.4.3 Conditional Invocation Processing

The meta-layer's processing of conditional invocations can be grouped into seven stages:

- Insertion
- Scheduling
- Activation
- Evaluation
- Wait for True Condition Evaluation
- Consequent Processing
- Return

As with method invocations, all stages of processing for a conditional invocation are not necessarily executed by the thread which requested the invocation. Initially, both the insertion processing and scheduling are always performed by the requesting thread. If the conditional invocation cannot be activated immediately then the invocation object is inserted into the scheduling queue and the thread jumps directly to blocking until the condition has been evaluated true. In this case, the intermediary stages of scheduling, condition activation, condition evaluation, and consequent processing are performed by other thread(s) in the post-processing of other invocations. The original thread blocks on one of two conditions depending on whether or not there is a consequent method invocation. If there is a consequent method invocation then the original thread blocks until the consequent invocation attains execution_ready status. It then executes the consequent operation on the target object, before performing the post-execution processing and returning control. If there is no consequent, then the original thread blocks until the conditional invocation attains condition_true status, before returning control to the caller.

4.4.3.1 Conditional Invocation Sequence

The processing for conditional invocations is similar to that of method invocations, but with a few differences. In the meta-layer approach to condition synchronisation, conditional invocations are not allowed to be nested within Method operations, so the scheduler ensures that this can not occur. Conditional invocations do not require any
call chain modeling for access decomposition – they involve only a single read-mode access claim on the target shared object of the condition. Conditional invocations themselves do not involve any Post-Processing. However, if they have a consequent method invocation, then that method invocation will perform post-execution processing.

Conditional invocations are processed by the Scheduler according to the following sequence of steps.

**INSERTION**

1. **Ensure not Nested Conditional** – if the thread object which is requesting this operation is an element of the current execution thread list then this is a nested conditional subinvocation. The scheduler does not allow this, rejects the invocation and produces an error message;

   If the current thread object is not an element of the current execution thread list then control proceeds to step 2.

2. **Assign Invocation ID** – the invocation is assigned a unique serial number from an invocation insertion counter. If the conditional invocation has a consequent method invocation, then the method invocation is assigned the same serial number as well.

3. **Insert into Scheduling Queue** – the invocation is inserted into the scheduling queue in order of invocation ID. This position is always the queue’s tail.

**SCHEDULING**

4. **Build Access Table** – an access table is created containing a read-mode access claim on the condition target.

5. **Check Access Conflict** – the invocation compares its own access table for conflict with those of the other invocations in the scheduling queue, starting from its own position in the queue. If there are no conflicts with any invocations in the scheduling queue, then the invocations in the execution list are checked as well.

   - If any conflict occurs then a reference to this invocation is stored as a release dependency in the winner invocation object. The invocation must wait until the higher order invocation finishes execution and releases access to the conflicting shared object(s). This conditional invocation object’s status is set to blocked_in_scheduling and control jumps to step 8.
• If no conflict occurs then control proceeds to step 6.

**ACTIVATION**

6. **Condition Activation** – the conditional invocation object now has access to test the condition.

• **Move to Active Conditional List** – the conditional invocation object is removed from the scheduling queue and added to the active condition list.

**EVALUATION**

7. **Evaluate Condition** – the ConditionalInvocation object calls the Condition object to execute the condition implementation function and evaluate the condition.

• If the condition evaluates to false then the conditional invocations status is set to condition\_false and control jumps to step 8.

• If the condition evaluates to true then the conditional invocation is removed from the active conditional list and a check is performed to see if there is a consequent method invocation.

   If there is no consequent then the conditional invocation’s status is set to condition\_true and control jumps to step 8.

   If there is a consequent then it is inserted into the meta-layer for scheduling using the same invocation ID as its parent conditional invocation. The consequent’s processing follows the steps as described previously for method invocation processing.

   After the attempt at scheduling, if the consequent invocation’s status is found to be execution\_ready, then its parent conditional invocation status is set to consequent\_execution and control jumps to Step 8

   If the consequent invocation’s status is not execution\_ready, then its parent conditional invocation status is set to consequent\_scheduled and the conditional invocation is added to the active consequents list. Control jumps to Step 8.

**WAIT FOR TRUE CONDITION EVALUATION**

8. **Block for Notification** – The invocation uses a synchronisation primitive here to block for the required condition.
• If the conditional invocation has no consequent then the invocation blocks until its status is condition_true. When the primitive is signaled, control proceeds immediately to step 14.

• If the conditional invocation has a consequent then the invocation blocks until the consequent method invocation's status is execution_ready. When the primitive is signaled, control then proceeds immediately to step 9.

If any prior stage of this conditional invocation's processing resulted in blocking, then its subsequent processing will have been performed, during the post-execution processing of other invocations, by other threads of control. Hence, this stage represents a rendezvous point, where the original requesting thread becomes active once again, in order to return, or perform actual execution of the consequent operation.

CONSEQUENT PROCESSING

9. Insert Thread into Execution Thread List – the current thread object is added to the execution thread list, to assist in detecting subinvocations from the consequent operation.

10. Execute Consequent Method Implementation Function – the consequent operation is executed atomically as in step 11 of method invocation processing.

11. Remove from Execution Thread List – the current thread object is removed from the execution thread list.

12. Release Access Locks – the scheduler releases the access locks held by the completed consequent invocation, as described in step 13 of method invocation processing.

- Remove from Execution List
- Notify Release Dependencies
- Dispose Call Chain Graph

13. Process Attention List – the scheduler processes, in order of priority, any invocations which may now be able to proceed in their invocation sequence, as described in step 14 of method invocation processing.

- Invocations Blocked in Access Decomposition
- Invocations Blocked in Schedule Queue
- Inactive Conditional Invocations
14. **Return** – The conditional invocation exits the meta-layer scheduler and control is returned to the `ConditionalInvocation` object. If there was a consequent invocation then its return value is stored in the method invocation's member `ReturnValue` object. The `ConditionalInvocation` unwraps this value and returns control and return value back to the caller.

### 4.4.4 Intra-Scheduler Concurrency

The Scheduler object manages the synchronisation and execution of all invocations concurrently. The Scheduler object is not threaded itself, its activities and control code are executed by the independent threads requesting operations on shared objects.

The Scheduler uses a monitor approach for synchronising the execution of concurrent internal threads. Figure 4.2 shows the various stages of progress of method invocations within the meta-layer. Two stages of the invocation sequence, `insertion` processing and `post-execution` processing require obtaining the scheduler's monitor to ensure mutually exclusive access to the scheduler's data structures.

The scheduler overrides the host language's non-deterministic native thread scheduling scheme by implementing a fair protocol for monitor acquisition, according to the invocation ordering rule.

In the case where multiple threads are waiting to obtain the monitor to perform insertion or post-execution processing, the Scheduler grants the monitor first to the invocations requiring post-execution processing, in order of execution completion. If there are no invocations requiring post-execution processing, then the monitor is awarded to the invocation according to the invocation ordering rule (i.e. the invocation with the lowest id, and hence which entered the meta-layer first).
Chapter 4. Implementation Issues in Proof-of-Concept

<table>
<thead>
<tr>
<th>Enter Scheduler</th>
<th>concurrent</th>
</tr>
</thead>
<tbody>
<tr>
<td>Assign Invocation Id</td>
<td>concurrent</td>
</tr>
<tr>
<td>Insertion Processing</td>
<td>concurrent</td>
</tr>
<tr>
<td>• attempt call chain finalisation</td>
<td></td>
</tr>
<tr>
<td>• build access table</td>
<td></td>
</tr>
<tr>
<td>• insert into schedule queue</td>
<td></td>
</tr>
<tr>
<td>&lt; finalised ? &gt;</td>
<td></td>
</tr>
<tr>
<td>• check for access conflict</td>
<td></td>
</tr>
<tr>
<td>• if no conflict -&gt; activate</td>
<td></td>
</tr>
<tr>
<td>&lt; finalised ? &gt;</td>
<td></td>
</tr>
<tr>
<td>• check for access conflict</td>
<td></td>
</tr>
<tr>
<td>• if no conflict -&gt; activate</td>
<td></td>
</tr>
</tbody>
</table>

**Monitor ensures mutual exclusion between insertion processing of concurrent invocations. Monitor is granted in order of invocation id.**

<table>
<thead>
<tr>
<th>Wait for Execution</th>
<th>concurrent</th>
</tr>
</thead>
<tbody>
<tr>
<td>Execute Method</td>
<td>concurrent</td>
</tr>
<tr>
<td>Post Execution Processing</td>
<td></td>
</tr>
<tr>
<td>&lt; this invocation &gt;</td>
<td></td>
</tr>
<tr>
<td>• release access locks</td>
<td></td>
</tr>
<tr>
<td>• release member field access locks</td>
<td></td>
</tr>
<tr>
<td>&lt; other invocations &gt;</td>
<td></td>
</tr>
<tr>
<td>• process Attention List</td>
<td></td>
</tr>
</tbody>
</table>

**METHOD INVOCATIONS**

| < finalisation blocked? > |
| • attempt call chain finalisation |
| • build access table  |
| < finalised? >       |
| • check for access conflict |
| • if no conflict -> activate |

**Monitor ensures mutual exclusion between post-execution processing of concurrent invocations. Monitor is granted in order of invocation id. Invocations requiring post-execution processing are granted the monitor before any invocations requiring insertion processing.**

<table>
<thead>
<tr>
<th>Exit Scheduler</th>
<th>concurrent</th>
</tr>
</thead>
</table>

**Figure 4.2 Intra-Scheduler Concurrency Scheme**
Chapter 5

Demonstration Examples

This chapter provides example solutions to various classical concurrency problems in order to demonstrate the use of the proof-of-concept implementation framework. Section 5.1 explains a solution to the Dining Philosophers problem based solely on access synchronisation. Section 5.2 introduces simple condition synchronisation using the Cigarette Smokers problem. Section 5.3 describes a more complicated example of condition synchronisation involving rendezvous based on the Sleeping Barber problem. Section 5.4 revisits the Cigarette Smokers problem showing the ease with which additional active participants can be added to an existing system. Section 5.5 revisits the Dining Philosophers problem to illustrate the use of generalisation and polymorphic shared classes.

The example solutions provided here all conform to a general structure. The new components defined can be divided into two categories: shared objects and normal objects.

- **Shared Objects** – these classes inherit from the `SharedObject` abstract base class provided by the framework. All objects which may be concurrently accessed by more than one independent thread of control are defined in this way. These classes define concrete inner subclasses of `Method` and `Condition` to implement the various operations and condition functions required to solve the problem. Composite operations defined by shared objects are guaranteed to execute with the atomicity and isolation properties extending over their complete duration. In order to prevent potential interference from other threads, an object which is only accessed by one thread may be defined as a shared object so that its operation's execute with the access protection provided by the meta-layer.
Normal Objects – all other 'non-shared' objects are considered as 'normal' objects, including the existing classes provided by the core language libraries. Operations on these classes operate outside the control of the synchronisation meta-layer. In the example solutions provided, the new 'normal' objects defined consist of the requisite 'main' application classes, as well as threaded objects which provide a thread of control and describe the active behaviour of the concurrent tasks within a system. These threaded objects make invocations on shared objects which are synchronised by the meta-layer. However, there is no atomicity over sequences of invocations made by 'normal' objects. Each invocation by a normal object on a shared object is a separate root invocation – with no guarantee of interference between separate root invocations. Hence operations on shared objects must be designed in such a way as to provide atomicity of appropriate granularity in order to prevent interference between the activities of competing threads.
Chapter 5. Demonstration Examples

5.1 The Dining Philosophers Problem

The Dining Philosophers problem [Dij72] provides a simple illustration of multiple processes competing for specific subsets of a limited resource. The Dining Philosophers problem is often used to demonstrate the complexities associated with designing concurrent programs. The example solution shows how the meta-level approach can be used to fairly, safely and automatically resolve competition between competing independent activities, without requiring any explicit coordination mechanisms within the implementation code of the components of the system.

5.1.1 Problem Description

Five philosophers sit around a circular table, as shown in Figure 5.1. Each philosopher spends his time alternatively thinking and eating, independently of the others. In the centre of the table is a large platter of spaghetti. Because the spaghetti is long and tangled – and the philosophers are not mechanically adept – a philosopher must use two forks to eat a helping. Unfortunately, the philosophers have only five forks at their disposal. One fork is placed between each philosopher, and they agree that each will use only the forks to their immediate left and right.

Figure 5.1 Dining Philosophers Problem
When one philosopher wants to eat, he picks up the fork on his left, when it is available, and then the fork on his right and then proceeds to eat for some amount of time. When he is done eating, he puts down both forks. If the forks are unavailable, then the philosopher must wait until they become available.

The goal is to write a program to simulate the behaviour of the five philosophers.

The program must avoid two undesirable situations:

a) Starvation – an unfair situation where one philosopher remains hungry and unable to eat, while a favored philosopher continues to alternatively think and eat.

b) Deadlock – an unrecoverable situation where no philosopher is able to acquire both forks and all progress in the system halts. i.e. each philosopher holds one fork and refuses to give it up, while waiting to acquire the other respective fork.

5.1.2 Analysis

Solving the Dining Philosophers problem requires coordinating access to the shared forks in a safe and fair manner. In order to eat, each philosopher requires exclusive access to both the fork on his left and on his right. If the forks are defined as shared objects, then the eat operation may be modeled as a composite multi-object operation – and hence be protected by the atomicity and isolation properties, and safe and fair scheduling properties provided by the meta-layer synchronisation scheme.

5.1.3 Example Solution

5.1.3.1 Overview

The example solution is composed of two 'shared' classes, and two 'normal' classes:

Shared Classes:

• Philosopher – a class for representing philosophers sitting at the table
• Fork – a class for representing the forks laid out on the table

Normal Classes:

• PhilosopherThread – a threaded class which actively drives a philosopher's activities
• DinersApp – the main application class
Chapter 5. Demonstration Examples

Figure 5.2 provides an illustration of the structure of the solution.

![Structure of Example Solution to Dining Philosophers Problem](image)

5.1.3.2 Dining Philosophers Source Code

Figure 5.3 shows the Java source code listing for the Philosopher shared object class. Each Philosopher is associated with two instances of the Fork class, for its left and right forks respectively. It defines two Method operations, think and eat.

```java
//
// Philosopher.java
//
import java.util.*;
import synch.*;

public class Philosopher extends SharedObject {
    public Fork leftFork, rightFork;
    private int eatCount;
    public Eat eat = new Eat();
    public Think think = new Think();

    private final static int MAX_DELAY = 2000;
    private static Random random = new Random();

    public Philosopher(String name, Fork left, Fork right) {
        super(name);
        leftFork = left;
        rightFork = right;
        eatCount = 0;
        metainitialise();
    }

    public class Think extends Method {
        public void impl() {
            int time = Math.abs((random.nextInt() & MAX_DELAY ));
        }
    }
}
```
The think operation simulates the philosopher's thinking cycle by sleeping for a random period of time. The operation does not involve any shared object interaction, and so does not require a meta-information declaration. The subinvocations to the standard Java library functions are thread-safe and do not result in any further interaction on shared objects and so are considered outside the scope of the synchronisation scheme. By default, the operation inherits a 'read' access mode on its target instance.

The eat operation simulates the philosopher's eating cycle by picking up both forks, sleeping for a random period of time, and then putting down the forks. Because the
Chapter 5. Demonstration Examples

operation involves subinvocations on shared objects, a meta-information function is required. There are four subinvocations, pickup and putDown on the respective leftFork and rightFork member field references. The pickup invocations also pass a parameter – a reference to the Philosopher who is picking up the fork. The Eat Method uses the mixin interface Mutative to declare a 'write' access mode on its target.

Figure 5.4 shows the Java source code listing for the Fork shared object class. Each Fork maintains an association with an instance of the Philosopher class, for keeping track of its current holder. It defines two Method operations, pickup and putDown. The pickup operation accepts a Philosopher object as a parameter and stores the reference in its holder field. This action requires a corresponding meta-information declaration.

The pickup Method uses the Mutative interface to declare a 'write' access mode on its target.

The putDown operation resets the holder reference field back to null. As this does not introduce any new reference values, no explicit meta-information is required.

As both operations are declared with 'write' mode target access, any composite operation defined by another shared class, which involves subinvocations upon either of these Fork operations, will require exclusive access to the corresponding target Fork instances for the complete duration of the composite operation execution.

/\  // Fork
//
import synch.*;

public class Fork extends SharedObject {
    public Philosopher holder;
    public PickUp pickUp = new PickUp();
    public PutDown putDown = new PutDown();

    public Fork(String name) {
        super(name);
        holder = null;
        metaInitialise();
    }

    public class PickUp extends Method implements Mutative {
        public void meta() {

    }
Chapter 5. Demonstration Examples

Figure 5.4 Fork.java Source Code Listing

generate("p");
assignMemberFld(memberFld("holder"), parameter("p"));

public void impl(Philosopher p) {
    holder = p;
}

public class PutDown extends Method implements Mutative {
    public void impl() {
        holder = null;
    }
}

Figure 5.5 shows the Java source code listing for the PhilosopherThread class. Each PhilosopherThread has its own independent thread of control which drives the activities of its particular Philosopher instance.

The PhilosopherThread loops continuously, alternatively invoking the think and eat operations on its particular Philosopher instance.

Figure 5.5 PhilosopherThread.java Source Code Listing

Figure 5.6 shows the Java source code listing for the DinersApp class. The DinersApp class constructs and appropriately initialises the required instances of
Fork, Philosopher, and PhilosopherThread. The start() method then starts execution of each PhilosopherThread.

```java
// DinersApp
//
import java.util.*;
import synch.*;

public class DinersApp {
    final static int SIZE = 5;
    Vector philosophers;
    Vector forks;
    Vector threads;

    public DinersApp() {
        philosophers = new Vector(SIZE);
        forks = new Vector(SIZE);
        threads = new Vector(SIZE);

        for(int i=0;i<SIZE;i++) {
            Fork f = new Fork("Fork" + i);
            forks.addElement(f);
        }

        for(int i=0;i<SIZE;i++) {
            Fork left = (Fork) forks.elementAt(i);
            Fork right = (Fork) forks.elementAt((i+SIZE-1)%SIZE);
            Philosopher p = new Philosopher("Phil"+i,left,right);
            philosophers.addElement(p);
            PhilosopherThread dt = new PhilosopherThread(p);
            threads.addElement(dt);
        }
    }

    public void start() {
        PhilosopherThread t;
        for(int i=0;i<SIZE;i++) {
            t = (PhilosopherThread) threads.elementAt(i);
            t.start();
        }
    }

    public static void main(String[] args) {
        DinersApp simulation = new DinersApp();
        simulation.start();
    }
}
```

Figure 5.6  DinersApp.java Source Code Listing
5.1.3.3 Execution Output

Figure 5.7 shows the output produced from the execution of the DinersApp program.

```
Philo begins thinking for 1246 ms.
Phili begins thinking for 872 ms.
Phil2 begins thinking for 1963 ms.
Phil3 begins thinking for 326 ms.
Phil4 begins thinking for 630 ms.
Phil3 finishes thinking.
Phil3 begins thinking for 434 ms.
Phil1 finishes thinking.
Phil1 begins thinking for 326 ms.
Phil1 finishes thinking.
Phil1 starts eating plate no. 1 for 804 ms.
Phil14 finishes thinking.
Phil4 begins thinking for 656 ms.
Phil3 begins thinking for 630 ms.
Phil10 finishes thinking.
Phil11 finishes thinking.
Phil13 finishes thinking.
Phil14 finishes thinking.
Phil10 starts eating plate no. 1 for 609 ms.
Phil13 starts eating plate no. 2 for 1739 ms.
Phil14 begins thinking for 1014 ms.
Phil12 finishes thinking.
Phil10 finishes thinking.
Phil1 begins eating plate no. 2 for 519 ms.
Phil10 begins thinking for 1995 ms.
Phil14 finishes thinking.
Phil11 finishes eating.
Phil1 begins thinking for 1671 ms.
Phil3 finishes eating.
Phil2 begins eating plate no. 1 for 876 ms.
Phil14 begins eating plate no. 2 for 232 ms.
Phil13 begins thinking for 704 ms.
Phil14 finishes eating.
Phil14 begins thinking for 1769 ms.
Phil13 finishes thinking.
Phil10 begins thinking.
Phil10 begins eating plate no. 2 for 75 ms.
Phil12 finishes eating.
Phil13 starts eating plate no. 3 for 1358 ms.
Phil12 begins thinking for 242 ms.
Phil10 finishes eating.
Phil10 begins thinking for 1851 ms.
Phil11 finishes thinking.
Phil11 starts eating plate no. 3 for 388 ms.
```

Figure 5.7 DinersApp Execution Output

5.1.4 Discussion

When a PhilosopherThread object invokes the eat operation on its philosopher, the eat Method creates an Invocation object which is processed by the Scheduler. The access decomposition mechanism examines the meta-information for the eat operation and produces an access table as shown in Table 5.1. Firstly, it will require
exclusive access to the target Philosopher object. Secondly, using the object references in the leftFork and rightFork member fields, and the meta-information defined for Fork.pickup and Fork.putDown, it determines that exclusive access is also required on both Fork objects.

<table>
<thead>
<tr>
<th>Shared Object</th>
<th>Access Mode</th>
</tr>
</thead>
<tbody>
<tr>
<td>Philosopher</td>
<td>write</td>
</tr>
<tr>
<td>leftFork</td>
<td>write</td>
</tr>
<tr>
<td>rightFork</td>
<td>write</td>
</tr>
</tbody>
</table>

Table 5.1 Access Table for an Invocation of Philosopher.eat.

The Invocation is then scheduled for execution against competing concurrent invocations by the general invocation scheduling mechanism. Deadlock and interference are avoided because the scheduler ensures safe lock acquisition. Starvation is avoided because the scheduler imposes a FIFO order on conflicting concurrent operations.

The example solution contains redundant features which are not necessary for correct synchronisation of the problem. Each Fork object does not strictly need to store a reference to its current holder. Also, the Fork.putDown operation is redundant, and included only for symmetry. There are two key features to the correct operation of the solution. Firstly, the Philosopher.eat operation is a composite operation protected by the atomicity and isolation properties provided by the meta-layer synchronisation scheme. Secondly, the Fork operations are defined as requiring 'write' mode on their target instances. The meta-layer automatically arranges safe and fair acquisition of the required objects and access modes on behalf of the invocation before beginning execution of each Philosopher.eat operation.

The solution solves the problem solely via the meta-layer's safe and fair mechanism for access synchronisation. There is an assumption that all clients of the Fork objects are Philosopher objects who will pick up and then put down a fork in one atomic action. Hence a Fork object in use by one philosopher is never accessible to another philosopher while the Fork is in the 'picked up' state. If required, a more elaborate solution allowing such intermediate states could be devised using condition synchronisation and philosopher actions of smaller granularity.

The Dining Philosophers problem is extended in Section 5.5, where a 'dexterous' philosopher - who possesses the rare skill of being able to eat with only one fork - is
introduced and shares the table, without modifying the behaviour of the normal philosophers.
5.2 The Cigarette Smokers Problem

The Cigarette Smokers problem [Pat71, Par78] is a single producer / multiple consumers problem which involves simple condition-based synchronisation. The example solution shows how condition synchronisation can be achieved using the meta-level approach. State-based conditions – used to control independently threaded tasks – are tested fairly, safely and automatically by the meta-layer, without requiring any explicit coordination mechanisms within the implementation code of the components of the system.

5.2.1 Problem Description

Suppose there are three smoker processes and one agent process, as illustrated in Figure 5.8. Each smoker continuously makes a cigarette and smokes it. Making a cigarette requires three ingredients: tobacco, paper, and a match. One smoker process has tobacco, the second paper, and the third matches. Each smoker has an infinite supply of his individual ingredient. The agent has an infinite supply of all three ingredients. The agent randomly selects two ingredients and places them on the table. The smoker who has the missing third ingredient picks up the other two, makes a cigarette, then smokes it. The agent waits for the smoker to finish. The cycle then repeats.

Figure 5.8 Cigarette Smokers Problem
5.2.2 Analysis

This simple producer/consumers problem does not contain any inherent potential for parallelism, but instead amounts to the coordination of four independent tasks sharing a common resource – the state of the ingredients on table.

Each of the three independent smoker tasks waits on the condition that the table has the exact two ingredients that it requires. It then consumes the ingredients and waits again for the same condition.

The agent task waits on the condition that the table has no ingredients. It places two random ingredients back on the table, and waits again for the same condition.

These requirements can be modeled via a shared class Table with two Method operations and two Conditions.

Method Operations:

- `PlaceIngredients` – to place two new ingredients on the table. This operation will be called by the agent.

- `RemoveIngredients` – to remove the two ingredients currently on the table. This operation will be called by the smokers.

Conditions:

- `NoIngredients` – true if there are no ingredients currently on the table.

- `MissingIngredient` – accepts an ingredient as a parameter and returns true if this ingredient is the only ingredient currently missing from the table. Does not return true if the table currently has no ingredients.

5.2.3 Example Solution

5.2.3.1 Overview

The example solution is composed of three 'shared' classes, and three 'normal' classes:

Shared Classes:

- Smoker – a class for representing a smoker
- Agent – a class for representing the agent
- Table – a class for representing the table

Normal Classes:
• SmokerThread – a threaded class which actively drives a smokers activities
• AgentThread – a threaded class which actively drives an agents activities
• SmokersApp – the main application class

Figure 5.9 provides an illustration of the structure of the solution.

![Diagram of Example Solution to Cigarette Smokers Problem](image)

Figure 5.9 Structure of Example Solution to Cigarette Smokers Problem

### 5.2.3.2 Smokers Problem Source Code

The three smokers are named Tom, Pat and Mat. Tom has an infinite supply of tobacco, Pat has paper, and Mat has matches. The ingredient types are defined as constant String objects by the SmokersApp class.

Figure 5.10 shows the Java source code listing for the Table shared object class. The Table has two private fields for storing the current ingredients. It defines two Method operations, `placeIngredients` and `removeIngredients`. It also defines two Conditions, `noIngredients` and `missingIngredient`.

Both Method operations are declared as Mutative, which ensures that they execute in mutual exclusion.

The ingredient parameters to the `placeIngredients` operation are references to Java string objects which are themselves, by definition, immutable. Hence, there can be no access synchronisation problems in operations involving them.
import synch.*;

public class Table extends SharedObject {

    private String ingredient1;
    private String ingredient2;

    public NoIngredients
        noIngredients = new NoIngredients();

    public MissingIngredient
        missingIngredient = new MissingIngredient();

    public RemoveIngredients
        removeIngredients = new RemoveIngredients();

    public PlaceIngredients
        placeIngredients = new PlaceIngredients();

    public Table() {
        super();
        ingredient1 = SmokersApp.NO_INGREDIENT;
        ingredient2 = SmokersApp.NO_INGREDIENT;
        metaInitialise();
    }

    public class NoIngredients extends Condition {
        public boolean cond() {
            if ( ingredient1 == SmokersApp.NO_INGREDIENT )
                return true;
            else
                return false;
        }
    }

    public class MissingIngredient extends Condition {
        public boolean cond(String i) {
            if ( ingredient1 == i )
                return true;
            else
                return false;
        }
    }

    public class RemoveIngredients extends Method implements Mutative {
        public void impl() {
            ingredient1 = SmokersApp.NO_INGREDIENT;
            ingredient2 = SmokersApp.NO_INGREDIENT;
        }
    }

    public class PlaceIngredients extends Method implements Mutative {

public void impl(String il, String i2) {
    System.out.println(".. ingredients " + il + " and " + i2 + " are put down on table.");
    ingredient1 = il;
    ingredient2 = i2;
}

Figure 5.10 Table.java Source Code Listing

Figure 5.11 shows the Java source code listing for the Agent shared object class. The Agent has a reference to the shared Table object in addition to its list of three ingredient types. It defines only one Method operation, placeRandomIngredients.

The placeRandomIngredients operation generates two different random ingredients and then places them on the table via the Table.placeIngredients subinvocation. Accordingly, the meta-information for the placeRandomIngredients operation declares the details of the subinvocation.

```java
public class Agent extends SharedObject {
    private String[] ingredients = new String[3];
    private Random random;

    public PlaceRandomIngredients placeRandomIngredients = new PlaceRandomIngredients();

    public Agent(Table t) {
        super();
        table = t;
        random = new Random();
        ingredients[0] = SmokersApp.TOBACCO;
        ingredients[1] = SmokersApp.PAPER;
        metaInitialise();
    }

    public class PlaceRandomIngredients extends Method {
        public void meta() {
            subinvocation(memberFld("table"), "placeIngredients");
        }
    }
```
public void impl() {
    // generate two random ingredients
    int i1 = Math.abs((random.nextInt() % 3));
    int i2 = i1;
    while (i2==i1) {
        i2 = Math.abs((random.nextInt() % 3));
    }
    table.placeIngredients.exec(ingredients[i1], ingredients[i2]);
}

Figure 5.11  Agent.java Source Code Listing

Figure 5.12 shows the Java source code listing for the AgentThread class. The AgentThread possesses its own independent thread of control which drives the activities of the Agent instance.

//
//  AgentThread
//
import synch.*;
public class AgentThread extends Thread {
    private Agent agent;
    private Table table;
    public AgentThread(Agent a, Table t) {
        super();
        agent = a;
        table = t;
    }
    public void run() {
        for(;;) {
            table.noIngredients.execTrue(
                agent.placeRandomIngredients.makeInv());
        }
    }
}

Figure 5.12  AgentThread.java Source Code Listing

The AgentThread loops continuously, invoking the following conditional invocation:

table.noIngredients.execTrue(
    agent.placeRandomIngredients.makeInv());
Chapter 5. Demonstration Examples

This statement is composed of two parts: the conditional invocation `table.noIngredients.execTrue()`, and its parameter, the consequent method invocation `agent.placeRandomIngredients.makeInv()`. The metalayer waits if necessary for the condition to be true, and then executes the consequent method invocation, under the atomicity of the true condition.

Figure 5.13 shows the Java source code listing for the Smoker shared object class. The Smoker has a reference to the shared Table object in addition to its name and ingredient type. It defines two Method operations, `getIngredient` and `takeRollSmoke`.

The `getIngredient` operation returns the ingredient type that this smoker possesses. It is used by the SmokerThread class.

The `takeRollSmoke` operation simulates the smoker's action of taking the ingredients off the table, rolling a cigarette and then smoking it. This involves a `removeIngredients` subinvocation on the shared Table object. Accordingly, the meta-information for the `takeRollSmoke` operation declares the details of the subinvocation. This operation is called by the SmokerThread when the appropriate conditions are met.

```java
//
// Smoker
//
import synch.*;
public class Smoker extends SharedObject {
    public Table table;
    private String name;
    private String ingredient;

    public GetIngredient getIngredient = new GetIngredient();
    public TakeRollSmoke takeRollSmoke = new TakeRollSmoke();

    public Smoker(String s, String i, Table t) {
        super();
        name = s;
        ingredient = i;
        table = t;
        metaInitialise();
    }

    public class GetIngredient extends Method {
        public String impl() {
```
Chapter 5. Demonstration Examples

```java
public class TakeRollSmoke extends Method implements Mutative {
    public void meta() {
        subinvocation(memberFld("table"), "removeIngredients");
    }
    public void impl() {
        table.removeIngredients.exec;
        System.out.println("— " + name + " takes ingredients then rolls and smokes a cigarette.");
    }
}
```

Figure 5.13 Smoker.java Source Code Listing

Figure 5.14 shows the Java source code listing for the SmokerThread class. Each SmokerThread possesses its own independent thread of control which drives the activities of its Smoker instance.

```java
// SmokerThread
import synch.*;
public class SmokerThread extends Thread {
    private Smoker smoker;
    private Table table;

    public SmokerThread(Smoker s, Table t) {
        super();
        smoker = s;
        table = t;
    }

    public void run() {
        String myIngredient = (String) smoker.getIngredient.exec();
        for(;;) {
            table.missingIngredient.execTrue(
                myIngredient, smoker.takeRollSmoke.makeInv());
        }
    }
}
```

Figure 5.14 SmokerThread.java Source Code Listing
The `SmokerThread` loops continuously, invoking the following conditional invocation:

```java
Table missingIngredient = new Table();
Agent agent = new Agent(missingIngredient);
AgentThread agentThread = new AgentThread(agent, missingIngredient);
```

This statement is composed of two parts: the conditional invocation `Table.missingIngredient.execTrue()`, the ingredient parameter `myIngredient`, and the consequent method invocation `Agent.smoker.takeRollSmoke.makeInv()`. The meta-layer waits if necessary for the condition to be true, and then executes the consequent method invocation, under the atomicity of the true condition.

Figure 5.15 shows the Java source code listing for the `SmokersApp` class. The `SmokersApp` class constructs and appropriately initialises the required instances of `Table`, `Agent`, `Smoker`, `AgentThread` and `SmokerThread`. The `start()` method then starts execution of the `AgentThread` and each `SmokerThread`.

```java
import synch.*;

public class SmokersApp {

    public final static String NO_INGREDIENT = new String("None");
    public final static String TOBACCO = new String("Tobacco");
    public final static String PAPER = new String("Paper");
    public final static String MATCH = new String("Match");

    protected Smoker torn, pat, mat;
    protected Agent agent;
    protected Table table;
    protected SmokerThread tomThread, patThread, matThread;
    protected AgentThread agentThread;

    public SmokersApp() {
        super();
        table = new Table();
        tom = new Smoker("Tom", TOBACCO, table);
        pat = new Smoker("Pat", PAPER, table);
        mat = new Smoker("Mat", MATCH, table);
        tomThread = new SmokerThread(tom, table);
        patThread = new SmokerThread(pat, table);
        matThread = new SmokerThread(mat, table);
        agent = new Agent(table);
        agentThread = new AgentThread(agent, table);
    }

    // SmokersApp
}
```
Chapter 5. Demonstration Examples

```java
public void start() {
    tomThread.start();
    patThread.start();
    matThread.start();
    agentThread.start();
}

public static void main(String[] args) {
    SmokersApp simulation;
    simulation = new SmokersApp();
    simulation.start();
}
```

Figure 5.15 SmokersApp.java Source Code Listing

### 5.2.3.3 Execution Output

Figure 5.16 shows the output produced from the execution of the SmokersApp program:

- ingredients Paper and Match are put down on table.
  -- Tom takes ingredients then rolls and smokes a cigarette.
- ingredients Match and Paper are put down on table.
  -- Tom takes ingredients then rolls and smokes a cigarette.
- ingredients Match and Tobacco are put down on table.
  -- Pat takes ingredients then rolls and smokes a cigarette.
- ingredients Paper and Match are put down on table.
  -- Tom takes ingredients then rolls and smokes a cigarette.
- ingredients Paper and Tobacco are put down on table.
  -- Mat takes ingredients then rolls and smokes a cigarette.
- ingredients Paper and Match are put down on table.
  -- Tom takes ingredients then rolls and smokes a cigarette.
- ingredients Paper and Tobacco are put down on table.
  -- Pat takes ingredients then rolls and smokes a cigarette.
- ingredients Paper and Tobacco are put down on table.
  -- Mat takes ingredients then rolls and smokes a cigarette.
- ingredients Paper and Tobacco are put down on table.
  -- Mat takes ingredients then rolls and smokes a cigarette.
- ingredients Tobacco and Paper are put down on table.
  -- Pat takes ingredients then rolls and smokes a cigarette.
- ingredients Tobacco and Paper are put down on table.
  -- Mat takes ingredients then rolls and smokes a cigarette.
- ingredients Tobacco and Match are put down on table.
  -- Pat takes ingredients then rolls and smokes a cigarette.
- ingredients Tobacco and Paper are put down on table.
  -- Mat takes ingredients then rolls and smokes a cigarette.
5.2.4 Discussion

The AgentThread object makes a conditional invocation based on the noIngredients condition of the Table. The meta-layer coordinates safe and fair testing of the condition – the condition can not be tested while another thread has write access on the condition target. When the condition becomes true, the meta-layer attempts scheduling of the consequent placeRandomIngredients Method on its agent. The consequent invocation will require the access table shown in Table 5.2, providing mutually exclusive access on the table during execution of the composite operation.

<table>
<thead>
<tr>
<th>Shared Object</th>
<th>Access Mode</th>
</tr>
</thead>
<tbody>
<tr>
<td>Agent</td>
<td>read</td>
</tr>
<tr>
<td>Table</td>
<td>write</td>
</tr>
</tbody>
</table>

Table 5.2 Access Table for Invocation placeRandomIngredients.

The SmokerThread object makes a conditional invocation based on the missingIngredient Condition of the table. Again, the meta-layer coordinates safe and fair testing of the condition. When the condition becomes true, the meta-layer attempts scheduling of the consequent takeRollSmoke Method on its smoker. The consequent invocation will require the access table shown in Table 5.3, providing mutually exclusive access on the table during execution of the composite operation.

<table>
<thead>
<tr>
<th>Shared Object</th>
<th>Access Mode</th>
</tr>
</thead>
<tbody>
<tr>
<td>Smoker</td>
<td>read</td>
</tr>
</tbody>
</table>
Chapter 5. Demonstration Examples

<table>
<thead>
<tr>
<th>Table</th>
<th>write</th>
</tr>
</thead>
</table>

Table 5.3 Access Table for Invocation `takeRollSmoke`.

The four independent tasks all wait for different conditions on the same target. The meta-layer ensures safe and fair testing of conditions – conditions are tested only when the meta-layer releases write access to the condition target (in this case the table) at the end of execution of root invocations (in this case the consequent operations). Inter-thread coordination is handled by the meta-layer, without requiring any explicit coordination mechanism in the component objects of the system.

The meta-level approach allows the use of a concrete abstraction to represent a particular state-based condition on a shared object. The conditional behaviour and activities of independent tasks are programmed in terms of these abstractions, rather than primitive synchronisation mechanisms which represent the conditions (such as mutexes or semaphores). This simplifies the programmers task of implementing such behaviour, and greatly reduces opportunities for safety and liveness failure due to low-level interference between independent threads.
5.3 The Sleeping Barber

The Sleeping Barber problem [Dij68, And91] is another classical synchronisation problem. The problem illustrates the client/server relationship between independent tasks as well as another important form of condition synchronisation called a rendezvous.

5.3.1 Problem Description

An easy-going town contains a small barber shop having two rooms and a some waiting chairs, as illustrated in Figure 5.17. There are three doors: from the street to the waiting room, from the waiting room to the barber's chair, and from the barber's chair back out to the street. The doors are assumed to be narrow and allow at most one person to pass at a time.

Customers enter through one door and leave through the other door. The barber spends his life serving customers, one at a time. When none are in the shop, the barber goes to sleep. When a customer arrives and finds the barber sleeping, the customer awakens the barber, sits down in the barber's chair, and sleeps while the barber cuts his hair. If the barber is busy when a customer arrives, the customer goes to sleep in one of the waiting chairs, and waits until it is his turn for a haircut. If there are no vacant waiting
chairs then the customer will not bother waiting for a haircut and will leave immediately. After receiving a haircut, the customer vacates the barber chair and exits the room. If there are waiting customers, the barber then awakens one and waits for the customer to sit in the barber's chair. Otherwise, the barber goes back to sleep until a new customer arrives.

5.3.2 Analysis

This problem involves more conditions than the Cigarette Smokers problem, and also involves a rendezvous, which occurs between the customer and the barber. The barber is modeled as an active process which consumes customers, and the waiting chairs form a bounded buffer in which customers can wait for their rendezvous with the barber. The customers are also active processes, who decide whether they will wait in the waiting room, or exit immediately without rendezvous with the barber for a haircut.

The barber's behaviour can be described by the following pseudocode:

```pseudocode
loop forever {
    wait till (new customer in barber chair)
    cut new customers hair
}
```

The customer's behaviour can be described by the following pseudocode:

```pseudocode
look for spare seat in waiting chairs *
if (there is a spare seat) {
    occupy spare waiting chair *
    wait till (barber chair vacant) *
    vacate waiting chair #
    occupy barber chair #
    wait till (hair has been cut)
    vacate barber chair
    exit
} else {
    exit without haircut
}
```

It is important to note that there exists synchronisation dependencies between some of the steps listed above. For instance, two independent customers should not simultaneously look for a spare seat and then both attempt to occupy it, because of the resulting interference. Hence a customer requires atomicity over the steps which represent looking for a spare seat, and then occupying a potential spare seat if it is available (each of these steps is marked with an asterisk "*", above).
Chapter 5. Demonstration Examples

In the same way, customers currently occupying a waiting chair, who are waiting for the barber chair to become vacant in order to move to it, require atomicity over those steps to prevent interference from other customer seeking to do the same (each of these steps is marked with a cross-hatch ‘#’, above). In addition, for fairness, customers should wait in order for their turn to move to the barber chair.

The meta-layer's approach to synchronisation allows designs which fulfill these requirements implicitly without requiring the explicit use of low-level synchronisation primitives to achieve coordination. Further, the approach allows the behaviour of independent tasks to be specified separately and individually, without taking explicit measures to coordinate their own behaviour with other potential activities within the system. The generalised meta-layer prevents any interference and ensures safety and fairness between competing threads dynamically at runtime.

The conditions identified relate to the occupancy state of the various chairs within the system, and also as to whether a customer has received a hair cut yet. These can be modeled via conditions based on two shared classes: Chair and Customer.

Chair Method Operations:

- *AddOccupant* – to add a particular customer as the occupant of a chair.
- *RemoveOccupant* – to remove the current customer as the occupant of a chair.
- *GetOccupant* – to return a reference to the current occupant of a chair.

Chair Conditions:

- *Vacant* – true if the chair is currently has no occupant.
- *Occupied* – true if the chair currently has an occupant.

Customer Method Operations:

- *FindAChair* – a composite operation which examines a set of chairs and occupies one if one is available. Used to find a seat in the set of waiting chairs.
- *VacateChair* – a composite operation to remove oneself from ones current chair. Used to remove oneself from the barber chair.
- *SwapChairs* – a composite operation to swap seats from current chair to a new chair. Used to move from a waiting chair to the barber chair.

Customer Conditions:
Chapter 5. Demonstration Examples

- *HairHasBeenCut* – true if the customer has received a haircut.
- *Seated* – true if the customer is currently occupying a chair.

5.3.3 Example Solution

5.3.3.1 Overview

The example solution is composed of two 'shared' classes, and three 'normal' classes:

Shared Classes:
- Customer – a class for representing the customers
- Chair – a class for representing both the barber chair and the waiting chairs

Normal Classes:
- BarberThread – a threaded class which actively drives the barbers activities
- CustomerThread – a threaded class which actively drives a customers activities
- BarberApp – the main application class

Figure 5.18 provides an illustration of the relationships between the classes in the solution.

![Class Relationships in Sleeping Barber Solution](image)

Figure 5.18  Class Relationships in Sleeping Barber Solution
Figure 5.19 provides an illustration of the structure of the solution, showing the relationships between object instances at runtime.

![Diagram showing the structure of the solution](image)

Figure 5.19 Structure of Example Solution to Sleeping Barber Problem

### 5.3.3.2 Sleeping Barber Source Code

Figure 5.20 shows the Java source code listing for the Chair shared object class. Each Chair stores a reference to the Customer who is its current occupant, or null if the chair is unoccupied. It defines three Method operations, getOccupant, addOccupant, and removeOccupant. It also defines two Conditions, Vacant and Occupied.

```java
// Chair.java

import synch.*;

public class Chair extends SharedObject {
    public Customer occupant;

    // Constructor
    public Chair(String name) {
        super(name);
        occupant = null;
        metalInitialise();
    }

    public Occupied occupied = new Occupied();
    public Vacant vacant = new Vacant();
    public GetOccupant getOccupant = new GetOccupant();
    public AddOccupant addOccupant = new AddOccupant();
    public RemoveOccupant removeOccupant = new RemoveOccupant();

    public class Occupied extends Condition {
    }
}
```

163
Both `addOccupant` and `removeOccupant` Method operations are declared as Mutative, which ensures that they execute in mutual exclusion.

The meta-information for the `getOccupant` Method uses the `returnSharedObject` action function. This ensures that if the Method is used as a suboperation within another composite operation, then the potential reference values it may return are taken into account by the access decomposition mechanism when determining the potential targets of further suboperations.
Chapter 5. Demonstration Examples

The vacant condition tests whether a chair is currently free. It is used by the Customer.findAChair operation to examine the occupancy state of the waiting chairs.

The occupied condition tests whether a customer is currently sitting in the chair. It is used by the BarberThread to test if there currently is a customer in the barber's chair.

Figure 5.21 shows the Java source code listing for the Customer shared object class. The Customer has a reference to the Chair object he is currently occupying, as well as a boolean flag to indicate whether a haircut has been received yet. It defines four Method operations and two Conditions.

The findAChair operation is a composite operation that accepts an array of Chair objects as a parameter (the set of waiting chairs). It looks at each element in the array attempting to find a vacant chair. If a vacant chair is found, then the customer occupies that chair. The meta-layer implicitly ensures access atomicity over all the chairs in the set for the duration of the operation execution. This operation is used by the CustomerThread to find a chair in the waiting room.

The vacateChair operation removes the customer from the chair he currently occupies. This operation is called by the BarberThread to ensure a customer is removed from the barber chair at the end of a haircut.

The swapChair operation accepts a Chair object as a parameter. It removes the customer from the chair he currently occupies, and then occupies the new chair. This operation is used by the CustomerThread to move from a waiting chair to the barber chair. The meta-layer implicitly ensures access atomicity over both chairs.

The cutHair operation sets the customers haircut flag to 'true'. It is used by the BarberThread to cut the customer's hair.

The seated condition tests whether the customer is currently sitting in a chair. It is used by the CustomerThread to determine whether the findAChair operation was successful.

The hairCut condition tests whether the customer has received a haircut. It is used by the CustomerThread to wait for the barber to cut the customer's hair.
// 
// Customer.java
//
import synch.*;
import java.util.*;

public class Customer extends SharedObject {
    private Seated seated;
    private HairHasBeenCut hairHasBeenCut;
    private FindAChair findAChair;
    private VacateChair vacateChair;
    private SwapChairs swapChairs;
    private CutHair cutHair;

    public Customer(String name) {
        super(name);
        myChair = null;
        hairCut = false;
        random = new Random();
        initialise();
    }

    public class Seated extends Condition {
        public boolean cond() {
            return myChair != null;
        }
    }

    public class HairHasBeenCut extends Condition {
        public boolean cond() {
            return hairCut;
        }
    }

    public class FindAChair extends Method implements Mutative {
        public void meta() {
            parameterise("chair");
            SharedObjParameter _chair = parameter("chair");
            subInvocation(_chair,"occupied");
            subInvocation(_chair,"addOccupant",self());
            assignMemberFld(memberFld("myChair"),_chair);
        }

        public void impl(Chair[] chair) {
            for (int i = 0; i<chair.length; i++) {
                if (!chair[i].occupied.test()) {
                    chair[i].addOccupant.exec(Customer.this);
                    myChair = chair[i];
                    break;
                }
            }
        }
    }
}

Chapter 5. Demonstration Examples
public class VacateChair extends Method implements Mutative {
    public void meta() {
        subinvocation(memberFld("myChair"), "removeOccupant");
    }

    public void impl() {
        myChair.removeOccupant.exec();
        myChair = null;
    }
}

public class SwapChairs extends Method implements Mutative {
    public void meta() {
        parameterise("newChair");
        subinvocation(memberFld("myChair"), "removeOccupant");
        subinvocation(parameter("newChair"), "addOccupant", self());
        assignMemberFld(memberFld("myChair"), parameter("newChair"));
    }

    public void impl(Chair newChair) {
        myChair.removeOccupant.exec();
        newChair.addOccupant.exec(Customer.this);
        myChair = newChair;
    }
}

public class CutHair extends Method implements Mutative {
    public void impl() {
        int time = BarberApp.HAIRCUT_BASE_DELAY
                + Math.abs((random.nextInt() % BarberApp.RANDOM_DELAY));

        System.out.println("— " + Customer.this + " starts a " + time + "ms haircut");
        try {
            Thread.currentThread().sleep(time);
        } catch (InterruptedException e) {} } 

        hairCut = true;

        System.out.println("++ " + Customer.this + " finishes a " + time + "ms haircut");
    }
}

Figure 5.21 Customer.java Source Code Listing
Figure 5.22 shows the Java source code listing for the BarberThread class. The BarberThread possesses its own independent thread of control which represents the barber's activities. In this example solution, no entity is required to explicitly represent the barber himself, only his activities and interactions with other objects.

```java
import synch.*;

public class BarberThread extends Thread {
    private Chair barberChair;
    public BarberThread(Chair bc) {
        super("barberthread");
        barberChair = bc;
    }
    public void run() {
        for (;;) {
            // wait for the barberChair to be occupied by a customer
            barberChair.occupied.waitTrue();
            // cut the customer's hair
            Customer c = (Customer) barberChair.getOccupant.exec();
            c.cutHair.exec();
            // show the customer out of the barber chair
            c.vacateChair.exec();
        }
    }
}
```

Figure 5.22 BarberThread.java Source Code Listing

The BarberThread loops continuously, invoking the following sequence of invocations:

1. `barberChair.occupied.waitTrue();`
2. `Customer c = (Customer) barberChair.getOccupant.exec();`
3. `c.cutHair.exec();`
4. `c.vacateChair.exec();`
Chapter 5. Demonstration Examples

The first operation, `barberChair.occupied.waitTrue()`, is a conditional invocation which waits for the barber chair to be occupied. The second operation `barberChair.getOccupant.exec()`, retrieves a reference to the customer in the barber chair. The third invocation `c.cutHair.exec()`, cuts the hair of the customer, while the fourth invocation `c.vacateChair.exec()`, shows the customer out of the barber's chair. Because these invocations are called from a normal 'non-shared' object, they are each separate root invocations to the meta-layer. After the fourth operation, the cycle repeats.

Figure 5.23 shows the Java source code listing for the CustomerThread class. Each CustomerThread has a reference to its unique Customer object in addition to the set of Chair objects representing the waiting chairs, and the Chair object representing the barber's chair. Each CustomerThread possesses its own independent thread of control which drives the activities of its Customer instance. A new Customer instance and its associated CustomerThread instance are created and started at random intervals by the BarberApp main application class.

Each CustomerThread invokes the following sequence of invocations:

```
1   customer.findAChair.exec(waitingChairs);
2   if (customer.seated.test()) {
3     barberChair.vacant.execTrue(
              customer.swapChairs.makeInv(barberChair));
4     customer.hairHasBeenCut.waitTrue();
5   }
```

The first invocation passes the set of waiting chairs as a parameter to the `findAChair` operation of the customer. After this operation returns, the thread tests whether the customer managed to occupy a vacant chair or not. If unsuccessful, the customer does not wait any longer and exits immediately. If successful, the thread must wait for an opportunity to occupy the barber's chair. The thread binds the consequent method invocation `customer.swapChairs` with the `barberChair` as a parameter, to the `vacant` condition on the barber's chair. The meta-layer waits if necessary for the condition to be true, and then executes the consequent method invocation, under the atomicity of the true condition. This moves the customer from his waiting chair to the barber chair in a safe and fair manner. Finally, in step 4 the CustomerThread performs a rendezvous with the barber thread by waiting until the
customer's hair has been cut. The CustomerThread waits for the BarberThread to perform services on its behalf (a haircut) before resuming execution.

```
// CustomerThread.java

import synch.*;

public class CustomerThread extends Thread {
    private Customer customer;
    private Chair barberChair;
    private Chair[] waitingChairs;

    public CustomerThread(String name, Customer cust, Chair bc, Chair[] wc) {
        super(name);
        customer = cust;
        barberChair = bc;
        waitingChairs = wc;
    }

    public void run() {
        // see if we can occupy a waiting chair
        customer.findAChair.exec(waitingChairs);
        if (customer.seated.test()) {
            // ok then we are sitting in a waiting chair
            // now we have to wait for the barber chair
            barberChair.vacant.execTrue(
                customer.swapChairs.makeInv(barberChair));
            // wait for the hairCut to be finished
            customer.hairHasBeenCut.waitTrue();
        } else {
            System.out.println("!! " + customer +
                " leaves without waiting for a haircut");
        }
    }
}
```

**Figure 5.23** CustomerThread.java Source Code Listing

Figure 5.24 shows the Java source code listing for the BarberApp class. The BarberApp class constructs and appropriately initialises the required instances of Chair and BarberThread. The start() method then starts execution of the BarberThread and begins creating and starting new instances of Customer and CustomerThread.
import java.util.*;
import java.awt.*;
import synch.*;

public class BarberApp {
    Chair barberChair;
    Chair[] waitingChairs;
    BarberThread barberThread;

    Random random = new Random();

    final static int NUM_WAITING_CHAIRS = 4;
    final static int NUM_CUSTOMERS = 50;

    final static int INTERVAL_BASE_DELAY = 400;
    final static int HAIRCUT_BASE_DELAY = 400;
    final static int RANDOM_DELAY = 200;

    public BarberApp() {
        super();
        barberChair = new Chair("barberchair");
        barberThread = new BarberThread(barberChair);
        waitingChairs = new Chair[NUM_WAITING_CHAIRS];
        for (int i=0;i<NUM_WAITING_CHAIRS;i++) {
            waitingChairs[i] = new Chair("waitChair" + i);
        }
    }

    public void start() {
        int count = 0;
        int time = 0;
        barberThread.start();
        for (int i=0;i<NUM_CUSTOMERS;i++) {
            int delay = INTERVAL_BASE_DELAY +
                    Math.abs((random.nextInt() % RANDOM_DELAY ));
            try { Thread.currentThread().sleep(delay); }
            catch (InterruptedException e) {}  
            Customer c = new Customer("Customer" + count);
            CustomerThread ct = new CustomerThread("CustomerThread" + count,
                        c,barberChair,waitingChairs);
            System.out.println("Customer" + c + " arrives at barbershop at time " + time + "ms");
            ct.start();
            time += delay;
            count++;
    }
public static void main(String[] args) {
    BarberApp simulation;
    simulation = new BarberApp();
    simulation.start();
}

Figure 5.24 BarberApp.java Source Code Listing

5.3.3.3 Execution Output

Figure 5.25 shows the output produced from the execution of the BarberApp program.

Customer0 arrives at barbershop at time 0ms
-- Customer0 sits down in waitChair0
-- Customer0 gets up out of waitChair0
-- Customer0 sits down in barberchair
-- Customer0 starts a 443ms haircut
Customer1 arrives at barbershop at time 493ms
-- Customer1 sits down in barberchair
-- Customer0 finishes a 443ms haircut
-- Customer0 gets up out of barberchair
-- Customer1 gets up out of waitChair0
-- Customer1 sits down in barberchair
Customer2 arrives at barbershop at time 871ms
-- Customer2 sits down in waitChair0
-- Customer1 starts a 425ms haircut
Customer3 arrives at barbershop at time 1208ms
-- Customer3 sits down in waitChair1
-- Customer1 finishes a 425ms haircut
-- Customer1 gets up out of barberchair
-- Customer2 gets up out of waitChair0
-- Customer2 sits down in barberchair
-- Customer2 starts a 400ms haircut
Customer4 arrives at barbershop at time 1530ms
-- Customer4 sits down in waitChair0
-- Customer2 finishes a 400ms haircut
-- Customer2 gets up out of barberchair
-- Customer3 gets up out of waitChair1
-- Customer3 sits down in barberchair
-- Customer3 starts a 572ms haircut
Customer5 arrives at barbershop at time 1948ms
-- Customer5 sits down in waitChair1
Customer6 arrives at barbershop at time 2402ms
-- Customer6 sits down in waitChair2
-- Customer3 finishes a 572ms haircut
-- Customer3 gets up out of barberchair
-- Customer4 gets up out of waitChair0
-- Customer4 sits down in barberchair
-- Customer4 starts a 568ms haircut
Customer7 arrives at barbershop at time 2794ms
-- Customer7 sits down in waitChair0
-- Customer8 arrives at barbershop at time 3132ms
-- Customer8 sits down in waitChair3
-- Customer4 finishes a 568ms haircut
-- Customer4 gets up out of barberchair
Chapter 5. Demonstration Examples

5.3.4 Discussion

5.3.4.1 Customer Thread

The CustomerThread object firstly makes an invocation of the findAChair Method on its Customer, passing the array of Chair objects representing the waiting chairs as a parameter. This composite operation involves testing the vacant condition on each chair to find a spare seat, and then using the occupy method to sit down in the selected chair. Due to the pessimistic nature of the access decomposition mechanism, this entails an invocation of occupy on each Chair element of the parameter array. Since occupy is a mutative operation, the findAChair operation requires exclusive access to all of the waiting chairs, and hence the access decomposition mechanism will determine the access table shown in Table 5.4. This implicit access synchronisation provided by the meta-layer protects execution of the findAChair operation from interference by other customers attempting to select a waiting chair at the same time. The fair invocation scheduling properties of the meta-layer ensures that customers are given an opportunity to find a chair in the order that they arrive at the barbershop.

<table>
<thead>
<tr>
<th>Shared Object</th>
<th>Access Mode</th>
</tr>
</thead>
<tbody>
<tr>
<td>CustomerN</td>
<td>write</td>
</tr>
<tr>
<td>Chair1</td>
<td>write</td>
</tr>
</tbody>
</table>
If the operation to find a chair was succesful, the CustomerThread object then makes a conditional invocation based on the vacant Condition of the barber's chair, with a consequent invocation of swapChairs, that will move the customer into the barber's chair. The meta-layer coordinates safe and fair testing of the condition and when the condition becomes true, the meta-layer attempts scheduling of the consequent swapChairs Method on the customer. The consequent composite operation will require the access table shown in Table 5.5.

<table>
<thead>
<tr>
<th>Shared Object</th>
<th>Access Mode</th>
</tr>
</thead>
<tbody>
<tr>
<td>Customer</td>
<td>write</td>
</tr>
<tr>
<td>Customer's Chair</td>
<td>write</td>
</tr>
<tr>
<td>Barber's Chair</td>
<td>write</td>
</tr>
</tbody>
</table>

Table 5.5 Access Table for Invocation swapChairs.

It may be the case that more than one (possibly four) CustomerThreads may be occupying a waiting chair, and waiting on the same condition of the same target – that the barber's chair becomes vacant. The meta-layer prioritises these conditional invocations according to their unique insertion invocation ID. When the condition becomes true, the meta-layer will determine the access table for the consequent invocation of highest order (lowest ID). Because the consequent invocation involves write access to the target of the concurrent conditional invocations, only the highest order conditional invocation's consequent will be executed – the other conditional invocations will remain in the scheduling queue and will have their conditions retested when the selected consequent finishes execution. In this case, it will turn out that the condition will have become false – the barber's chair will no longer be vacant – and hence the other conditional invocations will have to wait until another method invocation causes the condition to become true.

In this way, the meta-layer safely and fairly resolves conditional contention between threads competing over common resources (the waiting chairs and the barber's chair), without requiring any explicit coordination mechanism in the component objects themselves. This simplifies the programmer's task of implementing such behaviour, and
greatly reduces opportunities for safety and liveness failure due to low-level interference between independent threads.

5.3.4.2 Barber Thread

In order to prevent the Barber cutting a customers hair repeatedly, the BarberThread/CustomerThread interaction is designed such that the Barber always shows the current customer out of the chair before waiting to test the occupied condition on the barber's chair again.

In an alternate design, the CustomerThread may be assigned responsibility for vacating the customer from the barber's chair after the haircut rendezvous. In this case the BarberThread would have to test a slightly different condition than simply testing to see whether the chair was occupied - otherwise the BarberThread may start another haircut before the current customer has vacated the barber's chair. This could be achieved by replacing the occupied condition with one which accepted a Customer parameter and tested if the current occupant was not equal to the passed parameter. Hence the barber could then test to see if it was a new customer.
5.4 Adding Additional Active Participants: Cigarette Smokers Revisited

The meta-layer synchronisation approach allows a high degree of flexibility and adaptability when adding to or modifying a system. It is possible to add additional active participants to a concurrent system without requiring any modifications within the existing components to coordinate the additional patterns of interaction. The synchronisation scheme automatically adapts to coordinate the additional behaviour due to the safety and fairness properties provided dynamically by the meta-layer.

5.4.1 Extension Description

For example, the Cigarette Smokers problem introduced in Section 5.2 has four active participants:

- Agent – the producer participant who adds fresh ingredients to the table,
- Tom – the consumer participant who has an infinite supply of tobacco,
- Pat – the consumer participant who has an infinite supply of paper,
- Mat – the consumer participant who has an infinite supply of matches.

For demonstration purposes, another two active participants are added to the system:

- Agent2 – another producer participant who adds fresh ingredients to the table,
- Paul – another consumer participant who, like Pat, has an infinite supply of paper.

5.4.2 Analysis

The extended system corresponds to the illustration in Figure 5.26. Agent2 simply represents another instance of AgentThread with its own instance of an Agent shared object. Agent2 waits on the same condition as Agent – waiting for the table to become empty so that he can place two new random ingredients on it. This results in contention between Agent and Agent2. Paul represents another instance of SmokerThread with an associated Smoker shared object initialised with a supply
of the 'paper' ingredient. Paul waits on the same condition as Pat – that tobacco and matches are put down on the table.

![Diagram of agents and table with connections]

Figure 5.26 Cigarette Smokers problem with additional participants.

5.4.3 Implementation

Modifying the original example solution to the extended system simply requires trivial changes to the main SmokersApp class – creating the additional instances of Agent, AgentThread, Smoker and SmokerThread, and then starting the two additional threads.

5.4.3.1 Source Code

No changes are required to the component classes Table, Smoker, Agent, SmokerThread or AgentThread. Figure 5.27 shows the additional lines of code required in SmokersApp.java.

```java
Agent agent2 = new Agent("agent2",table);
AgentThread agent2Thread = new AgentThread(agent2,table);
Smoker paul = new Smoker("Paul",PAPER,table);
SmokerThread paulThread = new SmokerThread(paul,table);
paulThread.start();
agent2Thread.start();
```

Figure 5.27 Code Additions to SmokersApp.java.
5.4.3.2 Execution Output

Figure 5.28 shows the output produced from the execution of the extended SmokersApp program.

| inform: smokers problem initialised... |
| --- | |
| agent .. ingredients Tobacco and Paper are put down on table... |
| agent2 .. ingredients Match and Tobacco are put down on table... |
| agent .. ingredients Paper and Match are put down on table... |
| agent .. ingredients Tobacco and Paper are put down on table... |
| agent2 .. ingredients Tobacco and Paper are put down on table... |
| agent2 .. ingredients Match and Tobacco are put down on table... |
| agent .. ingredients Paper and Match are put down on table... |
| agent2 .. ingredients Tobacco and Paper are put down on table... |
| agent .. ingredients Paper and Tobacco are put down on table... |
| agent .. ingredients Paper and Tobacco are put down on table... |
| agent .. ingredients Paper and Tobacco are put down on table... |
| agent .. ingredients Paper and Tobacco are put down on table... |
| agent2 .. ingredients Match and Paper are put down on table... |
| agent .. ingredients Match and Paper are put down on table... |
| agent .. ingredients Match and Paper are put down on table... |
| agent .. ingredients Match and Paper are put down on table... |
| agent .. ingredients Match and Paper are put down on table... |
| agent .. ingredients Match and Paper are put down on table... |
| agent .. ingredients Match and Paper are put down on table... |
| agent .. ingredients Match and Paper are put down on table... |

Figure 5.28 Extended SmokersApp Execution Output

5.4.4 Discussion

The two areas of contention identified are resolved automatically by the safety and fairness properties of the meta-layer's conditional invocation mechanism. For example, the two independent agent processes both make the same conditional invocation that requires waiting for the table to be empty of ingredients, with a consequent method invocation to place two new random ingredients onto the table. The two conditional invocation requests are received by the meta-layer in some particular order, and hence one agent has initial order priority over the other. When the condition becomes true, the meta-layer first attempts scheduling and execution of the higher order invocation's consequent. Because the consequent invocation (placeRandomIngredients) access table involves write mode access on the shared Table object (the target of both of the original contentious conditions), processing of the lower order conditional invocation is blocked until after the higher order consequent invocation has finished execution. At that time, the condition is reevaluated and will subsequently test false. Hence the other agent must wait until the condition becomes true again. In the infinite loop of the AgentThread, the agent who had order priority will then request the same conditional invocation again. However, since it is a new separate root invocation,
it will now be the lower order invocation of the two. In this way, (due to the meta-layer's fairness, and its protection of atomicity over condition and consequent) the two independent agents take alternating turns at placing random ingredients on the table.

In the same manner, Paul and Pat will take alternating turns at taking their matching ingredients from the table whenever that condition is satisfied. The introduction of Paul, however, will have no affect on the behaviour of Tom and Mat.
5.5 Polymorphism, Inheritance and New Behaviour: Dining Philosophers Revisited

As well as additional participants exhibiting existing behaviour, it is also possible to add new forms of active participants to a concurrent system, representing different behaviours and interactions, without requiring modifications within the existing components to coordinate the new patterns of interaction. Further, this example demonstrates the use of inheritance to incrementally specify changes to existing behaviour, which introduces polymorphism into shared classes. Clients of such polymorphic classes can invoke operations on the general abstract class, and are not required to know the actual concrete class of the object they are requesting operations. The synchronisation meta-layer coordinates synchronisation dynamically based on the concrete class, simplifying the design and implementation of the client.

5.5.1 Extension Description

For example, in the Dining Philosophers problem, each participant in the system exhibits common behaviour: each philosopher requires two forks to eat.

For demonstration purposes, another type of philosopher is added system so that we have two types of 'eat' behaviour:

- Normal Philosopher – the existing form which requires two forks in order to eat,
- Dexterous Philosopher – a special type of philosopher who possesses the motor skills required to eat his meal with only his left fork.

The problem definition is changed so that at initialisation time, there is a random 1 in 2 chance that any seat at the table contains a dexterous philosopher as opposed to a normal philosopher.

5.5.2 Analysis

The new behaviour is created by defining a new shared class, Dexterous, which inherits from the existing Philosopher class. Dexterous overrides the eat operation, with a new implementation that involves 'picking up' and 'putting down' only his left fork. The overridden eat operation will provide its own corresponding meta-information declaration. This creates a polymorphic set of Philosopher classes. The active PhilosopherThread object which invokes operations on its Philosopher
instance will act transparently on the general case of philosopher. The meta-layer, however, will automatically provide the access synchronisation required for each philosopher's `eat` operation dynamically.

### 5.5.3 Implementation

#### 5.5.3.1 Source Code

Figure 5.29 shows the Java source code listing for the `Dexterous` shared object class. Each `Dexterous` object is still associated with the two instances of the `Fork` class, for its left and right forks respectively. It inherits the member fields and operations from its `Philosopher` superclass, and redefines the `eat` operation.

The overridden `eat` operation simulates the dexterous philosophers eating cycle by picking up only the left fork, sleeping for a random period of time, and then putting back down the left fork. Because the operation makes subinvocations, a meta-information function is required. There are two subinvocations, `pickUp` and `putDown` on the `leftFork` member field references.

Due to the limitation of the proof-of-concept implementation explained in Section 4.3.5, a wrapper method is required to ensure dynamic method lookup for the `eat()` operation. In addition, the callers of this operation (`PhilosopherThread`) must also be changed to use the wrapper method.

```java
// Dexterous.java

import java.util.*;
import synch.*;

public class Dexterous extends Philosopher {
    public Dexterous(String name, Fork left, Fork right) {
        super(name, left, right);
        metaInitialise();
    }

    private Eat eat = new Eat();
    public void eat() { eat.exec(); } // wrapper method

    public class Eat extends Method implements Mutative {
        public void meta() {
            subinvocation(memberFld("leftFork"),"pickUp",self());
            subinvocation(memberFld("leftFork"),"putDown");
        }
    }
}
```
public void impl() {
    leftFork.pickUp.exec(Philosopher.this);
    int time = Math.abs((random.nextInt() % MAX_DELAY));
    System.out.println(Dexterous.this + " starts eating plate no. " + ++eatCount + " for " + time + " ms.");
    try { Thread.currentThread().sleep(time); } catch (InterruptedException e) {} 
    leftFork.putDown.exec();
    System.out.println(Dexterous.this + " finishes eating.");
}

Figure 5.29  Source Code Listing for Dexterous.java.

Modifying the original example solution to the extended system simply requires trivial changes to the main DinersApp class – random creation of one of the two philosopher types (Philosopher, or Dexterous) before initialising its driving PhilosopherThread object. No changes are required to the component classes within the system, except for modifying PhilosopherThread to invoke the wrapper method to avoid the dynamic method lookup limitation. Figure 5.30 shows the new initialisation loop used in DinersApp.java to construct the objects in the extended system, with the modified lines marked with a comment.

for(int i=0;i<SIZE;i++) {
    Fork left = (Fork) forks.elementAt(i);
    Fork right = (Fork) forks.elementAt((i+SIZE-1)%SIZE);
    Philosopher p;
    if ((random.nextInt() % 2)==0) //*
        p = new Philosopher("Phil"+i,left,right); //*
    else 
        p = new Dexterous("Dext"+i,left,right); //*
    philosophers.addElement(p);
    PhilosopherThread dt = new PhilosopherThread(p);
    threads.addElement(dt);
}

Figure 5.30  Code Modifications to DinersApp.java.
5.5.3.2 Execution Output

Figure 5.31 shows the output produced from the execution of the extended DinersApp program.

Dext0 begins thinking for 375 ms.
Phill begins thinking for 315 ms.
Dext2 begins thinking for 172 ms.
Dext3 begins thinking for 330 ms.
Dext4 begins thinking for 20 ms.
Dext4 finishes thinking.
Dext4 starts eating plate no. 1 for 1959 ms.
Dext2 finishes thinking.
Dext2 starts eating plate no. 1 for 704 ms.
Phill finishes thinking.
Phill starts eating plate no. 1 for 252 ms.
Dext3 finishes thinking.
Dext3 starts eating plate no. 1 for 1011 ms.
Dext0 finishes thinking.
Phill finishes eating.
Dext0 starts eating plate no. 1 for 862 ms.
Phil1 begins thinking for 72 ms.
Phil1 finishes thinking.
Dext2 finishes eating.
Dext2 begins thinking for 450 ms.
Dext2 finishes thinking.
Dext2 starts eating plate no. 2 for 1349 ms.
Dext3 finishes eating.
Dext3 begins thinking for 80 ms.
Dext0 finishes eating.
Phill starts eating plate no. 2 for 166 ms.
Dext0 begins thinking for 252 ms.
Dext3 finishes thinking.
Dext3 starts eating plate no. 2 for 415 ms.
Phil1 finishes eating.
Phil1 begins thinking for 354 ms.
Dext0 finishes thinking.
Dext0 starts eating plate no. 2 for 1827 ms.
Dext3 finishes eating.
Dext3 begins thinking for 207 ms.
Phill finishes thinking.
Dext4 finishes eating.
Dext4 begins thinking for 119 ms.
Dext3 finishes thinking.
Dext3 starts eating plate no. 3 for 995 ms.
Dext4 finishes thinking.
Dext4 starts eating plate no. 2 for 1428 ms.
Dext2 finishes eating.
Dext2 begins thinking for 43 ms.
Dext2 finishes thinking.
Dext2 starts eating plate no. 3 for 962 ms.
Dext3 finishes eating.
Dext3 begins thinking for 55 ms.
Dext3 finishes thinking.
Dext3 starts eating plate no. 4 for 1446 ms.
Dext0 finishes eating.
Phill starts eating plate no. 3 for 36 ms.
Dext0 begins thinking for 223 ms.
Phil1 finishes eating.
...

Figure 5.31 Extended DinersApp Execution Output
5.5.3 Discussion

This example demonstrates two important features of the meta-layer approach: firstly, the schemes compatibility with code reuse via inheritance, and secondly, the ability for clients who invoke operations on polymorphic classes to generalise such requests while still satisfying specialised synchronisation requirements.

The mechanics of the meta-layer approach is based on the meta-information associated with the individual operations involved. This allows subclasses to selectively reimplement operations, or define new ones, without causing any conflict between the implementation or synchronisation code of operations in the superclass and those in subclasses. In accordance with the principles of object-oriented design, this allows designs to be extended via modifications that are additive rather than invasive. The access synchronisation requirements of operations on shared objects is determined based on the meta-information associated with those operations, no matter in which host class those operations are defined. Hence, the meta-layer approach retains compatibility with the concept of inheritance.

The meta-layer approach relieves the caller of operations on shared objects from all responsibility for access synchronisation. Access synchronisation requirements for each invocation are determined dynamically, which ensures it is completely compatible with the concept of client generalisation.
Chapter 6

Alternative Implementations

This chapter discusses alternative design choices and extensions in the implementation of the meta-layer synchronisation scheme. Section 6.1 describes an alternate design which employs an optimistic locking approach, removing the need for call chain analysis and object interaction meta-information. Section 6.2 describes an optimisation extension which allows access decomposition results to be cached, resulting in greater runtime efficiency. Section 6.3 describes simple extensions to support asynchronous invocations and 'future' style function return values. Section 6.4 investigates alternate choices for the scheme's universal invocation ordering rule. Section 6.5 discusses an extension for attaching conditions to methods in order to provide encapsulation for method guards. Section 6.6 describes extensions to support time-outs in the scheduling of both method invocations and conditional invocations.

6.1 Optimistic Locking Approach

6.1.1 Context

An implementation of the strict two-phase locking technique involves a pessimistic approach to locking, because all potential resources must be identified and locked before any are used. This style of approach prevents any conflicts from occurring once execution is allowed to start. In contrast, an optimistic approach to two-phase locking allows resource locks to be acquired incrementally, as necessary, but requires a mechanism to detect and resolve conflict between concurrent operations when they occur.

Conflict resolution mechanisms in concurrent transactional database systems typically involve selecting one of the conflicting transactions for 'abort' or 'roll back', and reverting all resources it had updated back to their initial state. Although this 'rollback'
mechanism is suitable for database records, it is not applicable to the general case of operations on systems of objects in a multithreaded object-oriented program\textsuperscript{1}.

However, if we are prepared to limit the context to only such cases where a 'rollback' mechanism is applicable, then we can develop an alternate design for a generalised synchronisation meta-layer which uses a more optimistic approach to two-phase locking.

Such contexts are characterised by the following limitations:

- operations on shared objects may not involve subinvocations of operations on any objects which are outside the control of the meta-layer synchronisation scheme – such operations may not be automatically reversible.

- the total effect of any operation on any shared object must be completely reversible simply by reverting the internal memory state of the shared objects updated by the operation back to their prior values.

6.1.2 Alternative Design

This section describes the design of a scheme which incorporates a more optimistic locking approach than that described in Chapter 3. However, this design still detects many forms of conflict before they occur, and so cannot be considered completely optimistic. A completely optimistic approach does not bother to detect access conflict until the completion of an atomic action. At this point, the operation is either committed or aborted. In contrast to the extreme of that approach, this design detects conflict as a thread steps through the invocation points of a composite operation, and schedules concurrent threads to avoid conflicts where possible. Abortion of operations is only considered as a last attempt to resolving interference between concurrent threads.

An optimistic locking approach to the meta-layer synchronisation scheme retains the basic principles and features of the pessimistic approach:

- safety via transparent and automatic enforcement of access synchronisation for all operations on shared objects.

- atomicity and isolation execution properties provided automatically for composite operations.

\textsuperscript{1}See Section 2.5.4.3.
• decoupling of method invocation from method execution via the reification of
  invocations and their subsequent scheduling under the control of the
  synchronisation meta-layer.

• pervasive readers/writer access approach based on operation target access
  mode meta-information.

• access conflict resolved by a universal invocation ordering rule.

• compatibility with object-oriented principles of encapsulation, inheritance,
  generalisation and composition.

An optimistic approach differs from the pessimistic approach in the following ways:

• no shared object interaction (field or action) declarations required in meta-
  information – only target access mode.

• no access decomposition stage required to model potential call chains.

6.1.2.1 Conflict Detection

At each invocation point for operations on shared objects, control is passed to the
meta-layer. The meta-layer keeps track of the details of all the objects accessed
(invocation targets) and access modes (via operation meta-information) for each root
invocation. These details form the access tables that are used to protect the atomicity
and enforce the isolation properties on composite operations for the duration of each
root invocation. At each (sub)invocation point, the meta-layer updates the root
invocation’s access table to take into account the new subinvocation (target + access
mode). The access table is then compared against the access tables of other
concurrently executing root invocations.

6.1.2.2 Conflict Resolution and Scheduling

Execution of operations is only allowed to proceed if no conflicts are detected. If any
access conflict is detected, then conflict resolution is required. There are a variety of
strategies for conflict resolution:

• FIFO per Object – If a thread invokes an operation (or suboperation) that
  requires access on a target that conflicts with the access table of another root
  invocation in progress, then the calling thread is blocked and must wait in turn
  in a queue for access to the contentious target. This strategy can lead to
deadlock situations, in which case one or more of the participant invocations
must be rolled back. In this strategy the scheduler must be able to detect such deadlock situations.

- Invocation Ordering Rule – If a thread invokes an operation (or suboperation) that requires access on a target that conflicts with the access table of another root invocation in progress, then one of the invocation’s is selected to continue, while the other is aborted. The victim is selected based on an invocation priority rule such as FIFO per root invocation, thread priority or operation priority. This allows situations in which a longer higher priority invocation causes the abortion of a smaller lower priority invocation that may have almost completed execution. This strategy avoids deadlock situations, but may result in increased occurrences of abortions/roll backs.

6.1.2.3 Roll-Back Mechanism

When a thread requests a write-mode operation on a shared object, a 'back-up' copy of the object's internal state is made before execution is allowed to proceed. The copy is used in the rollback mechanism if subsequently required. However, if a root operation's call chain subsequently involves multiple invocations of write-mode operations on the same target instance, only one 'back-up' copy of the instance state is required. In this way, the meta-layer maintains for each root invocation, a single back-up copy of the internal state of those shared objects which have been modified.

If roll-back of a root invocation is required, the meta-layer simply refreshes the internal state of the modified objects from their back-up copies. The aborted root invocation is then restarted anew, and will have to wait for access to the contentious targets.

In an alternate scheme, sets of 'back-up' copies of the internal state of mutated shared objects may be kept at points corresponding to the first use of each target involved in the root invocation. In the event access conflict occurs over a particular target, this approach allows a partial roll-back of the root operation to the first point in which the contentious object was used. The partially rolled-back operation then waits to reacquire access to the contentious object before resuming execution from the roll-back point.

6.2 Caching Access Decomposition Results

This section describes an optimisation extension to the design of the meta-layer's access decomposition mechanism. The design extension reduces the computational overhead involved in synchronising operations which do not inherently involve any
dynamic forms of interaction among successive invocations of the same operation on the same target object.

6.2.1 Rationale

6.2.1.1 Computational Costs of a Dynamic Approach

The meta-layer synchronisation scheme described in Chapter 3 takes a dynamic, invocation-time approach to detecting and resolving access synchronisation conflict between concurrent operations on a system of shared objects. In practice, however, many of the interdependencies between shared objects are often characterisable as static or nonvolatile relationships. Hence, the meta-layer may devote considerable computational resources to detecting and resolving the same conflict scenario, at each root invocation point, time and time again.

The benefit of the dynamic approach is represented by the flexibility of the scheme in adapting to new conflict scenarios; either those encountered in an environment involving dynamic relationships, or those resulting from the reuse, extension, or addition of components to an existing system to form a new application.

The cost of the dynamic approach is the computational resources expended at runtime when the access decomposition process is performed to determine the access requirements for the same operation, over and over again for every request of the same operation on the same target.

6.2.1.2 Static Relationships Between Shared Objects

Often, relationships between objects are of a static or nonvolatile nature. For example, when a 'aggregate' object is created and initialised, and an association is made via a member field reference to another 'delegate' object, such that the association remains constant for the life of the aggregate object.

In such a situation, the aggregate object may define a composite operation which involves a subinvocation on the delegate. In this case, each time a caller requests execution of the composite operation, the meta-layer access decomposition mechanism creates a call chain model which combines the details of the subinvocation, with the potential values for the object reference stored in the member field used as the target.

However, if the member field reference used as the subinvocation target is known to be a nonvolatile association, then the call chain modeling performed by the access
decomposition mechanism in successive separate root invocations is needlessly repeated.

For example, in the Dining Philosophers problem treated in Section 5.1, each Philosopher instance has static relationships with his two (left and right) Fork instances. Consequently, the call chain modeling required for each invocation of the Philosopher.eat() operation is repeated identically at each eating cycle.

This section introduces an optimisation which allows parts of the call chain models required for these situations to be computed once, and then cached for immediate use the next time they are required.

6.2.1.3 Declaring Member Field References as Nonvolatile Associations

Some form of meta-information is required per member field in order for the meta-layer to delineate a particular shared object member field reference as either dynamic or static. With the use of language extensions, this information could be specified with the use of some field modifier keyword, in the same way that access control modifiers are used to control field visibility.

The Java language uses the 'static' keyword to indicate that a particular variable is a class variable. To avoid confusion, the meta-layer scheme consequently uses the term 'nonvolatile' to denote that a particular shared object member field holds a reference to another shared object that will stay constant over the life of the referrer object.

The SharedObject base class provided by the meta-layer scheme is extended to include the abstract function:

```
public void fieldMetaInfo().
```

This function is called by the meta-layer during the meta-information initialisation stage of each shared class. Concrete classes of shared objects may override this function in order to declare any member reference fields as nonvolatile.

Such declarations have the form:

```
declareNonVolatile(fieldName);
```

where

```
fieldName
```

is a string containing the name of a local nonvolatile member field

This calls a function in the SharedObject base class which records the meta-information in the list of member fields for that concrete class.
Nonvolatile shared object member reference fields must be initialised in the shared object's constructor, and there must be no way for the reference value to be set to a new value apart from null. That is, no Method operations defined by the class may set the field to a new reference value.

Any shared object member reference fields not declared nonvolatile are implicitly regarded as volatile.

6.2.1.4 Reusable Static Call Chain Nodes

The parts of the call chain model that may conceivably be cached for subsequent use are the call chain nodes. Each call chain node models the potential interaction resulting from the invocation(s) of a particular Method operation on a particular shared object instance. In the case of composite operations, call chain nodes form directed graph structures to represent the structure of potential call chains.

If it can be determined that the information represented by a call chain node is static with respect to subsequent invocations of the operation in separate root invocations, then the CallChainNode object is denoted static, and may be stored for future use. During access decomposition of the next root invocation which may potentially involve this same operation on the same instance, the cached CallChainNode object (and its links to any child nodes) may be inserted immediately into the new call chain graph.

Each call chain node is a reflection of the potential activities performed by the Method operation that it corresponds to. Hence, the particular properties that determine whether a call chain node is static relate to the potential actions performed by its Method's implementation function, and hence the meta-information it declares.

If a Method operation is characterisable as nonvolatile, then it will have static access requirements, and it will produce a static call chain node. If the Method operation is volatile, then the access decomposition mechanism will need to build a new call chain node for each root invocation's call chain graph.

6.2.1.4 Characteristics of Static Call Chain Nodes.

Any shared object reference field not explicitly declared as nonvolatile is regarded as volatile. Hence, the following types of references to shared objects are considered volatile fields:

- any shared object reference parameters
- shared object member field references not explicitly declared nonvolatile
• any references to new shared objects instantiated within the operation
• any shared object reference results of subinvocations to volatile Method operations

The following actions within a Method operation characterise it as volatile:
• subinvocations based on a volatile target shared object field
• the use of a volatile shared object field as a return result
• assigning new shared object reference values to any member fields

Hence if all of the shared object reference variables accessed within a Method implementation are nonvolatile, then the Method itself is nonvolatile, and its corresponding call chain node will be static and suitable for caching and reuse.

6.2.2 Design Extensions

6.2.2.1 Initialisation of Class MetaInformation

The initialisation process for each concrete SharedObject classes meta-information is extended to automatically process the fieldMetaInfo() function of each shared class, in order to identify which member fields are nonvolatile.

6.2.2.2 Initialisation of Method MetaInformation

The initialisation process for each concrete Method class’ MethodMetaInformation object is extended to examine the volatility of the shared object reference fields involved in the shared object interaction declarations. Hence, the meta-layer determines at initialisation time whether the invocation of a particular Method operation will produce a static call chain node.

6.2.2.3 Decouple Call Chain Nodes from Invocation Call Chain Graphs

The design for access decomposition and call chain modeling is modified so that the CallChainNode objects can be used as flyweights [GHJV94] – objects that can be used in multiple contexts simultaneously.

All contextual information coupling a CallChainNode object to the CallChainGraph of the Invocation in which it appears is removed. CallChainNode objects are linked via CallChainLink objects. The associations between CallChainLinks and CallChainNodes are managed via a hashtable in each CallChainGraph.
Chapter 6. Alternative Implementations

This allows the same static CallChainNode object to be reused and shared among the call chain graphs of multiple invocations.

6.2.2.4 Cache Static CallChainNode Objects per Method

The Method base class is extended to hold a reference to a CallChainNode object. If a concrete Method operation is determined to be nonvolatile, then the first time a CallChainNode is built for the Method, a reference to the newly constructed node is saved in the base class Method object.

6.2.2.5 Revise CallChainNode Factory of Method Class

The makeCallChainNode() object factory function in the Method base class is revised to reuse a cached static call chain node if it is available, rather than creating a new call chain node instance and rebuilding its corresponding interaction model from the Method's meta-information once again.

6.2.3 Discussion

The optimisation described above has been implemented in the proof-of-concept implementation. The optimisation feature is switchable via a boolean flag in the Scheduler object's source code. The savings in call chain node objects created can be measured with the use of the profiling tool distributed with the Java environment.

For example, the optimisation can be applied to the Cigarette Smokers problem discussed in Section 5.2. In this system, all forms of association are static, and all Method operations result in static access requirements. That is, each of the participants interacts only with the shared Table object. The relationship to the Table object may be declared as nonvolatile in each of the collaborator components – the Agent and Smoker shared objects.

The AgentThread is then modified to perform only 500 iterations of the cycle, before stopping the execution of the program. The Java profiling tool is then used to measure the number of CallChainNode instances created during program execution.

Without the optimisation feature enabled, the meta-layer creates 2003 CallChainNode objects during synchronisation of the 500 cycles (1 producer/3 consumers).

In contrast, with the optimisation feature enabled, the meta-layer creates only 9 CallChainNode objects during the same set of interactions.
This is a large savings when we consider that the CallChainNode object itself is an aggregate of a number of linked lists and hashtables, and also requires non-trivial initialisation processing in order to determine and model its relationships with other CallChainNode objects.

6.3 Asynchronous Invocation

In synchronous invocations, the calling thread of control waits for the execution of the requested operation before continuing execution of the sequence of instructions in the calling code. Traditional sequential programming languages use the synchronous style of invocation by default. In contrast, an asynchronous invocation allows a caller to request an operation be executed, but immediately proceed to executing the next instruction of the calling code sequence. This usually involves the implicit creation of a new thread of control to perform execution of the requested suboperation.

6.3.1 Implementation Extensions for Asynchronous Semantics

The design of the synchronisation meta-layer reifies the invocation concept, which in turn decouples method invocation from method execution. This provides an opportunity to easily support an asynchronous style of invocation execution.

As explained in Section 4.3, when a caller requests execution of a Method operation on a shared object, there are two steps to the invocation process. The first step is the reification of the invocation by a call to the makeInv() function on the target shared object's corresponding Method member instance. The second step is where the newly instantiated Invocation object is passed to the meta-layer for scheduling and execution.

The normal synchronous mechanism involves calling synchSchedule() on the new Invocation object which in turn calls the meta-layer Scheduler's scheduleMethodInvocation() function, passing itself as a parameter. After control returns from the meta-layer, the synchSchedule() function returns the Invocation's member ReturnValue object to the caller.

To invoke operations in the asynchronous style, a caller may use the asynchSchedule() function of the new Invocation object. This function simply creates a new thread of control which calls the meta-layer Scheduler's scheduleMethodInvocation() function on the original threads behalf. The asynchSchedule() function returns control to the caller immediately, providing asynchronous execution semantics.
6.3.2 'Future' Style Return Values

For asynchronous invocations of operations that return a value, a mechanism is required for coordinating the delivery and use of the return value back to the calling code. One common approach for coordinating the use of return values of asynchronous invocations is the future mechanism [Hal85, LS88].

In the 'future' mechanism, a future variable is created in conjunction with an asynchronous invocation. The caller proceeds with its own execution in parallel with the execution of the requested function, until an explicit evaluation of the future is attempted. At this point the client will block only under the condition that the requested function has not already returned, thus supplying the value to the future. The future variable is not explicitly visible to the requested function, the future mechanism coordinates the delivery of the function return value to the future variable.

It should be noted that the design of the proof-of-concept implementation already contains the necessary levels of abstraction to add support for a future-style return value mechanism. The meta-layer scheduler wraps the return value of each Method implementation function executed in a ReturnValue object which is contained as a member field of each Invocation object. The ReturnValue class defines a value() accessor function to unwrap and return the actual value of the operation. The Method class' convenience wrapper function exec(), which by convention is used by clients to invoke operations on shared objects, returns the result of this value() function.

To provide a future-style mechanism for the meta-layer scheme, the ReturnValue class is simply extended to recognise asynchronous style invocations, and block any thread requesting a ReturnValue object's inner value until the asynchronous thread executing in the meta-layer has finished execution and set the inner value.

Callers who use the asynchSchedule() asynchronous invocation function are immediately returned the invocation's member ReturnValue object. The callers thread of control can continue and will not block until the caller attempts to access the inner value of the object through its value() function. At that point the caller will be blocked if the asynchronous invocation has not returned yet.
6.4 Alternative Invocation Ordering Schemes

The synchronisation meta-layer scheme described in this thesis uses a FIFO invocation ordering rule. Each root invocation is assigned a unique chronological serial ID number immediately upon entering control of the meta-layer, and before any form of synchronisation or blocking is effected. If the access requirements of a new root invocation do not conflict with those of any other invocations currently within the meta-layer, then the new invocation may execute immediately. However, if conflict is detected, invocation IDs impose an invocation order that is used to resolve the various forms of access conflict and contention between concurrent invocations, such as:

- conflicting access to shared object member field reference values (method invocation access decomposition phase)
- conflicting access to logical execution locks protecting shared objects (general invocation scheduling phase)
- contention between conditional invocations seeking to test conditions and method invocations seeking to execute operations.
- contention between multiple conditional invocations, with conflicting consequent method invocations, waiting on the same condition on the same target.

It is the mechanics and operation of the meta-layer itself that prevents interference, race conditions, and deadlock situations. The advantage of the FIFO scheme is that it provides fairness and prevents starvation by guaranteeing that if the conflict resolution mechanism blocks a conflicting invocation, it will subsequently 'unblock' the same invocation and allow it to continue at some definite point. Hence, the conflict resolution mechanism guarantees that it is not possible for an invocation blocked by access conflict to continue waiting indefinitely.

In some situations, such fairness between competing invocations is not required, or a different conflict resolution scheme may be preferred. The high-level design of the meta-layer scheme can be easily adapted to employ alternate forms of invocation ordering for its conflict resolution mechanism.

---

2See Section 3.6.2.
3Not to be confused with conditional invocations - which may, of course, wait for an indefinite period for their condition to become true.
6.4.1 Invocation Priority

A number range is defined to represent the possible values for invocation priority. A priority value is attached to each root invocation object and used, if required, to impose an order upon conflicting invocations. A universal mechanism for attaching a particular priority value to each individual root invocation may be selected from one of the following:

- Root Operation Priority – As an additional element of meta-information, a Method definition may include a priority value that will be attached to root invocations of the operation.
- Root Invocation Priority – A client may specify a priority value at the root invocation point.
- Thread Priority – The meta-layer may extract a priority value for the root invocation from the priority value of the thread executing the request.

It may occur that conflict arises between competing invocations which possess identical invocation priority values. To prevent deadlock and/or infinite loops within the operation of the synchronisation meta-layer, the ordering rule must be deterministic for any two particular invocations – i.e. the rule should always impose the same ordering result upon successive application to two particular invocations. For this reason, it may be useful to retain the unique chronological serial ID numbers, which can then be used as a secondary ordering scheme for invocations which have identical priorities. This allows a tiered conflict resolution approach that allows priority-based scheduling of invocations, while ensuring fairness between invocations of the same priority.

6.5 Encapsulation of Method Guards

The synchronisation meta-layer scheme described in this thesis provides atomicity and isolation for all method invocations. These properties are provided dynamically at the granularity level of root invocations. The nature of automatic access synchronisation, however, conflicts with the use of condition synchronisation nested within such atomic root invocations.

Consequently, the meta-layer scheme supports only a constrained form of condition synchronisation, where conditional invocations may be created which bind a single

---

4See Section 3.7.1.
Chapter 6. Alternative Implementations

condition synchronisation event to the consequent atomic execution of particular operation. The meta-layer allows no further condition synchronisation to be nested within the execution of the consequent operation. Hence, such conditional invocations may be requested only from clients external to the control of the meta-layer, and not from within method operations defined by shared classes.

Guards\(^5\) represent an abstraction for condition synchronisation coupled to the execution of operations. The mechanics of the guard concept are similar to those of the conditional invocation abstraction offered by the meta-layer, however the guard concept allows for the encapsulation of such conditional interfaces within the target class definition. The client simply requests the operation, the target class itself describes the conditions which must be satisfied in order for the request to proceed.

In the interests of encapsulation, it may be desirable to extend the meta-layer scheme to allow the association of conditions with the methods defined by a shared class in order that external clients may simply request a method operation without explicitly constructing a conditional invocation (from a condition object and a consequent method invocation object) themselves. In this way, shared classes could encapsulate the details of any guarding conditions, reducing the coupling between clients and the shared class itself.

The design of the meta-layer already contains the abstractions necessary for such an extension. The abstract Method base class is easily extended to support the optional specification of an associated Condition object. If a Condition is specified, then when an external client makes a request for the operation, the Method object creates a ConditionalInvocation object using the associated Condition object, instead of the usual MethodInvocation object. The ConditionalInvocation is scheduled and executed as per the established invocation control mechanisms.

However, such situations are only applicable for external requests (or root invocations) made from 'normal' objects outside the control of the meta-layer, as nested conditional invocations are not allowed within composite operations on shared objects.

6.6 Invocation Time-Outs

In some condition synchronisation situations, rather than waiting forever for a particular condition to become true, a client may wish to place an upper bound on how

\(^5\)See Section 2.4.3.
long it should remain suspended. If the specified time limit elapses before the condition becomes true, then the conditional invocation returns immediately with an error code indicating a time-out.

In other circumstances (for example, schemes utilising a priority-based invocation ordering rule rather than a fair FIFO rule), time-outs may also be useful for certain normal method invocations. In these situations, if an invocation cannot be executed before a specified time limit elapses, due to access conflict with competing invocations, then the method invocation returns immediately with an error code indicating a time-out.

However, since the scheme schedules requests at the root invocation level, such time-out values are only meaningful for root invocations. Nested invocations never have to wait for access, rather, they execute immediately, due to the access decomposition stage and scheduling mechanisms of the meta-layer scheme.

The meta-layer synchronisation scheme already contains the abstractions necessary to support both types of time-out features.

6.6.1 Invocation Access Conflict Time-Out

The meta-layer scheme's Invocation base class is extended to hold an optional access conflict time-out value and an invocation start timestamp. The time-out value specifies, in milliseconds, the upper bound on the time that the invocation should remain in the access decomposition or scheduling stages of the synchronisation meta-layer, before returning time-out failure. The invocation start timestamp is used to hold the time at which the invocation enters control of the synchronisation meta-layer. The Invocation base class is also extended to provide functionality which allows a client to set an invocation's access conflict time-out value. The default would be to wait indefinitely.

The meta-layer scheme's Scheduler class is extended to assign a timestamp to new invocations, and to time-out invocations from the access decomposition and scheduling stages of the meta-layer if their access conflict time limit elapses. The timed-out invocation would return immediately with its ReturnValue object encapsulating a special time-out value. The time-out of a method invocation may immediately effect the progress of other invocations currently within the control of the meta-layer, if other blocked lower order invocations may then be able to proceed within the access decomposition or scheduling stages.
6.6.2 Conditional Invocation Wait Time-Out

The meta-layer scheme's `ConditionalInvocation` class is extended to hold an optional *condition wait time-out* value. The time-out value specifies, in milliseconds, the upper bound on the time that the invocation should remain waiting in the active condition stage\(^6\) of the synchronisation meta-layer, before returning time-out failure. The `ConditionalInvocation` class is also extended to provide functionality which allows a client to set an invocation's condition wait time-out value. The default would be to wait indefinitely.

The meta-layer scheme's `Scheduler` class is extended to time-out conditional invocations from active conditional stage of the meta-layer if their condition wait time limit elapses. The timed-out conditional invocation would return immediately with its `ReturnValue` object encapsulating a special time-out value. The time-out of a conditional invocation would not immediately effect the progress of other invocations currently within the control of the meta-layer.

\(^6\)See Section 4.5.
Chapter 7

Conclusion

Section 7.1 provides a summary of the contributions of the work, while Section 7.2 describes aspects of the research that could form the subject of subsequent investigation.

7.1 Summary

The aim of this thesis was to investigate an approach for addressing the conflicts between the design goals of the object-oriented approach and the necessities of synchronisation to ensure safe computations in multithreaded programs. The research was motivated by the failure of existing schemes to provide atomicity and isolation execution properties necessary for synchronisation while still allowing the unrestricted use of object-oriented features such as encapsulation, generalisation, reuse and composition, in program designs.

The workings of a meta-level approach to synchronisation in multithreaded object-oriented programs was presented. The scheme introduces a high-level abstraction that implicitly provides automatic atomic access synchronisation for all operations on shared objects. The scheme decouples method execution from method invocation by passing control of the evaluation, scheduling and execution of invocations on shared objects to a generic synchronisation meta-layer. It provides atomicity and isolation extending implicitly over both simple and composite operations, through the introduction of a dynamic pre-execution analysis stage and a subsequent scheduling stage which employs a strict two-phase locking protocol to protect and isolate operation execution. A high-level design of the scheme was provided, discussing aspects such as the shared object model, invocation mechanism, access decomposition mechanism, and the generic scheduling mechanism. Finally, the compatibility of the approach with the concept of condition synchronisation was discussed, and a
mechanism for employing a constrained form of condition synchronisation was described.

The meta-level scheme provides a dynamic, adaptable, flexible and reusable approach to synchronisation in multithreaded object-oriented programs based on the following characteristics:

- **Safety** – The scheme automates the processes required for a client to organize synchronisation for operations. The access synchronisation mechanism is transparently coupled to the operation invocation mechanism, resulting in reliable isolation for all operations. In addition, the removal of explicit synchronisation instructions from operation implementations reduces complexity and allows more flexibility, reusability and maintainability.

- **Retaining compatibility with O-O design principles** – The scheme retains full compatibility with the basic principles of object-oriented design: encapsulation, generalisation and inheritance. This enables software designers to realise the benefits of an object-oriented approach in managing complexity in the design and maintenance of systems of shared components. In particular, the ability to successfully encapsulate and reuse component classes, and the availability of polymorphism and dynamic binding to achieve generalisation from the caller's perspective, are both key O-O techniques for managing software complexity.

- **Atomicity and isolation execution properties for composite operations** – The scheme works transparently with the principle of functional composition, such as aggregation or collaboration between objects. The execution qualities of atomicity and isolation protecting simple operations individually are implicitly extended to encompass arbitrary composite operations as a whole, through the use of an access decomposition stage and a strict two-phase locking protocol.

- **Liveness** – The scheme's meta-layer automatically manages the execution of concurrent operations, removing scheduling control from the participant objects. Through the use of a fair invocation ordering rule to resolve conflict, the scheme provides fair and reliable scheduling qualities, free of undesirable liveness failures such as the starvation of operation requests, or access deadlock between concurrent operations.

A novel aspect of the meta-level synchronisation approach described here is the way in which it addresses the issue of part/whole relationships between the components of system. Object-oriented design aims for complete encapsulation, in which the part implementations have no explicit knowledge of the whole's structure or private
implementation. In contrast however, the mechanics of synchronisation often require a
centralised and concrete view of such part/whole relationships – control and
coordination of the interaction between concurrent behaviour requires identification
and knowledge of the entities involved. The meta-level scheme described here retains
the flexibility of the object-oriented approach by not attempting to explicitly
encapsulate complete views of part/whole relationships within any single class
definition. Rather, class definitions declare details of such relationships only within
their immediate contexts, and the complete view is built dynamically at invocation
time by the generic meta-layer, unifying the state of current relationships to form the
larger concrete view of the system. This resolution retains the ability to completely
encapsulate component classes, while providing a way to determine, even in dynamic
cases, details of such part/whole relationships (i.e. the participants involved in a
composite multi-object operation). The access decomposition process is managed by a
generic mechanism which is an internal component of the meta-layer scheme. Hence,
shared classes and clients that invoke their operations are shielded from the
complexity of the access decomposition mechanism.

A proof-of-concept implementation of the scheme was built using the Java™ 1.1
language. It is available as a Java package containing a framework of reusable classes.
Various implementation issues related to the scheme's shared object model, invocation
mechanism, and general scheduling mechanism were discussed.

Demonstration example solutions to various classical concurrency problems were
provided, based on the proof-of-concept implementation of the scheme. The initial
eamples provide an illustration of the use of the scheme in solving various problems
involving access synchronisation, condition synchronisation, and condition-based
rendezvous. These examples demonstrated how the automatic provision of atomicity
over composite operations can simplify the design of synchronisation in multithreaded
object-oriented programs. Further examples demonstrated the introduction of
additional active participants to an existing system, and the use of inheritance,
generalisation and polymorphic shared classes to achieve reuse and reduce complexity.
The latter examples demonstrated the flexibility and adaptability of the scheme's
dynamic approach when additions or modifications are made to the systems of
objects and interactions in multithreaded programs.

Alternative approaches and strategies in the implementation of the high level scheme
were discussed. Strategies include an optimistic locking approach which obviates the
need for the costly access decomposition stage, and an optimisation to cache access
decomposition results which can vastly improve the resource efficiency of the scheme.
Extensions to support asynchronous invocations, 'future' style return values, and invocation time-outs were also described. Alternative invocation ordering schemes, such as priority-based schemes were discussed. Finally, extensions to support the encapsulation of conditions acting as method guards were discussed.

7.2 Future Research

Aspects of the research presented in this thesis could form the basis for further work. This section outlines some possible issues which may be investigated or addressed.

I Implementation

Chapter 3 describes the high level design of a meta-level synchronisation scheme for scheduling operations on shared objects in order to obtain execution properties of atomicity and isolation. In Chapter 4, a proof-of-concept implementation, based on a framework of classes using the Java™ 1.1 language, is described. Ideally, a native implementation of the scheme is required, comprising of language extensions, a compilation system, and a native runtime support environment. Custom language extensions, specific to the semantic requirements of the meta-level synchronisation scheme, would allow a reduction in the software text verbosity seen in this thesis, in both the shared class implementations and the client code which invokes operations on them. In addition, a sufficiently advanced native compilation system may be able to determine various meta-information elements implicitly, while an optimised native runtime support environment is required to reduce runtime resource utilisation.

II Optimisation

The meta-layer synchronisation scheme described in this scheme is primarily an experimental effort to address conflicts between the design ideals of the object-oriented approach and the necessities of synchronising composite multi-object operations. As such, this thesis does not consider the runtime efficiency of the scheme. Various possibilities for optimisation in both the high-level design and low-level implementation of the scheme require exploration.

III Control Over Granularity of Atomicity

The meta-layer synchronisation scheme implicitly fixes the granularity of atomicity over composite operations to encompass the entire scope of each root invocation presented to the meta-level scheduler at runtime. The scheme does
not allow participant component classes to exercise any control over granularity, and provides no support for any forms of multi-granular operations. The scheme is dynamic in that it automatically adjusts granularity to ensure atomicity for each root invocation. However, the approach suffers from inflexibility. The scheme does not allow the release of any resources until all are released at the end of the root operation – which may limit liveness and availability in systems featuring large grained atomic actions. Investigation is required to determine the possibility of mechanisms which allow the specification of multi-granularity without compromising the ability of the scheme to provide atomicity over arbitrary composite operations. A multi-granularity mechanism may lead to more flexibility in the use of condition synchronisation within composite operations, however, deadlock issues may arise.

IV Access Synchronisation

The meta-layer scheme's approach to access synchronisation uses a readers/writer protocol pervasively. This simple yet powerful approach allows for intra-object concurrency in addition to mutual exclusion policies. It also is compatible with the theme of composition – differing accesses to the same resource within a composite call chain may be summarised by the stronger form of access. Investigation is required to determine the suitability of integrating other access synchronisation protocols with the meta-layer scheme.

V Condition Synchronisation

In accordance with its automatic access atomicity/isolation focus, the meta-layer scheme features a constrained form of condition synchronisation which limits the use of such condition based behaviour to immediately before or between atomic actions. Hence, condition based behaviour is illegal within the body of an operation on a shared objects. In combination with either the proposition of multi-granularity, or an optimistic approach featuring a rollback mechanism, investigation is required to identify possible approaches for removing these restrictions on condition synchronisation. This may allow the use of method guards as required throughout a composite call chain, instead of restricting their use to only the root operation. However, allowing nested condition synchronisation inevitably introduces the potential for deadlock situations.
Appendix A

Framework Class Overview

This chapter briefly introduces the component classes of the scheme and their usage within the framework. A class diagram depicts the classes, their structure and the relationships between them.

A.1 Framework Class Hierarchy

Figure A.1 illustrates the class inheritance diagram for the components of the proof-of-concept implementation framework for the synchronisation meta-layer.

Public Core Components:

- **SharedObject** – the abstract base class which is subclassed for defining new classes of objects to be shared.

- **Method** – the abstract base class which is subclassed for defining operations on concrete subclasses of SharedObject.

- **Condition** – an abstract subclass of Method which is further subclassed for defining conditions on concrete subclasses of SharedObject.

- **Mutative** – a convenient mixin interface which may be declared by a concrete Method operation to associate a 'write' target access mode to the operation. The Method base class supplies 'read' target access mode for operations as default.

- **Invocation** – the abstract class for invocations which are managed by the meta-layer scheduler. It has two concrete subclasses: MethodInvocation and ConditionalInvocation.
Appendix A. Framework Class Overview

- **MethodInvocation** – a class used to represent a request to execute a Method operation on a SharedObject.

- **ConditionalInvocation** – class used to represent a request to wait for a Condition on a SharedObject to become true, then optionally to execute a consequent MethodInvocation on a SharedObject.

- **ReturnValue** – a wrapper around the return value of an invocation of a Method operation on a SharedObject. If the return value is of a primitive type, it is wrapped in an appropriate object.

Meta-Information Components:

- **SharedObjRefFld** – an abstract base class which represents a field which can store a reference to a SharedObject instance.

- **SharedObjMemberFld** – a subclass of SharedObjRefFld which represents those member fields of a SharedObject which may store references to other SharedObject instances.

- **SharedObjNewInstance** – a subclass of SharedObjRefFld which represents a reference to a new SharedObject instantiated within the operation body.

- **SharedObjParameter** – a subclass of SharedObjRefFld which represents a formal parameter to the Method operation which holds a reference to another SharedObject instance.

- **SharedObjSubResult** – a subclass of SharedObjRefFld which represents the value returned from a subinvocation from within this operation, which may be a reference to another SharedObject instance.

- **SubinvocationDescriptor** – a meta-information object which describes the details of a potential subinvocation from within a Method operation.

Private Components:

- **Scheduler** – the global singleton object which coordinates the execution of all invocations on SharedObject instances.

- **AccessClaim** – identifies a particular access mode on a particular target SharedObject instance.
Appendix A. Framework Class Overview

- **AccessTable** - a table of **AccessClaim** objects that are used to represent the total potential access requirements of a root invocation.

- **MemberFieldValueTable** - manages potential values (for use by call chain nodes in the access decomposition process) of those member fields of a **SharedObject** which may store references to other **SharedObject** instances. Also manages synchronisation of access to those values by separate invocations concurrently performing access decomposition.

- **CallChainGraph** - a graph of **CallChainNode** objects that represents a model of the potential call chain of a root invocation, used to determine an **AccessTable** for the invocation.

- **CallChainNode** - a component node of a **CallChainGraph**. Each node represents the effects of the invocation(s) of a particular **Method** operation on a particular target **SharedObject** instance.

- **MethodMetaInfo** - an object which stores an indexed representation of the various aspects of meta-information associated with its corresponding **Method** class.

- **SubinvocationFamily** - manages the multiplicity of call chain nodes stemming from a **SubinvocationDescriptor** based on a particular target field. There may be multiple potential target reference values for the field, and hence multiple corresponding subnodes.

- **RefFldDependency** - a meta-information object which represents an action (i.e. potential member assignment, subinvocation, return value) dependent on a particular **SharedObjRefFld** (parameter, subresult, member field, new instance) within a **Method** operation. Used in call chain node revision processing.

- **SmartSharedObjSet** - manages the set of potential values for a field that holds a reference to a **SharedObject**.

- **SmartSharedObjTable** - manages the set of potential values for a group of related fields that hold references to a **SharedObject**.

- **DirtyReferenceValue** - used to represent new reference values added to a **SmartSharedObjSet** or a **SmartSharedObjTable**.
Appendix A. Framework Class Overview

Figure A.1  Framework Class Inheritance Diagram
A.2 Framework Class Relationships

A.2.1 SharedObject Class Relationships

Figure A.2  SharedObject Class Relationship Diagram
A.2.2 Method Meta-information Class Relationships

Figure A.3  MethodMetaInfo Class Relationship Diagram
A.2.3 Invocation Class Relationships

![Diagram of Invocation Class Relationships]

Figure A.4 Invocation Class Relationship Diagram
Appendix A. Framework Class Overview

A.2.4 Access Decomposition Class Relationships

![CallChainNode Class Relationship Diagram](image_url)

Figure A.5  CallChainNode Class Relationship Diagram
Appendix B

Object Reference Management

This chapter details the three classes responsible for managing the multiplicity of potential values of references to shared objects used in the meta-layer's access decomposition mechanism.

B.1 Object Reference Management Classes

The meta-layer's access decomposition mechanism makes use of the following three collection classes to manage the multiplicity of potential values of references to shared objects within call chain nodes.

- **SmartSharedObjectSet** - used to manage potential return values.
- **SmartSharedObjectTable** - used to manage potential parameter values and results of subinvocations.
- **MemberFieldValueTable** - used to synchronise access to, and manage potential values of member fields of shared object targets.

These classes provide a mechanism for managing additions to the set of potential values of a field. This mechanism is used in call chain node maintenance during the iterative process of access decomposition. Once initialised, these collection classes maintain a list of 'dirty' values—new reference values added to the collection which were not present the last time the collection's owner node observed the values. After node revision processing has been performed to take the dirty values into account, the collection is then marked 'clean'. If the collection is subsequently updated ('dirtied') by other nodes, then the owner node must perform revision again. Once more, the subset of 'new' dirty reference values will be available for the owner node to take into account again.

The dirty list minimises the revision processing that each call chain node must perform when the potential values within the fields its interaction model depends on...
(parameters, subinvocation results, member fields) are updated by other nodes within the call chain graph.

### B.2 SmartSharedObjectSet

The SmartSharedObjectSet class manages a set of potential SharedObject reference values. It keeps track of additions to the set of values by maintaining a list of 'dirty' values. It is used within the scheme's access decomposition process to manage the potential return values for call chain nodes representing Method operations that return a SharedObject.

Figure B.1 illustrates the class relationships for the SmartSharedObjectSet class and the operations it provides to:

- add new reference values to the collection
- retrieve the current set of reference values from the collection
- determine whether there are any 'dirty' reference values in the collection
- retrieve only the 'dirty' reference values from the collection
- mark the collection as 'clean'

![SmartSharedObjectSet Class Diagram](image)

**Figure B.1 SmartSharedObjectSet Class Diagram**

#### B.2.1 SmartSharedObjectSet Interface

The SmartSharedObjectSet class provides the following interface:

- boolean addValue(SharedObject o)

  - adds the shared object reference o to the set of values. If the value did not already exist in the set, then a DirtyReferenceValue object is created,
Appendix B. Object Reference Management

added to the list of dirty values, and true is returned. Returns false if the value already existed in the set.

* boolean addValues(Vector v)
  
  - adds a set of shared object references to the list in the same manner as above. Returns true if any new values were added to the set.

* Vector getValues()
  
  - returns the current set of potential SharedObject reference values.

* boolean dirty()
  
  - returns true if there are any dirty values in the set, false if there are no dirty values.

* Vector getDirtyValues()
  
  - returns the current set of DirtyReferenceValue objects.

* void cleanup()
  
  - removes all DirtyReferenceValue objects from the set, marking the set clean again. This does not remove any values from set.

B.2.2 SmartSharedObjectSet Object Lifecycle

A SmartSharedObjectSet is created if necessary during the call chain node construction process. The meta-layer examines the MethodMetaInfo to see if the node’s corresponding Method operation returns any references to shared objects. If so, then a new instance of SmartSharedObjectSet is created.

A SmartSharedObjectSet instance may be updated by its owner node during node revision processing. Caller’s of the owner node are notified when the SmartSharedObjectSet instance becomes dirty, and will examine its dirty list as part of their own revision processing.

A SmartSharedObjectSet instance is destroyed when its owner call chain node object is destroyed, during the invocation’s post-execution processing.

B.3 SmartSharedObjectTable

The SmartSharedObjectTable class is similar to the SmartSharedObjectSet class except that it manages the potential values of a group of related fields at once.
In call chain nodes representing Method operations which accept `SharedObject` parameters, a `SmartSharedObjectTable` instance is used to manage the potential values for all the parameters at once. Each field of the table corresponds to a particular formal `SharedObject` parameter to the Method – a `SharedObjParameter` meta-information object.

In call chain nodes representing Method operations which make subinvocations that return `SharedObject` result values, a `SmartSharedObjectTable` instance is used to manage the potential values for all the subinvocation results at once. Each field of the table corresponds to a particular subinvocation result – a `SharedObjSubResult` meta-information object.

Figure B.2 illustrates the class relationships for the `SmartSharedObjectTable` class and the operations it provides to:

- add a new field to the table
- add new reference values for a particular field to the table
- retrieve the current set of reference values for a particular field
- determine whether there are any 'dirty' reference values in the table
- retrieve only the 'dirty' reference values from the table
- mark the collection as 'clean'

![SmartSharedObjectTable Class Diagram](image)

**Figure B.2 SmartSharedObjectTable Class Diagram**

### B.3.1 SmartSharedObjectTable Interface

The `SmartSharedObjectTable` class provides the following interface:

- `void addField(SharedObjectReferenceField f)`
Appendix B. Object Reference Management

- adds an entry into the table to store potential shared object reference values for field f.

• boolean addValue(SharedObjectReferenceField f, SharedObject o)
  - adds the shared object reference o to the set of values for field f. If the value did not already exist in that set, then a DirtyReferenceValue object is created, added to the list of dirty values, and true is returned. Returns false if the value already existed in the set.

• boolean addValues(SharedObjectReferenceField f, Vector v)
  - adds a set of shared object references to the list in the same manner as above. Returns true if any new values were added to the set.

• Vector getValues(SharedObjectReferenceField f)
  - returns the current set of potential SharedObject reference values for field f.

• boolean dirty()
  - returns true if there are any dirty values in the set, or false if there are no dirty values.

• getDirtyValues()
  - returns the current set of DirtyReferenceValue objects.

• cleanup()
  - removes all DirtyReferenceValue objects from the set, marking the set clean again. This does not remove any values from set.

B.3.2 SmartSharedObjectTable Object Lifecycle

A SmartSharedObjectTable is created if necessary during the call chain node construction process. The meta-layer examines the MethodMethodInfo to see if the node's corresponding Method operation accepts any shared object parameters, or performs any subinvocations that return references to shared objects. If so, then new instances of SmartSharedObjectTable are created accordingly.
Appendix B. Object Reference Management

A parameter table may be updated by caller nodes during their node initialisation or revision processing. The owner node is notified if the parameter table becomes dirty, and will examine the parameter table's dirty list as part of its revision processing.

A subresult table may be updated by called nodes during their node initialisation or revision processing. The owner node is notified if the subresult table becomes dirty, and will examine the subresult table's dirty list as part of its revision processing.

Any SmartSharedObjectTable instances are destroyed when its owner call chain node object is destroyed, during the invocation's post-execution processing.

B.4 MemberFieldValueTable

The MemberFieldValueTable class is a subclass of the SmartSharedObjectTable, and is similar in that it manages the potential values for a group of related fields at once — the member fields of a particular shared object instance which contain references to other shared objects.

However, the MemberFieldValueTable class is responsible for another important area of functionality required for the safety of the access decomposition process. While the SmartSharedObjectSet and SmartSharedObjectTable classes manage the proliferation of potential reference values within the nodes of a single call chain graph, the same MemberFieldValueTable instance is accessible by nodes belonging to the call chain graph's of separate invocations. Hence, the MemberFieldValueTable class must ensure synchronised read/write access by nodes in different call chain graphs, according to the invocation ordering rule, to the member field reference values it contains.

Further, the SmartSharedObjectSet and SmartSharedObjectTable classes share the same pattern of a single owner/observer (the owner call chain node) with multiple contributors (other call chain nodes which supply additional input reference values). However, instances of the MemberFieldValueTable class do not have a single owner/observer node. Rather they are shared by all the call chain nodes which represent operations on the table's corresponding shared object instance. Hence, a single table may have multiple observer nodes, and must maintain a separate viewpoint of the dirty list for each observer node.

Figure B.3 illustrates the class relationships for the MemberFieldValueTable class and the additional operations it provides to:
Appendix B. Object Reference Management

- allow a node to request access (inspective or mutative) to the contents of the table
- allow a node to release access (inspective or mutative) to the contents of the table
- check a node's access status
- update a node's observational view point

Figure B.3  MemberFieldValueTable Class Diagram
B.4.1 Synchronising Access to Member Fields Between Competing Invocations

Access synchronisation is required to ensure that call chain models are not built using reference values within member fields of shared objects that may be modified before actual execution of the operation is scheduled, leading to an incorrect access table result. The access decomposition process for a particular invocation must be temporarily blocked if any member field access required conflicts with the member field access determined for other invocations either executing or to be scheduled for execution ahead of the invocation according to the invocation ordering rule. In accordance with the rest of the scheme, this access synchronisation is carried out under a readers/writer access policy.

The MemberFieldValueTable class uses access requests to record the types of access required to it at any one time by the various invocations concurrently within the meta-layer. As call chain nodes request access to a particular MemberFieldValueTable instance, it maintains an access request per root invocation which records the strongest form of access (read/write) required by the nodes of the call chain graph of that particular invocation. These access requests are stored in a sorted list according to the invocation ordering rule.

Node requests for access to the values in the table are handled by examining the list of requests. The invocation that the requesting node belongs to is determined, then its corresponding access request is located in the table's request list.

- If the access request for the invocation requires only read access, then the request list is checked for any higher order invocations which require write access. If there are no write requests from higher order invocations then the node is granted access. Otherwise access is denied at this stage.

- If the access request for the invocation requires write access, then the request list is examined to see if there are any higher order invocations at all. If there are no higher order invocations then the node is granted access. Otherwise access is denied at this stage.

Under certain circumstances, a node may request access to a table that results in the creation of an inconsistency in the request list\(^1\). For example, a new node in a high order invocation may request write access to a table which has previously granted read

---

\(^1\)See Section 3.6.3.
access to another invocation of lower priority. If the lower priority invocation has not begun execution yet, then its access must be revoked, and its call chain graph rebuilt. The MemberFieldValueTable class manages notification of any such interrupted invocations by flagging the corresponding call chain nodes as waiting.

B.4.2 Sharing Potential Values Between Multiple Observer Nodes

Within a particular call chain graph, there may be call chain nodes which represent different operations on the same target object. If a target shared object's member field values may be potentially modified, these new values must be observed by all of the call chain nodes which access that instance's member fields. However, each dependent node's point of observation occurs separately, hence the MemberFieldValueTable class must maintain separate viewpoint of the dirty list for each observer node.

The access request abstraction, which is used to record the strongest form of access required by the nodes associated with an invocation, is also used to record all the nodes within a call chain graph that have requested access to a particular MemberFieldValueTable instance. Associated with each dependent node is an observation value—a reference to the last DirtyReferenceValue object that the particular node last observed. The dirty list is maintained in order of new additional reference values, and in this way, each MemberFieldValueTable instance maintains a separate viewpoint of the dirty list for each of its observer nodes. When an observer node requests the set of dirty reference values, the table uses the observation value associated with the node to limit the set of values returned to only those that the node has not already observed.

B.4.3 MemberFieldValueTable Object Lifecycle

During the call chain node construction process, the meta-layer examines the MethodMetaInfo to see if the node's corresponding Method operation accesses any of the target objects shared object member fields. If so, then the target shared object is asked for a reference to its associated MemberFieldValueTable instance. If the table does not already exist, then the target creates a new instance and returns a reference to it. Otherwise, it returns a reference to the existing member field value table.

During initialisation processing, nodes request inspective or mutative access according to the details in the MethodMetaInfo. If access is granted, the meta-layer then examines the MethodMetaInfo to see which particular shared object member fields are accessed. The member field value table is then called to ensure that it currently
contains accurate values for those fields. If necessary, the table values are filled using Java's reflection features to access the actual values in the subject shared object.

During revision processing, nodes check their access status to ensure that their member field access has not been interrupted by another higher order invocation.

The contents of the member field value table may be updated by nodes containing member field assignments, during their node initialisation or revision processing. Other observer nodes within the same call chain graph are notified if the member field value table becomes dirty, and these other nodes will examine the table's dirty list as part of their revision processing.

A MemberFieldValueTable instance is not destroyed when call chain node objects which observe it are destroyed during the invocation's post-execution processing. Rather, a MemberFieldValueTable instance exists, if necessary, for the lifetime of its associated subject shared object. However, when call chain node objects are destroyed, they call the member field value table to release their access. This may allow the table to grant access to the waiting nodes of other lower order invocations.
Appendix C

Publications From This Thesis

This chapter provides a full reproduction of an earlier paper by the author which presented preliminary results of this research - "Synchronisation in Multithreaded Object-Oriented Programs: A New Scheme" in *Proceedings of The 3rd International Conference on Object-Oriented Technology (WOON'98)*, St. Petersburgh Electrotechnical University, St. Petersburgh, Russia, June 1998.

SYNCHRONISATION IN MULTITHREADED OBJECT-ORIENTED PROGRAMS: A NEW SCHEME

Michael Lawler
mj10@uow.edu.au

Department of Computer Science
The University of Wollongong, N.S.W. 2500, Australia.

Abstract. There are conflicts between object-oriented design ideals and current approaches to synchronisation in multi-threaded environments. Components often break encapsulation to achieve synchronisation, or fail to address the issues of atomicity and isolation for composite operations. This paper presents a high-level description of a new synchronisation scheme which addresses these conflicts. Invocations on shared objects are managed at run-time by a generic meta-level scheduler. Meta-information describing operations is used by the scheduler to reason about object interactions, allowing additional control over the scheduling and execution of concurrent operations.

1 Introduction

There are inherent conflicts between object-oriented design ideals and the necessities of reliable computation in a multi-threaded environment.

The major design benefits in an object-oriented (OO) approach derive from OO support for encapsulation and information hiding, and from dynamic binding and polymorphism [1] [2]. The public aspects of a class declaration specify the services provided by an instance of the class; the private description defines details of data owned and service mechanisms. If a class is well designed, a client cannot determine internal details of an instance and cannot become dependent on such details. An actual object may be an aggregate composed from many other objects, or may rely on collaborations with other independent objects when handling requests for its services, or may be actually an instance of some more specialized class quite unknown to the client; such details are
immaterial to a client which sees merely an object that can respond to varied requests for services. This enables the construction of complex software that is highly decoupled, whose elements are independent of each other, and therefore highly flexible, maintainable and reusable.

With intra-program concurrency, as realised through some threads mechanism, there are different concerns. In many cases, the operations of a thread are inherently dependent on the state of the components belonging to some other thread. For example, in a producer-consumer system, if the consumer components are unable to process or buffer data, the producer’s operations must be suspended; thus, correct operation of the producer requires knowledge of the state of a consumer. Similarly, when shared data must be updated by separate threads, these threads must coordinate their activities so as to avoid races leading to inconsistent updates, and must schedule their operations to avoid conditions such as deadlock.

Combining thread style concurrency with object orientation appears to lead to conflicts; a thread running in a method of one object may need to "break the encapsulation" of another object so as to examine its state, another thread might need to break the encapsulation of some data manager object in order to identify and lock all structures that might be affected by an update operation. It is often difficult to find the appropriate structure for and placing of the synchronisation code needed in such situations. Synchronisation code devised for one case may prove inappropriate for related situations, resulting in a loss in the reusability often claimed for OO approaches.

The combination of OO approaches and concurrency complicate the programmer’s task. But there are strong forces driving program development in the direction of combined OO/threaded-concurrency. Symmetric multiprocessor CPU systems are becoming more common and providing direct hardware support for such software architectures. Already, the effective operation of many programs such as browsers depends on the workings of multiple threads as are needed to provide interactive response while sustaining multiple network connections. This form of software architecture will become more common, with the overall operations of a program being realised through multiple threads handling subtasks. These threads may be partially independent, working with private constellations of objects, but will interact through shared data. The scheduling of thread access to shared data is a problem that programmers will encounter more frequently.

Automated mechanisms for resolving the synchronisation problems in an OO system would greatly benefit programmers. This paper presents some new ways of describing and invoking object operations that assist in the tasks of design and implementation of multithreaded OO programs. A proof of concept implementation in Java is also presented.

2 Issues in the Synchronisation of Composite Operations

Existing synchronisation mechanisms for concurrent object-oriented computing are based on the scheduling of client method invocations on a server object in a way that satisfies constraints on the permitted states of the server [11]. These existing mechanisms focus on the synchronisation invocations of a single server object. However, it is common for a method invocation on a server object to involve further nested interactions with other objects; these may be collaborators or components aggregated into the server object. As a result, it is often necessary to widen the focus when planning synchronisation mechanisms so as to take account of the complete set of objects involved in the execution of a composite operation.
Appendix C. Publications From This Thesis

For computations to be reliable, concurrent composite operations must often proceed atomically and in isolation[10]. Atomicity is the execution property that guarantees that a sequence of instructions executes or appears to execute indivisibly. If an operation is atomic, it is executed entirely or not at all. Isolation is the execution property such that the execution of multiple concurrent operations are serialised such that the result of each operation is identical to the result where it is executed by itself in isolation.

Synchronised access patterns must be planned for all objects participating in an operation so as to provide both atomicity and isolation [3] [7]. When only a single object is involved, a monitor style lock will suffice; but the approach of successively acquiring individual locks does not satisfy the requirements of atomicity and isolation in situations where an operation affects many objects. The requirements for atomicity and isolation must be considered with respect to the entire call chain structure and all the object interaction contained within. In these cases, synchronisation schemes that work in conjunction with traditional method invocation mechanisms generally have been unsatisfactory either because they fail to fully address the identified issues of atomicity and isolation, or they solve the problems in a manner that violates object-oriented encapsulation [6] [8].

A common approach for simple synchronization constraints places responsibility for control within the target object. Synchronisation code can be incorporated directly in the method implementation code, or can be represented as a separate description of a constraint on method execution (via the use of additional language features such as guarded methods, behaviour sets). Regardless of implementation, these schemes attempt to free the caller from the burden of synchronisation responsibility by letting the target of each operation handle the synchronisation issues at each invocation point.

The use of a simple monitor style lock on each object can provide automatic isolation for individual operations. But such locks do not guarantee isolation when a process involves many successive operations on the same object. Further, the mechanism is prone to deadlock in situations where an operation on a principal server object involves a nested series of calls to collaborating objects, each of which locks itself while executing a function. In both cases, correct operation requires the approach of "two-phase locking" [4]; locks must first be acquired on all objects that will be affected by a planned sequence of operations, these locks must be maintained while the operations are performed, and then all must be released. An individual target object cannot take the responsibility for locking itself for the duration of an indeterminate series of operation requests.

In an alternative approach, the client may assume synchronisation responsibility, by first arranging synchronization and then invoking the operation. However there are again problems. It is harder to guarantee isolation when separate mechanisms are used for the synchronization locking and method invocation. The coding of the client becomes complex because the programmer must forsee the ramifications of a planned sequence of operations and so determine the locks that must be acquired. If a request to a server object requires that it work with collaborator objects, then these too may have to be locked; identification of the affected objects may necessitate that the client "break the encapsulation" of the server object. Further, in a readers/writer style concurrency scheme, the characteristics of access mode of operations on target objects must also be identifiable. These various problems are typically addressed in an ad hoc way leading to complexity, unreliability, and lack of reuse.

Our aim is to largely automate the processes required for a client to organize synchronization and method invocation. The proposed system seeks to achieve:
Appendix C. Publications From This Thesis

(a) full compatibility with object-oriented language features:
- complete encapsulation of component objects
- dynamic binding and generalisation in client code
- work transparently with composition (aggregation/collaboration)

(b) automatic enforcement of access synchronisation

(c) atomicity and isolation for composite operations, without introducing deadlock opportunities.

A server object should be able to conceal details of its collaborators and aggregated sub-objects. However, since the correct scheduling of operations needs to take account of these other objects, some information must be made available so that a general automatic scheduling system can analyse, dynamically, the locking requirements for a desired operation. A server class must incorporate an interface with functions that provide a scheduler with necessary data.

Automatic enforcement of access synchronisation for operations requires transparent coupling of the synchronisation mechanism and the operation invocation mechanism. Synchronisation becomes a precondition to operation execution.

An automatic scheduler will have to organise some form of two phase locking mechanism for each requested operation on a server (and its collaborating objects). Before an operation can start, appropriate reader-writer style locks must be acquired for all affected objects. When all locks have been acquired, a scheduler can permit a thread to continue with the desired operation. A scheduler will permit different threads to proceed with concurrent non-interfering operations. A scheduler must delay requested operations if some of the objects potentially affected are either in use by currently executing threads, or are associated with prior operation requests, or are not in required states.

3 High Level Description of Proposed Scheme

3.1 Overview

The proposed synchronisation scheme operates in a heterogenous object environment, in which objects can be divided into two types:

- 'normal' objects
- 'shared' objects

A heterogenous environment allows some components of a system to be designed and built within the context of traditional sequential object models. Normal objects are defined and their methods are invoked via the standard mechanisms of the underlying language.

Only those parts of a system that involve independent concurrent access to shared objects require adaptation to the new synchronisation scheme. Operations on shared objects can only involve simple built in data types and references to other shared objects.

The synchronisation scheme for shared objects uses:
- a generic meta-level scheduling system
Appendix C. Publications From This Thesis

- a new way of defining the member functions of the classes of which the shared objects are instances
- a new mechanism for invoking operations on shared objects

A general invocation scheduling system is created as a meta-layer in the runtime architecture. This scheduling system is responsible for synchronising the concurrent execution of all operations on shared objects. When independently threaded tasks invoke operations on shared objects, control of each request is handed to the meta-layer for evaluation, scheduling and execution.

Shared objects must be instances of classes that can supply the scheduler with necessary "meta-information" for each possible operation. In these classes, operations ('member functions', 'methods') are bundled together with the descriptive meta-data allowing the scheduling system to reason about invocations.

When a thread of control requests execution of an operation on a shared object, the usual method call mechanism is replaced with a more elaborate mechanism that allows control over the synchronisation, scheduling and execution of the request. The first step in this mechanism involves reification of the request – an "Invocation"[14] object is created, its fields identify the server object and method involved along with the parameters for the actual method call. Invocation objects are automatically passed to the scheduler for processing.

The scheduler accesses the meta-information for the operation as provided by the server object referred to within the Invocation object. This information can be analysed by the scheduler both to identify whether read (non-exclusive) or write (exclusive) access is needed on an object, and to explore potential call chains resulting from nested object interactions.

The scheduler blocks an operation request until it can acquire simultaneously all of the appropriate locks for the complete set of objects involved. After this step, the operation can be performed without requiring further synchronisation for any nested actions. When the call chain returns, the scheduler releases the locks used and returns control back to the calling thread.

3.2 Shared Object Model

Objects, that are to be shared, must be instances of classes designed to work with the scheduling system. Such classes do not permit any public access to data members, nor do they define any conventional publicly accessible member functions. Instead, all publicly accessible parts of a class are defined using "Method" objects.

A Method object combines details of the actual function that will perform an operation (name, argument list, return type, body of implementation) and the additional meta-information describing scheduling requirements. In the current prototype Java-based implementation, Method objects are defined as "inner classes"[15] of the classes that represent shared objects. Both implementation and the meta-information are defined in terms of methods of the Method object. As an inner class, the code of a Method object implementation can act directly on the member fields of the shared object to which it belongs.
The meta-information declared for individual operations describes target access mode (e.g., non-exclusive (read) or exclusive (write) access to a target), and object interactions. The object interaction data allows the scheduler to determine all potential object interaction within a complete composite operation ("decompose the operation") prior to actual execution.

The aim of the decomposition mechanism is to determine all the possible subinvocations of operations upon shared objects. To this end, the meta-information provides the dynamic runtime system with details of:

1. possible subinvocations (identifying target, method, shared object parameter variables)
2. assignments of values to those member fields of a shared object that reference other shared objects
3. any references to shared objects returned as results of calls

Section 3.4.1, below, outlines the access decomposition mechanism that uses this meta-information to identify pessimistically all shared objects that might be involved in a particular operation. The invocation scheduling mechanism, described in section 3.4.2, arranges the logical acquisition of appropriate locks prior to the start of an operation.

### 3.3 Invocation Mechanism

The steps involved in the invocation of an operation on a shared object are:

1. **Invocation Reification:** a client request for an operation on a shared object is invoked via one of its Method objects; this results in the creation of an Invocation object.
2. **Meta-Level Interception:** the Invocation object registers with the meta-level scheduling system, the invoking thread gets to execute the control code of steps 3 to 5 before finally performing the desired operation, step 6, and then exiting from the meta layer, steps 7 and 8.
3. **Object Access Decomposition:** the scheduler uses meta-information about the requested operation to determine the complete set of objects and access modes required.
4. **Run-Time Scheduling:** The Invocation is scheduled against other concurrent Invocations based on the complete set of objects and access modes predicted for the entire invocation path.
5. **Access Lock:** The meta-layer acquires all access locks on the Invocation's behalf.
6. **Atomic Execution:** The entire invocation path is performed atomically and in isolation.
7. **Access Unlock:** The meta-layer releases the access locks.
8. **Control and return value are returned to the client.**
3.4 Scheduling Meta-Layer

3.4.1 Access Decomposition Mechanism

The access decomposition mechanism uses the method meta-information relating to a requested operation to construct a list of shared objects that may be involved in the operation so that these can be locked appropriately prior to the actual start of the operation. The mechanism works by building a representation of the potential call chain, taking into account all possible combinations of shared object and method call that might result from the initial "root" invocation.

The mechanism simulates execution, building nodes that correspond to the stack frames that will be created during actual execution. Each node represents the invocation(s) of a member function of a shared object. The meta-information for the object identifies operations that it may invoke, the potential targets of these operations must be determined dynamically. The potential targets will be shared objects referenced by member fields of the current object, or objects passed as parameters, or those referenced in return values of suboperations. The scheduler constructs a set representing the possible values for each object reference used. For example, if an operation selects an item from a collection, then the scheduler associates all objects in the collection with the reference representing the result of that selection operation. If the scheduler determines that an operation involves a reference variable that might be changed by a currently executing, or a previously decomposed but not yet executing operation, then the decomposition process is suspended until such previously scheduled activities have completed. Suboperations invoked are explored recursively for each potential target.

3.4.2 General Invocation Scheduling Mechanism

Once the set of objects and respective access modes involved in an invocation have been determined, execution of the invocation is then scheduled against other competing concurrent invocations.

The generalised scheduler maintains a list of currently executing invocations, and a queue of waiting invocations. When a new invocation is introduced, the scheduler compares the object access set of the new invocation with that of the invocation at the tail of the waiting queue. An access conflict occurs between two invocations if they require conflicting access to the same target object. For example, if one invocation requires write/exclusive access while the other requires read/non-exclusive access then there is a conflict - the two invocations must not be allowed to execute concurrently. If there are no access conflicts detected between the invocations, then the new invocation advances up the queue, and again is compared with the next queued invocation. The new invocation is inserted behind the first conflicting invocation encountered.

When an invocation reaches the head of the waiting queue, a similar comparison is made between the set of objects that it affects and those of the invocations currently executing. If the head invocation does not conflict with any executing invocations, it is removed from the queue and added to the list of executing invocations and actual execution of the operation starts. When an operation finishes execution, it is removed from the list of executing operations, and the scheduler re-examines the waiting queue for possible promotions.

The list of executing invocations and their object access modes forms a set of logical locks which guarantee the safety and isolation of concurrent operations. The meta-level
Appendix C. Publications From This Thesis

scheduler alone manages object access for invocations via a "two-phase locking" style, and so guarantees atomicity and prevents deadlock.

The waiting queue and the order it imposes on conflicting operations ensures fairness for concurrent operations. No thread can starve out another because a FIFO order is imposed. Conversely, the analysis of access sets allows the scheduler to execute non-conflicting operations in an optimally concurrent fashion. If an independent thread requests an operation that does not conflict with any waiting or executing operations, then it may be executed immediately.

4 Example: Dining Philosophers

The dining philosophers problem provides a simple illustration of multiple processes competing for specific subsets of a limited resource.

4.1 Problem

Dijkstra's Dining Philosophers problem[16] involves five philosophers sitting around a circular table. Each philosopher spends his time alternatively thinking and eating. Each philosopher has a fork to his left, and must borrow a fork from the philosopher on his right in order to eat. The philosophers must find some way of sharing their forks in such a way as to avoid starvation and deadlock. Starvation occurs if a philosopher never gets access to both of the forks he needs and so never gets to eat. Deadlock occurs if all philosophers are holding one fork and refusing to give it up, while attempting to acquire another.

4.2 Solution

Solving the Dining Philosophers problem requires coordinating access to the forks in a safe and fair manner. This example solution involves an independently threaded class PhilosopherTask, and two shared object classes Fork and Philosopher. The structure of the solution is illustrated in Fig. 1.

![Diagram of Dining Philosophers solution](image)

The PhilosopherTask class describes a philosopher's behaviour. As illustrated in Fig. 2, it owns a member field, phil, to refer to a particular Philosopher instance, and a method, run(), which defines the task procedure.
Philosopher extends Thread

```
Philosopher phil;
run()
```

while(true) {
    phil.think();
    phil.eat();
}

Fig. 2. PhilosopherTask

Philosopher is a shared object class for representing individual philosophers. As illustrated in Fig. 4 it owns two member fields, `leftFork` and `rightFork` which refer to the two Fork instances it uses, and Method objects representing its `think()` and `eat()` operations. The `eat()` implementation involves nested invocations on the two Fork objects. The meta-information for this operation declares the access mode, and details of the `pickUp()` and `putDown()` subinvocations.

```
Philosopher extends SharedObject
Fork leftFork;
Fork rightFork;

think() meta-information
implementation

eat() meta-information
implementation
```

Fig. 3. Philosopher

Fork is a shared object class for representing an individual fork. As illustrated in Fig. 4, it contains no state fields, only Method objects representing it's `pickUp()` and `putDown()` operations. The meta-information characterises both operations as requiring exclusive access.

```
Fork extends SharedObject

pickUp() meta-information
implementation

putDown() meta-information
implementation
```

Fig. 4. Fork

A main procedure constructs the five instances of each class. Each Philosopher object is linked to the Fork objects on its left and right, and each PhilosopherTask object is linked to an individual Philosopher object. Each PhilosopherTask object continually invokes the `think()` and `eat()` operations on its respective Philosopher object.
When a PhilosopherTask object invokes `Philosopher.eat()`, the `eat()` Method object creates an Invocation object which is processed by the scheduler. The access decomposition mechanism (section 3.4.1) examines the meta-information for the `eat()` operation. Firstly, it will require exclusive access to the target Philosopher object. Secondly, using the object references in the `leftFork` and `rightFork` member fields and the meta-information defined for `Fork.pickUp()` and `Fork.putDown()`, it determines that exclusive access is also required to both Fork objects. The Invocation object is then scheduled for execution against competing concurrent invocations by the general invocation scheduling mechanism (section 3.4.2). Deadlock and interference are avoided because the scheduler ensures safe lock acquisition. Starvation is avoided because the scheduler imposes a FIFO order on conflicting concurrent operations.

4.3 Benefits of Solution

This solution of the Dining Philosophers problem has the following benefits:

- No synchronisation primitives are needed in implementation code since access synchronisation is declared at a higher level via meta-information.
- Components are highly decoupled. Task level code (PhilosopherTask) is not aware of the access requirements inherent in the Philosopher.eat() operation.
- Component classes are reusable in different access scenarios.
- The system is extensible. For example, additional components and interactions may be added to the system without reworking of the existing synchronisation mechanisms.

5 Discussion

5.1 Applicability

The proposed scheme is applicable in scenarios where multiple independent threads of control require synchronised access to shared data objects. The scheme is useful for providing atomicity and isolation over composite operations when static mechanisms or structural exclusion techniques do not address the dynamic nature of operations. This may occur when the set of objects involved, or the levels of nesting, can only be determined at runtime. These situations arise naturally from the use of the OO principles of encapsulation and generalisation.

5.2 Consequences

5.2.1 Atomicity vs Concurrency

All interactions with shared data are serialised as a sequence of 'root' invocations on shared objects. A root invocation occurs when a thread not executing under control of the scheduler requests an operation on a shared object. The resulting root invocation encompasses any further subinvocations within the operation execution. The scheduler automatically enforces atomicity and isolation over the scope of each root invocation. This constrains concurrent execution of competing operations. Hence the atomicity of operations (root invocations) must be traded off against the desired granularity of concurrency.
5.2.2 State-Dependent Synchronisation

Guarded methods[5] allow explicit state-dependent synchronisation conditions to be defined for individual object operations, and enforce these conditions by blocking requests at invocation time until the conditions are met. In the case of composite operations, this can result in successive incremental synchronisation steps at each sub-invocation.

State-dependent synchronisation in the form of guarded methods is incompatible with the two-phase locking approach of this scheme. It is meaningless for the scheduler to reason about future object state before execution of a root invocation. In addition, the approach of locking all objects involved in an operation precludes all other threads from changing those objects' states, once the operation is started.

Section 5.4 briefly introduces a mechanism for managing state-dependent synchronisation with this scheme.

5.3 Related Schemes

5.3.1 Active Object

Lavender and Schmidt [12] describe the Active Object pattern which decouples method execution from method invocation in order to simplify synchronised access to an object by independent threads. This pattern shares similar concepts to the scheme proposed in this paper, such as the reification of operation requests (termed "Method objects"), and an explicit synchronisation mechanism which controls scheduling and execution of such requests. The Active Object pattern describes a well known model (Actors) which has been used in concurrent programming for some time.

However, unlike the scheme proposed in this paper, the Active Object pattern does not consider composite operations which span multiple active objects [9]. Rather, it focuses on aspects of single object operations. Hence, issues of atomicity, isolation and deadlock in such operations are not addressed.

5.3.2 Structured Transactions

Lea [5] describes transaction-based techniques that can be applied in general purpose concurrent programming contexts to achieve interference-free execution for multi-object operations. In these models, each class supports a standardized transaction protocol which propagates control through successive objects involved in composite operations. Each method requires a transaction control argument (in addition to normal arguments), and is responsible for using this to manage and isolate actions in accordance with a given policy. These techniques can ensure isolation and guarantee atomicity for arbitrary composite operations over multiple objects, however the scheme suffers from the way access to the objects involved proceeds progressively with the call chain.

In optimistic schemes, sub-operations attempt to perform actions, but rollback to initial states upon detection of interference from other threads. This can be problematic, because transactions may continually abort due to continual interference, and because the consequences of some operations cannot be rolledback.

Alternatively, in pessimistic schemes sub-operations incrementally acquire locks. This avoid interference, but can result in deadlock.
The scheme proposed in this paper avoids these problems via the access analysis of operations and a "two-phase locking" style.

5.3.3 Concurrent Eiffel

Meyer[13] describes the SCOOP approach to concurrency via extensions to the Eiffel language. The approach introduces an architecture where objects are structurally attached to particular processors (threads), and additional constraints and invocation semantics for managing interactions.

A static concurrency structure between classes is enforced - components must be coded in terms of their relative location to the objects with which they interact, preventing components from being reused in different interaction scenarios. Operations on objects that are attached to the same processor are executed in mutual exclusion, preventing intra-object concurrency such as the readers/writer policy.

To provide atomicity and isolation over composite operations involving separate processors, the scheme involves a blocking reservation mechanism which requires that the targets of such calls be formal parameters of their enclosing routines. However, atomicity and isolation over a dynamic set of objects may not be achievable with a static parameter list. A client must pass as parameters, all the objects to be reserved, but identifying these objects before actual invocation may break encapsulation. Incrementally reserving objects as suboperations are performed hides the nested reservation requirements from the client, but does not provide atomicity and isolation over the root operation and introduces the possibility of deadlock.

In contrast to the static explicit nature of the SCOOP reservation mechanism, the scheme proposed in this paper provides reservation dynamically and implicitly via meta-information on operations.

5.4 Further Developments and Examples

The prototype implementation of the synchronisation scheme extends the concepts described here in a number of ways:

- **State-Dependent Synchronisation** is handled by a mechanism that allows clients to "wait" for a certain "condition" on a shared object before executing an attached operation. The scheduling mechanism accepts a form of invocation which describes a pre-condition relative to a root operation. In this manner, a singular level of guarding per composite operation is achieved, while still retaining atomicity and isolation for operations via two-phase locking.

- **Access Decomposition Optimisations** allow caching of the various results of operation decomposition. In practice, the most operations do not involve dynamic object access patterns, and accordingly, the meta-layer can be optimised to use cached decomposition information for operations involving static object access patterns.

- **Asynchronous Invocations / Futures** involve only simple extensions to the scheme. After reification of an Invocation, a separate thread can be created automatically to process the operation asynchronously. Invocations return a ReturnValue object which can be extended to support 'Future' style semantics.
Appendix C. Publications From This Thesis

Working examples of solutions to classical synchronisation problems, including examples of state-dependent synchronisation, are available on the World Wide Web.

- **Dining Philosophers**  
  http://www.uow.edu.au/~mjll0/research/diners
- **The Smoker's Problem**  
  http://www.uow.edu.au/~mjll0/research/smokers
- **The Sleeping Barber**  
  http://www.uow.edu.au/~mjll0/research/barber

References


Bibliography


[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.


234. Department of Computer Science, University of York, York, United Kingdom, June 1994.


