SEPARATION OF CONCERNS IN CONCURRENT PROGRAMS USING FINE GRAINED JOIN POINTS

A thesis submitted to the University of Manchester for the degree of Master of Philosophy in the Faculty of Engineering and Physical Sciences

2012

By
Chenchen Xi
School of Computer Science
# Contents

Abstract 8

Declaration 9

Copyright 10

Acknowledgments 11

1 Introduction 12

1.1 Aspect-oriented programming 13

1.1.1 Concerns and software design 13

1.1.2 Crosscutting concerns and aspects 14

1.2 Code tangling in concurrent programs 15

1.2.1 Explicit synchronization 15

1.2.2 Compiler directives for synchronization 16

1.2.3 Separation of concerns in concurrent programs 18

1.3 Java-based concurrent computation 19

1.4 Challenges and contributions 20

1.4.1 Aspects for concurrent computation 20

1.4.2 AspectJ and beyond: join points for complex behaviour 21

1.5 Outline 22

2 Aspect Oriented Programming 23

2.1 Motivation 23

2.2 Concepts 27

2.3 AspectJ Language 30

2.3.1 Other Languages 31

2.4 Tools:abc 34
3 Concurrent Concerns 38

3.1 Aspect-oriented model for loops 39
3.1.1 The loops model in Java 39
3.1.2 Example to separate a loop concerns in OO 41
3.1.3 An extended loops join point 42

3.2 Aspect-oriented model for synchronization 47
3.2.1 The synchronization model in Java 47
3.2.2 synchronized block 47
3.2.3 synchronized method 48
3.2.4 An object-oriented model for synchronization 49
3.2.5 synchronization join point 52

3.3 Miscellaneous 53
3.3.1 User defined basic block 55
3.3.2 Brief draft of basic block 56

3.4 Summary 57

4 Fine grained concurrent join points 59

4.1 Fine grained join point models for concurrent programs 60
4.1.1 Aspects for the Java-Grande Forum benchmarks 60
4.1.2 Source code versus bytecode 62
4.2 The extended synchronized block join point model 66
4.2.1 synchronized block in general case 67
4.2.2 synchronized block with extra exit nodes 68
4.2.3 synchronized block body join point 69
4.2.4 Context exposure 70
4.2.5 Implementation in abc: SynAJ and SynBodyAJ 73
4.2.6 Aspects for synchronization 77

4.3 Related work 78

4.4 General block 79
4.4.1 General block join point 80
4.4.2 Aspects for general block 81

4.5 If-Then-Else block 82
4.5.1 If-Then-Else join point 84
5 Applications and evaluation 87
5.1 Aspects for flexibility in implementing synchronization strategies . 89
5.2 Applications for logging locked time . . . . . . . . . . . . . . . . 92
5.3 Aspects for synchronized block selection . . . . . . . . . . . . . 93
5.4 Thread control and rescheduling . . . . . . . . . . . . . . . . . . 96
5.5 Converting synchronized blocks into transactions . . . . . . . . 97
5.6 ConTest reimplementation for concurrent program debugging . . . 102
5.7 Evaluation Metrics . . . . . . . . . . . . . . . . . . . . . . . . . 106
5.7.1 Evaluation criteria for coupling, cohesion, and conciseness 106
5.7.2 Evaluation criteria for separation of concerns . . . . . . . . 109
5.7.3 Summary . . . . . . . . . . . . . . . . . . . . . . . . . . . 110
5.8 Distributed lock migration . . . . . . . . . . . . . . . . . . . . . . 110
5.9 Record and Playback (R&P) for concurrent program testing . . . 113
5.9.1 Separation of Concerns Measures . . . . . . . . . . . . . . . 117
5.9.2 Size Measures . . . . . . . . . . . . . . . . . . . . . . . . . . 118
5.9.3 Coupling and Cohesion Measures . . . . . . . . . . . . . . . 119
5.10 Summary . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 121

6 Conclusions 123
6.1 Contributions . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 123
6.1.1 Contributions to concurrent programming . . . . . . . . . . 125
6.1.2 Contributions to aspect-oriented programming . . . . . . . . 125
6.1.3 System modularity evaluation . . . . . . . . . . . . . . . . . 126
6.2 Critique . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 127
6.3 Related and future work . . . . . . . . . . . . . . . . . . . . . . . 128

Bibliography 130

A AspectJ syntax guide 141
A.1 General structure of aspects . . . . . . . . . . . . . . . . . . . . . . 141
A.2 Inter-type declarations . . . . . . . . . . . . . . . . . . . . . . . . 142
A.3 Pointcut descriptors . . . . . . . . . . . . . . . . . . . . . . . . . . 142
A.3.1 Pointcut definition . . . . . . . . . . . . . . . . . . . . . . . . 145
A.3.2 Context exposure . . . . . . . . . . . . . . . . . . . . . . . . . 146
A.3.3 Primitive pointcuts . . . . . . . . . . . . . . . . . . . . . . 147
A.3.4 Matching . . . . . . . . . . . . . . . . . . . . . . . . . . . . 154
A.4 Advice . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 157

B UML for Original R&P 159
List of Tables

4.1 Comparison of weaving capabilities. . . . . . . . . . . . . . . . . . 79
# List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.1</td>
<td>Control flow graph for two nested loops with a break statement jumping outside the outer loop.</td>
<td>45</td>
</tr>
<tr>
<td>3.2</td>
<td>Examples of two block join points ((a),(b)) and corresponding computational flow model (c).</td>
<td>56</td>
</tr>
<tr>
<td>4.1</td>
<td>Control flow graph of simple synchronized block.</td>
<td>67</td>
</tr>
<tr>
<td>4.2</td>
<td>Control flow graph of nested synchronized blocks.</td>
<td>68</td>
</tr>
<tr>
<td>4.3</td>
<td>Synchronized block (a) and associated code without synchronization (b).</td>
<td>71</td>
</tr>
<tr>
<td>4.4</td>
<td>Control flow graph for synchronization with multiple exits.</td>
<td>71</td>
</tr>
<tr>
<td>4.5</td>
<td>Control flow graph without synchronization.</td>
<td>72</td>
</tr>
<tr>
<td>4.6</td>
<td>Control flow graph of simple block.</td>
<td>81</td>
</tr>
<tr>
<td>4.7</td>
<td>Control flow graph of if-then-else block.</td>
<td>84</td>
</tr>
<tr>
<td>5.1</td>
<td>Performance results for different selections schemas.</td>
<td>95</td>
</tr>
<tr>
<td>5.2</td>
<td>Bugs revealed for different implementations of ConTest: Bubble Sort.</td>
<td>104</td>
</tr>
<tr>
<td>5.3</td>
<td>Bugs revealed for different implementations of ConTest: 2’d Bubble Sort.</td>
<td>104</td>
</tr>
<tr>
<td>5.4</td>
<td>Bugs revealed for different implementations of ConTest: ID.</td>
<td>105</td>
</tr>
<tr>
<td>5.5</td>
<td>Bugs revealed for different implementations of ConTest: Double Lock.</td>
<td>105</td>
</tr>
<tr>
<td>5.6</td>
<td>Concern measure for distributed lock migration.</td>
<td>112</td>
</tr>
<tr>
<td>5.7</td>
<td>Concern measures for Record and Playback.</td>
<td>118</td>
</tr>
<tr>
<td>5.8</td>
<td>Size measure for Record and Playback.</td>
<td>119</td>
</tr>
<tr>
<td>5.9</td>
<td>Coupling measures for Record and Playback.</td>
<td>120</td>
</tr>
<tr>
<td>B.1</td>
<td>Package for Original R&amp;P Instrumentation</td>
<td>159</td>
</tr>
</tbody>
</table>
Abstract

With the advent of multicore processors, there is an increasing amount of interest in building concurrent applications capable of fully utilising their features. Developing applications for these platforms to take full advantage of the power of multicore capabilities remains a complex, error-prone, and challenging endeavour.

Unfortunately, concurrency is not uniformly and externally expressed in most existing application models. The result is that concurrency and thread management are hidden within objects or components and intermixed with their functionalities. After demonstrating the tangling of code that results from the interaction of these two major concerns, this thesis describes a method of improving the concurrent program by applying a novel technique to separate the concerns: aspect-oriented programming, which aims to encapsulate concerns that crosscut the main program flow in separate entities into aspects.

The most mature aspect-oriented tool available at the time this project was being undertaken is AspectJ, which is an extension of Java. AspectJ can be used to write an aspect to a separate concern in a sequential program to avoid code tangling, but it is often inapplicable for concurrent programs. The problem lies in the fact that the points where parallelisation should occur are not natural join points in AspectJ (i.e. points where AspectJ can intervene). Consequently, this thesis proposes a set of fine-grained join points capable of completely handling concurrent programs. This model goes beyond present AspectJ models and demonstrates the need to recognise complex behaviour for an effective separation of concerns.

Finally, aspects for implementing concurrent programs according to different schemes are presented, together with evaluation results. This highlights the flexibility of aspects for implementing concurrent programs, a flexibility which is always a cross-cutting concern with respect to the main concern of base applications.
Declaration

No portion of the work referred to in this thesis has been submitted in support of an application for another degree or qualification of this or any other university or other institute of learning.
Copyright

i. The author of this thesis (including any appendices and/or schedules to this thesis) owns any copyright in it (the “Copyright”) and s/he has given The University of Manchester the right to use such Copyright for any administrative, promotional, educational and/or teaching purposes.

ii. Copies of this thesis, either in full or in extracts, may be made only in accordance with the regulations of the John Rylands University Library of Manchester. Details of these regulations may be obtained from the Librarian. This page must form part of any such copies made.

iii. The ownership of any patents, designs, trade marks and any and all other intellectual property rights except for the Copyright (the “Intellectual Property Rights”) and any reproductions of copyright works, for example graphs and tables (“Reproductions”), which may be described in this thesis, may not be owned by the author and may be owned by third parties. Such Intellectual Property Rights and Reproductions cannot and must not be made available for use without the prior written permission of the owner(s) of the relevant Intellectual Property Rights and/or Reproductions.

iv. Further information on the conditions under which disclosure, publication and exploitation of this thesis, the Copyright and any Intellectual Property Rights and/or Reproductions described in it may take place is available from the Head of School of School of Computer Science (or the Vice-President).
Acknowledgments

I would like to express my thanks to my supervisor, Professor John R. Gurd for his invaluable advice and guidance during three years. Many thanks to Dr. Bruno Harbulot for the advice and help he provided. I would also like to thank the members of the Center for Novel Computing, at the University of Manchester, and in particular those with whom I shared a friendly office for three years.

I would like to acknowledge funding from the Department of Computer Science - via Teaching Assistant position, and again my supervisor, who helped me actively bidding for any funding opportunity.

Finally, to my family, thank you for your never-ending support.
Chapter 1

Introduction

Recent advances in parallel computing algorithms and concurrent patterns, such as map/reduce, pipeline, and fork/join, can help to simplify the development of applications for multicore processors. However, in most existing application models, concurrency is not uniformly and externally expressed, with the result that concurrency and thread management are hidden within objects or components and intermixed with their functionalities.

This thesis addresses the problems of improving the programming model in concurrent programs, in order to decouple the various concerns that must be dealt with in parallel computing. Toward this aim, techniques are presented that assist the separation process. These techniques are important because it is essential to avoid code tangling in concurrent programs for many scientific software development and maintenance tasks such as debugging and testing, and it is difficult for existing tools to encapsulate the concurrent concern separately from the base code.

Traditional parallel applications suffer from classic tangling problems [KLM+97], since parallelisation concerns cut across multiple application modules. Core functionality (i.e. domain-specific logic) is usually mixed with parallelisation concerns, which include partitioning work into parallel tasks, the concurrent execution of those tasks, the synchronization of parallel access to shared data structures to avoid data races, and the distributed execution of the tasks. The code related to these concerns is tangled with an application code, which makes it more difficult to understand, reuse, and evolve the core functionality and parallel code. Traditional parallel programming is mainly focused on performance issues. Aspect-oriented
programming (AOP) [KLM97] contributes to reconciling high-level and high-performance computing, by modularising the above concerns more effectively [SR02].

It is well known that concurrent programming, or multithreaded programming using traditional object-oriented approaches such as through C++ or Java thread packages, is difficult and requires highly skilled developers. Such multithreaded programs are not only error prone but also very difficult to debug and test. The difficulty lies in the fact that concurrency control is embedded in objects and components in these traditional models. Furthermore, functional and non-functional behaviours, such as thread synchronization, are not separately addressed. The result is that the management of the concurrency and thread synchronization becomes intermixed with, and hidden inside, functional behaviours.

### 1.1 Aspect-oriented programming

Software engineering aims to provide clear processes and mechanisms for designing and maintaining software. These processes are driven by the interaction between diverse concerns. Software engineering mechanisms are intended to provide programmers with a means to abstract and encapsulate these concerns and interactions. The software engineering paradigm AOP aims to deal with so-called “crosscutting concerns”, as described in the following subsections.

#### 1.1.1 Concerns and software design

As defined in [IEE00], “concerns are those interests which pertain to the system’s development, its operation or any other aspects that are critical or otherwise important to one or more stakeholders”. Concerns can be more generally defined as “any matter of interest in a software system” [SR02].

Designing a piece of software involves abstracting the relevant concerns into the appropriate constructs. Using procedural languages such as C, Pascal, or Fortran, designers encapsulate concerns into functions, procedures, or subroutines, which can then be grouped into modules with various degrees of interdependency. In object-oriented programming, the use of classes and inheritance provides a further degree of abstraction. Functional, procedural, or object-oriented programming languages have a common way of providing mechanisms for abstraction:
concerns placed into functions, procedures, or objects can be seen as being functional units of the system [KLM+97].

Methods, classes, and libraries are object-oriented constructs for encapsulating concerns at several degrees of granularity. An object-oriented design aims to make concerns to match these constructs. As far as possible, this consists of mapping the concerns involved into a set of objects, with associated actions, linked by inheritance (is-a) or by composition (has-a) relationships. As a result, the links between the objects represent the relationship between the purposes of differing concerns. However, this form of decomposition into functional units is not suitable for all types of concerns.

1.1.2 Crosscutting concerns and aspects

Some concerns interact with each other in such a way that they cannot be properly encapsulated within object-oriented constructs. These interactions lead to code tangling [FF02], where the elements of code of two concerns are in the same unit and cannot be dissociated, or to code scattering [FF02], when a concern involves code spread across several units. Two concerns that are related in such a way that they imply code scattering or code tangling are said to crosscut one another. A concern that crosscuts the main purpose of a unit is a crosscutting concern (with respect to that unit’s decomposition).

For example, in a system that provides its users with an e-mail service and a file repository service, the two sets of objects supplying those services will have an authentication concern in common. At least two units (sets of objects) will contain statements for this authentication concern. This is an example of code scattering. If monitoring certain activities in a specific context is required, statements will be added between the functional code (which performs what is to be monitored). This is an example of code tangling. More generally, tracing and logging are typical examples of concerns that almost always crosscut the main component’s purpose: the concern of tracing the behaviour of a component is different from the main concern implemented by this particular component. Further examples of code tangling in the context of concurrent programs are presented in Section 1.2.

AOP [KLM+97] is a programming paradigm aimed at improving the separation of concerns in software. In AOP, functional, procedural, and object-oriented paradigms are augmented with a means for separately encapsulating crosscutting
concerns. In an aspect-oriented design, crosscutting concerns are encapsulated into aspects. Chapter 2 provides a further description of aspect-oriented programming and its mechanisms.

1.2 Code tangling in concurrent programs

Concurrent applications are often focused on two concerns: the algorithm used by the calculation itself (related to the scientific model) and the parallelisation of its implementation. These are two separate concerns. However, in current procedural or object-oriented implementations, these elements of code are usually interlaced within the same unit of the system design. This is an example of code tangling that occurs as a direct consequence of scientific-model concerns and high-performance concerns crosscutting each other.

One example where crosscutting arises is in a code written to be executed in parallel for achieving high performance. Section 1.2.1 provides an overview of the crosscutting involved when the synchronization in the concurrent program has to be explicitly managed. Section 1.2.2 shows that hot compiler directives enable the programmer to avoid the explicit coding of synchronization. Section 1.2.3 demonstrates how, through analysis of the implementation of the same algorithms, the concern of synchronization crosscuts the calculation concern which is the main purpose of the programs. This section illustrates the drawbacks of the design of these examples in terms of modularity.

1.2.1 Explicit synchronization

Explicit synchronization can be implemented using languages such as C (for example, by explicitly managing the interaction between Unix-style processes). However, this type of programming is not often used directly for concurrent applications, which more commonly implement parallelism using compiler directives.

Java is the language used for most examples in this thesis. The default for implementing synchronization in Java is a form of explicit synchronization. Java provides multithreading natively through its Thread class. Concurrent instances of Thread can be executed over several processors, using currently available Java Virtual Machines (JVM). The language and application programming interface (API) mechanisms used for implementing several concurrent threads rely on either
extending the `Thread` class or implementing the `Runnable` interface [CWH00, OW04].

Parallelising a sequential program requires both a specific design using the Thread-related classes and the explicit instantiation and synchronization of those threads. This usually implies a refactoring of the objects describing the computation [OW04]. As a result, elements of the computation are placed within constructs defining the threads, and statements for managing the threads are placed within the units originally dedicated to describing the computation, as shown in Listing 1.1.

This example takes the classical producer–consumer problem as a case study. In this problem, the producer places the product into the buffer, from which the consumer obtains the product. If the buffer is not full, the producer goes on producing until it is full. If the buffer is not empty, the consumer may continue to consume the product until the buffer is empty. This implementation uses the Java programming language. Java implements the synchronization by the keyword “synchronized” and the methods `wait()` and `notify()`. A partial code for the producer-consumer problem in Listing 1.1 shows the class `CubbyHole` and the main method `put()`. The code of method `get()` is abbreviated. Since the instructions for the original algorithm function concern and the multithreading synchronized concern cannot be clearly separated from each other, this is an example of both code tangling and code scattering.

1.2.2 Compiler directives for synchronization

In general, explicit synchronization is difficult. In concurrent applications, synchronization is often implemented using compiler directives or APIs (for example OpenMP [CDK+01]). These techniques allow the programmer to synchronize protected data with little modification to the original structure and to hide extra complexity caused by the explicit creation of processes and threads.

OpenMP provides a set of compiler directives and library functions that can be used in C, C++, and Fortran to parallelise programs for shared-memory multiprocessor machines. There is also an equivalent for OpenMP for Java: JOMP [BK00]. Listing 1.2 shows a basic example of synchronization with OpenMP in C. The OpenMP directive is used to give each parallel thread exclusive access to a shared variable until its current update is completed.

Other compiler directive approaches for implementing parallelism have been
CHAPTER 1. INTRODUCTION

Listing 1.1: Code tangling in producer–consumer implementation

class CubbyHole {
    // the length of buffer
    private int[] goods = new int[4];
    private int front;
    private int rear;
    // the number of products
    private int count;
    // the number of blocked consumers
    private int getter;
    // the number of blocked producers
    private int putter;
    public CubbyHole(){
        front=0;rear=0;count=0;putter=0;getter=0;
    }
    public synchronized int get(int id) {...}
    public synchronized void put(int value, int id) {
        if(count>=4){
            putter++;
            try {wait();} catch (InterruptedException e) {} 
        }
        rear=(rear+1)%4;
        goods[rear]=value;
        System.out.println("Producer "+id+" put: "+
        goods[rear]);
        count++;
        if(getter>0){
            getter--;
            notify();
        }
    }
}

Listing 1.2: Example of synchronization in C using OpenMP

#include <stdio.h>
#include <stdlib.h>
int globalCount = 0;
int addCount () {
    //pragma omp critical
    {
        globalCount = globalCount + 1;
        /* same as globalCount++ */
    }
    return 0;
}
provided in Java, such as javar [AJB97], which uses a pre-processor for restructuring programs into multithreaded programs by using annotations similar to OpenMP compiler directives.

1.2.3 Separation of concerns in concurrent programs

There are also other examples in which a concurrent code suffers from code tangling and code scattering. For instance, many linear algebraic systems of equations arise from scientific problems and need to be resolved. In most cases, these systems involve large matrices that contain many zero-valued coefficients (i.e. sparse matrices). Specific algorithms have been developed to deal with sparse matrices. For linear-algebra applications, two or three concerns can be identified: the means of resolution of the system (e.g. Gaussian pivot), the sparsity of the matrix, and, possibly, the parallelisation of the algorithm. The resulting piece of software, if written using a procedural or an object-oriented language such as Fortran or C++, will have interlaced statements for these three concerns. Although this software can be efficient, such a design does not favour adaptation to another type of sparsity (for example, from upper triangular to banded) or to another type of parallelisation (for example, suitable for another architecture).

Separation of concerns as a design and programming principle is not new. It has been recognised as an important principle for handling software complexity [Dij97] since the 1970s. This principle states that a given problem involves varying types of concerns. These should be identified and separated to cope with complexity and to achieve the required engineering quality factors such as robustness, adaptability, and reusability [Dij97]. The entire concept of parallel programming hinges on the design, development, and deployment of threads within an application and on the coordination between threads and their respective operations. The general aim of the work described in this thesis is to provide computer-literate scientists with a means of enforcing a better separation of the concerns related to thread and concurrency management from computation, and to model different behavioural dimensions with different types of encapsulations, by providing adequate abstraction mechanisms for implementing concerns. The context in which this is attempted is that of the object-oriented Java programming language.
CHAPTER 1. INTRODUCTION

1.3 Java-based concurrent computation

“Java” consists of two parts: the Java language [GJSB05] and the JVM [LY99]. In general, programs written in the Java language are executed on the JVM, which is mainly used for running written programs. Although their respective designs have influenced each other, they do not necessarily need one another. On the one hand, the Java code can be compiled into a native code for a particular architecture, for example with the GNU Compiler for Java, GCJ [gcj]. On the other hand, the Java byte-code can be generated from other languages and can be run on the JVM [Tol08, GC00]. One of these languages is AspectJ (an aspect-oriented extension to Java). Java technology, including both language and virtual machine, has become increasingly popular over the past ten years. The JVM and its standard API\(^1\) have made Java applications highly portable. Moreover, the programming language is relatively easy to use, especially since it implicitly relies on the virtual machine for memory management and security. These advantages have made Java the technology of choice for a large number of applications across a wide range of domains. Several standard features such as portability, networking API, and built-in multithreading have made Java particularly attractive for scientific programming. Its suitability as both a programming language and a running environment for concurrent computation has been questioned several times [CCFL98, Art00, MMG\(^+\)00, BMPP01, Thi02]. This has become more pertinent since the appearance of virtual machines with just-in-time (JIT) compilers, such as HotSpot [Inc97], which have increased performance to an “acceptable” level.

The Java Grande Forum (JGF) [FSS99, CRP01, MFG02] “[aims to] develop community consensus and recommendations for either changes to Java or establishment of standards (frameworks) for Grande libraries and services” [JGF]. A Grande application “is any application, scientific or industrial, that requires a large number of computing resources” [Pan98]. The JGF has identified five critical issues related to the use of Java for large computational problems:

- Multidimensional arrays are currently represented as arrays of arrays;
- Complex arithmetic is not supported with a primitive type;
- Lack of lightweight classes;

\(^1\)The Java development kit (JDK) and runtime environment (JRE) provide the standard Java API.
• Using floating-point hardware is at odds with Java’s portability;

• Operator overloading is not possible.

These issues have been discussed [MMG+00, BMPP01], and some solutions have been proposed. Since Java 1.2, floating-point operations can take advantage of the underlying hardware. Evolution toward a Java environment for high-performance computing is encouraging in terms of both the language and specifications and the JVM implementations. On the one hand, although the Java Community Process has been set up to address the evolution of the specifications of Java (for both the language and the virtual machine), via Java Specification Reviews (JSRs), each review often takes several months. On the other hand, there has been a clear improvement in performance each time a new JVM has been released, particularly on PCs [BMPP01].

However imperfect Java is for high-performance computing, this is the platform of choice for most of the supported aspect-oriented tools. Therefore, for practical reasons, the experiments, the tools, and the methods developed throughout this thesis are Java-based. The performance results presented in the thesis depend heavily on the JVM environment within which the test cases have been executed.

1.4 Challenges and contributions

As a continuation of research for a Master project [Xi06], which was a project with only a simple implementation for synchronized block join points (with only before and after advice), this thesis not only fully implements the function of around advice and rm_proceed() as an extension to the original synchronization of join points. It also focuses on establishing why the refactoring of concurrent programs may benefit from AOP and gathers a wide range of examples and applications to analyse and support the use of these fine-grained join points.

1.4.1 Aspects for concurrent computation

The main challenge addressed throughout this thesis is the decoupling of computational models from their parallelisation expression. The first main contribution of the thesis is to provide methods that enable aspects to achieve this goal, as
presented in Chapters 3 and 4. More practically, aspects that implement diverse concurrent computing strategies are described in Chapter 5. The flexibility induced by these aspects makes the concurrent feature, a pluggable unit, reusable across applications.

Since AspectJ cannot intercept basic parallel mechanisms, such as synchronization, that are at the heart of multithread computing, this thesis proposes various object-oriented models for describing synchronization strategies in a multithreaded environment that allow AspectJ to be used for encapsulating parallelisation in an aspect. Moreover, multithread programs are always difficult to test and debug owing to the undeterminable execution order of each thread. A suitable aspect for synchronization is also a natural place to trace a running multithread program.

This thesis highlights practical issues that arise when using AOP to solve the problem of code tangling in concurrent programs. Since AspectJ is probably the most mature aspect-oriented tool, an attempt is made to use it for separating model and concurrent concerns in Java scientific code. In order to evaluate the separation of concerns of Java code in concurrent applications, this thesis uses a set of aspect-oriented metrics that are widely applied in aspect-oriented software development [CK91, KKG07, CSF+06, FSG+08, FCS+08] for evaluating separation of concerns.

1.4.2 AspectJ and beyond: join points for complex behaviour

The main problem encountered when applying aspect-orientation to scientific software is identification of the adequate abstractions for the domain. Filman et al. define the abstractness constraint in AOP: “The constructs of an aspect-oriented programming language must be abstract enough to match the natural abstractions of the problem domain. However, they also must be concrete enough to match the realisation of the implementation platform. This constraint aims to minimize the implementation effort and enable efficiency” [FECA04].

The abstractions AspectJ can match and at which it can intervene are its join points. For example, in the domain of scientific computing, one natural abstraction is synchronization. However, current AspectJ join points are limited to
relatively simple Java behaviour and, in particular, do not include synchronization. The second main contribution of this thesis is to provide AspectJ with a set of join points that correspond to complex behaviour in the AspectJ model: the join point for synchronizations, the join point for if-then-else blocks, and the join points for general blocks.

Chapter 4 provides AspectJ with the new set of join points that enables aspects to intervene directly at concurrent computing constructor levels. A secondary contribution of the general block and if-then-else block join points are developed to support more flexible usage to separate concurrent concerns from base code. These fine-grained join points can help separating and modularising synchronization, parallelisation, and exception handling. A code base selected by these join points might be appropriately extended to be consistent with the block hierarchy of the program.

1.5 Outline

Chapter 2 introduces aspect-oriented programming, which is the approach used to address the problem, and presents the background work. After this, Chapter 3 presents traditional techniques for parallelising Java applications in concurrent programs. Chapter 4 goes a step further and provides AspectJ with a set of join points for parallel concerns. An extended synchronization join point, if-then-else join point, and general block join point are introduced. Application examples and an evaluation of the two approaches are presented in Chapter 5. Finally, Chapter 6 concludes.
Chapter 2

Aspect Oriented Programming

This chapter provides the motivation for, and main concepts of, Aspect-Oriented Programming. Section 2.1 presents the motivation for this new technology by explaining crosscutting concerns and their related programming paradigm. Section 2.2 presents the main concepts behind AOP and their novelty, explaining the basic terms and definitions which form the key vocabulary used in this thesis, e.g. concerns, separation of concerns and Aspect-Oriented Programming. Section 2.3 and Section 2.4 give an overview of current AOP languages and tools. Section 2.5 presents some related work regarding this thesis.

2.1 Motivation

The design and evolution of programming languages is one of the most important areas of computer science. Programming languages define the manner in which programmers communicate with their machines. These languages give programmers layers of abstraction with which to work, so that they can accomplish tasks without reaching into the hardware. The languages also attempt to increase efficiency by automating many hardware-bound tasks. Anyone who has attempted to write a large project in an assembly language understands the necessity for higher level languages.

Over the last 50 or so years, languages have continued to evolve in order to support their ever-increasing usage. Program design and maintenance became issues with the dawn of software engineering, when people sought to formalise methods of constructing correct, efficient and easily modified programs. Languages evolved in order to support these new requirements. Block structure was
born out of a desire for modularity. Object-orientation (OO) was created from a desire to have language constructs to model real-world objects and encourage the reuse of software. OO has been around for more than two decades now, and has become the default paradigm in many peoples’ minds.

However, object-orientation still has its problems. While it encourages the reuse of software, practical experience has shown that OO does not handle this as effectively as originally expected. Pre-packaged software often does not suit the programmer’s needs, and ill-constructed interfaces make it difficult to use these packages. Furthermore, OO should enhance maintainability by causing redesign to affect as few modules/classes as possible. However, as programs continue to become larger and larger, it is increasingly difficult to cleanly separate concerns into modules.

Aspect-Oriented Programming (AOP) may be the next step in the steady evolution of the OO paradigm, or possibly it will evolve into a completely new paradigm independent of OO. AOP offers a solution to a design and maintenance problem which has plagued software developers for years: how to create modules with few or no crosscutting concerns. AOP introduces the notion of aspects, and provides a means for taking crosscutting concerns out of modules and placing them in a centralised space.

Typically, crosscutting concerns are scattered or tangled as code, making them harder to understand and maintain. They are usually scattered by virtue of a function (e.g. logging) being spread over a number of unrelated functions, which may use it, possibly in entirely unrelated systems, in different source languages, etc. This means that changing logging can require the modification of all affected modules. Crosscutting concerns are not only tangled with the mainline function of the systems in which they are expressed, but also with each other. This means that changing one concern entails understanding all the tangled concerns, or having some means by which the effect of changes can be inferred. For example, consider a banking application with a conceptually simple method for transferring an amount from one account to another, as shown in Listing 2.1.

This transfer method overlooks certain considerations, which would be necessary for a deployed application. For example, it requires security checks to verify that the current user is authorised to perform this operation; the operation should be in a database transaction in order to prevent accidental data loss. For diagnostics, the operation should be logged to the system log, and so on. Having
Listing 2.1: Example of aspects in a banking application.

```java
void transfer(Account fromAccount, Account toAccount, int amount) {
    /**
     * This is a transfer function without security check
     */
    if (fromAccount.getBalance() < amount) {
        throw new InsufficientFundsException();
    }
    fromAccount.withdraw(amount);
    toAccount.deposit(amount);
}
```

addressed these new concerns, a simplified version would look somewhat like the code shown in the Listing 2.2.

In this code, other interests have become tangled with the basic functionality (sometimes called the business logic concern). Transactions, security, and logging all exemplify crosscutting concerns.

One should also consider what happens if the code suddenly needs to be changed, for example to update the security considerations for the application. In the program’s current version, security-related operations would appear to be scattered across numerous methods, and such a change would require a major effort. Therefore, such crosscutting concerns are not appropriately encapsulated in their own modules. This increases the complexity of the system and makes the evolution of the code considerably more difficult, because the concern for security would crosscut the model, and would not be encapsulated in its own entity.

AOP attempts to resolve this problem by allowing the programmer to express these kinds of crosscutting concerns in stand-alone modules. These modules are called Aspects, which can contain advice and inter-type declarations, therefore making it easier to read, re-use and adapt the code in other contexts. For example, a security module can include advice which performs a security check before accessing a bank account. This way, both the check and the places at which it should be made can be maintained in one location in the code. Further, a well-designed module can anticipate later program changes so that, if another developer creates a new method to access the bank account, the advice will be correctly applied to the new method when it is executed. The principles of such an implementation are described in the next section.
Listing 2.2: Example of aspects in a banking application.

```java
void transfer(Account fromAccount, Account toAccount,
              int amount) throws Exception {
    if (!currentUser().canPerform(OP_TRANSFER))
        throw new SecurityException();

    if (amount < 0)
        throw new NegativeTransferException();

    Transaction tx = database.newTransaction();

    /**
     * This is a "wrapped" transfer function with security check,
     * transaction execution in database and logging.
     */
    try {
        if (fromAccount.getBalance() < amount)
            throw new InsufficientFundsException();

        fromAccount.withdraw(amount);
        toAccount.deposit(amount);

        tx.commit();
        systemLog.logOperation(OP_TRANSFER, fromAccount,
                                toAccount, amount);
    }
    catch (Exception e) {
        tx.rollback();
        throw e;
    }
}
```
2.2 Concepts

Aspect-Oriented Programming aims to link concerns which cut across each other, and encapsulate them transparently as separate program entities. This leads to the definition of the term ‘software concern’.

Concern is found in many publications about AOP, even though the term itself appears to be hard to define. Filman et al. define a concern as ‘a thing in an engineering process about which it cares’ [FF02]. Another definition of concern is ‘an interest which pertains to the system’s development, its operation or any other matters that are critical or otherwise important to one or more stakeholders’ [vCC05]. In this thesis, a concern is considered to be a concept, functionality or any kind of requirement, which is implemented by a software system.

A crosscutting concern is a concern whose implementation is scattered throughout the implementation of other concerns in a software system [vCC05]. In particular, concerns which cannot be modularised within a certain programming model are considered to be crosscutting, because the elements of their implementations are scattered and tangled within the elements of other concerns.

The term ‘scattering’ denotes the occurrence of elements which belong to the implementation of one concern in modules encapsulating other concerns, whereas tangling characterises the occurrence of multiple concerns mixed together in one module. The term ‘crosscutting’ is used to characterise scattered or tangled elements in the implementation of a concern.

The general style of programming, which arises from this aim, consists of program statements in the following form:

“In programs P, whenever condition C arises, perform action A.” [FF02]

This generates the following definitions: a join point is a point in the structure or in the execution of a program, where a concern which crosses that part of the program may intervene. According to the above formulation, join points are points which can be used to express potential conditions C in programs. Join points can be seen as hooks in a program where other program parts (specifically, aspects) can be conditionally attached and executed.

A pointcut is a subset of all possible join points. The expression of a pointcut is the pointcut descriptor (often, the term ‘pointcut’ is used in place of ‘pointcut descriptor’). A pointcut descriptor defines the condition C in the above formulation. This condition matches a subset of join points, which is the pointcut. The piece of code A which is to be executed when condition C arises (i.e. at a join
point of the pointcut) is called the *advice*.

The unit of code which defines the pointcuts and the advice related to the same concern is called an *aspect*. An aspect can also be more generally defined as a unit that encapsulates a crosscutting concern. In many AOP languages, an advice is a method-like construct, which can be invoked at a join point. It can declare input parameters for accessing information, which is available at a join point. An advice contains a block of statements that are executed either before or after the join point, or which replace it completely.

The counterparts of aspects are components known as base codes, which are the functional units of code that do not contain aspect-orientated statements, but only base actions. Components are units of code that are written using functional, procedural or object-oriented languages. To some extent, the advice in aspects could be considered as a component, within which aspects could intervene. However, the problems which can arise out of interaction between aspects lie beyond this introduction, and must be the subject of continuing research.

Mixing components and aspects together, so that the behaviour specified by the aspects occurs where and when it is supposed to, is a process of *weaving*. Some implementations use a specific type of compiler, called a *weaver*, to generate an executable form of component and aspect. Other implementations perform the weaving at runtime or load-time, using mechanisms equivalent to a runtime or load-time form of compilation. Although the details of these implementations are outside the scope of this introduction, some examples of aspect-oriented languages and frameworks are given in the next section.

AOP is not about writing macros or inserting codes at some given line number. Rather, it is about applying certain actions when a specifiable behaviour occurs. Thus, mechanisms for aspect orientation rest on the following three pillars:

- a model of the behaviour which can be recognised and exploited (the join points),
- a means of characterising a subset of these possible behaviours (the ability to define pointcuts), and
- a means of implementing the behaviour defined in the aspects at the place, and at the time, the expected behaviour defined in the pointcuts occurs (the weaving of the advice) [KHH+01].
CHAPTER 2. ASPECT ORIENTED PROGRAMMING

The components need not be aware of the effect of the aspects to which they are subjected. Specifically, components may be prepared to be subjected to aspects, for example via annotations or refactorings, but should not be prepared in a manner which would couple them tightly with the behaviour of any potential aspect to which they may be subjected. This notion of obliviousness is one of the main assets of AOP in improving the flexibility of software development. This means that, in some cases, the integration of certain aspects into a final version of the code is optional. However, failure to integrate some aspects may completely change the behaviour of the application concerned. For example, an aspect which would check the consistency of some data may be necessary to prevent faults, whereas an aspect which would be used by the programmer for debugging rarely needs to be integrated into the final version of a project.

Reasoning about aspects is still an open problem. Filman and Friedman propose that,

“better AOP systems [should be] more oblivious. They minimise the degree to which programmers (particularly the programmers of the primary functionality) have to change their behaviour to realise the benefits of AOP” [FF02].

However, full obliviousness has proven to be difficult to achieve in practice. Decoupling crosscutting concerns from the base system provides benefits in term of readability, but full obliviousness can prevent the programmer of the advised units from knowing what will happen when these units are utilised and, in most cases, will not be free of undesired side-effects. Even in mainstream AOP languages such as AspectJ (see Section 2.1.3), tools have been developed to assist programmers to understand the interactions between the aspects and the components.

More recently, Kiczales and Mezini [KM05] have proposed a different way of reasoning about aspects, in which:

“aspects cut new interfaces through the primary decomposition of a system. This implies that in the presence of aspects, the complete interface of a module can only be determined once the complete configuration of modules in the system is known. While this may seem anti-modular, it is an inherent property of crosscutting concerns [...].”

Aspects change the concepts of modules as they are used in procedural and object-orientated languages, though they provide the ability to view and to reason about cross-sections of the system. Related work on aspects and modularity
CHAPTER 2. ASPECT ORIENTED PROGRAMMING

includes [Cha05], [Ald05] and [SGS+05].

2.3 AspectJ Language

Several aspect-oriented languages and frameworks have been developed. Aspects must use a language to describe the actions to be performed according to their advice. Obviously, the components into which the aspects are woven are also written using a language. The language used for both the components and the advice is usually the same, and this is normally a general purpose language such as C or Java. Most of the aspect-oriented languages and tools are based on procedural, functional or object-oriented languages which exist outside AOP. A list of aspect-oriented languages and tools can be found on the Aspect-Oriented Software Development (AOSD) website [ws].

One of the most popular and mature aspect-oriented languages is AspectJ [Asp, KHH+01]. The AspectJ project started at the Xerox Palo Alto Research Center, where leading researchers published some of the founding articles on Aspect-Oriented Programming [KLM+97]. AspectJ is an aspect-oriented extension to Java. It uses regular Java statements to write the advice, but also defines a few specific constructs for encapsulating aspects and for writing pointcuts. A summary of the syntax can be found in Appendix A. AspectJ uses a specific compiler (a weaver), which produces a standard bytecode which can be executed on any Java virtual machine.

AspectJ mainly works on the interfaces of the classes. The basic join points AspectJ can use are calls to methods, executions of methods, accesses to fields, instantiations of objects and executions of exception handlers. Pointcut descriptors are then written as logical expressions, which define which of these join points are to be picked out. The selection is based on the name of the objects and the methods involved, and on their signature, using regular expressions. It is also possible to refine the conditions, for example by selecting certain calls only if they are called from within the control flow of a particular method. The advice can be executed before, after, or around these pointcuts.

Aspects in AspectJ can be compared to classes in Java. AspectJ defines the aspect keyword for declaring aspects. These aspects can contain pointcut descriptors, advice, and even regular Java fields and methods (which can be used by the advice). Two examples of AspectJ’s syntax are shown in Listings 2.3 and
2.4. The full reference and programming guide on AspectJ can be found on the official AspectJ web-site [Asp].

Listing 2.4 shows an example of around advice. This kind of advice will be used for parallelisation in the following chapters. An around advice replaces the execution of the intercepted join point. One of the principal AspectJ constructs for around advice is proceed. This keyword can only be used in around advice. As its name indicates, it proceeds with the execution of the intercepted join point. The main feature of proceed is that it creates arguments which must match the parameters of the advice, although these arguments can be modified before going ahead with the execution of proceed. In this example, proceed(v) will execute a join point which consists of setting Test.value. However, proceed replaces the values of the original pointcut arguments with its own arguments. In this example, if v > AUTHORISED_MAX, the value AUTHORISED_MAX is going to be passed to proceed, and the execution of the join point which corresponds to setting the value of Test.value is going to use AUTHORISED_MAX for the new value, instead of the original value of v. Moreover, the use of proceed is optional; not using it implies that the join point advised is not going to be executed.

Along with compilers, such as that for AspectJ, a set of tools exists to assist developers and designers to use aspect-oriented technologies. Eclipse is a development environment which fully supports AspectJ via the AspectJ Development Tool (AJDT) [CCHW04], which is an optional plug-in. It can assist the development of AspectJ projects with various features, such as graphically illustrating where the aspects are woven in. Since development environments are often a matter of taste, similar AspectJ plug-ins exist for Emacs, JBuilder and Netbeans. Other tools, such as 'Aspect Browser', can help to find crosscutting concerns in existing Java projects by producing representations.

2.3.1 Other Languages

There are several other languages that provide a framework or set of criteria of presentation that satisfy the synchronization join point presentation. As AspectJ’s pointcut-advice model is the most popular model. It offers a better granularity than other approaches and considers more general kinds of concerns than the adaptive programming.

Furthermore, the pointcut-advice model adapts extensively the pull approach. It allows tracking sub points in the control flow of the application, for example
Listing 2.3: Example of an aspect, using AspectJ (before and after advice).

```java
/**
 * This simple class only contains a field.
 * It can be used in various places in a larger programme.
 */
public class Test {
    public int value = 0;
}
/**
 * This short aspect prints out "The value is going to
 * be modified." each time the "value" field of any
 * instance of "Test" is about to be assigned.
 */
aspect SimpleTracing {
    /**
     * This pointcut picks out all the points where the
     * "value" field of any instance of "Test" is
     * modified.
     */
    pointcut modifyingValue(): set(int Test.value);
    /**
     * This piece of advice is executed before each join
     * point picked out by the pointcut defined above.
     * The body of the piece of advice is written as the
     * body of a Java method would be.
     */
    before(): modifyingValue() {
        System.err.println("The value is going to be modified.");
    }
}
/**
 * This aspect is similar to the aspect above, but it
 * gets the context and prints out the new value as well.
 */
aspect TracingWithContext {
    pointcut modifyingValue(): set(int Test.value);
    /**
     * This piece of advice also gets the argument to
     * the setting of the new value. This element of
     * dynamic context can be used from within the piece
     * of advice.
     */
    after(int v): modifyingValue() && args(v) {
        System.err.println("The new value is: " + v);
    }
}
```
Listing 2.4: Example of an aspect, using AspectJ (around advice).

```java
/**
 * This simple class only contains a field.
 * It can be used in various places in a larger programme.
 */
public class Test {
    public int value = 0;
}

/**
 * This aspect enforces a maximum value for the "value"
 * field of any instance of "Test".
 */
aspect SaturateValue {
    pointcut modifyingValue(): set(int Test.value);

    /* This piece of advice is executed instead of the
     * action of setting the "value" field of any
     * instance of "Test". "proceed" executes the
     * intercepted join point, but replaces the value of
     * the argument to "set".
     */
    void around(int v): modifyingValue() && args(v) {
        if (v > AUTHORISED_MAX)
            proceed(AUTHORISED_MAX);
        else
            proceed(v);
    }
}
```
points where methods are invoked and fields are set. Hence, AspectJ is chosen as the candidate to implement separation of concurrent concerns in Java applications.

**Hyper/J**

Hyper/J [TO05] is the Java implementation of an approach to Multi-Dimensional Separation of Concerns (MDSoC), known as hyperspaces. A hyperspace is a set of units. A unit can be either primitive, such as a field, method, and constructor, or compound, such as a class, interface, and package. A hyperslice is a modularization mechanism that groups all the units that implement a concern, which consists of a set of classes and interfaces. Hyperslices must be declaratively complete. They must have a declaration, that can be incomplete (stub) or abstract, for any unit they reference. Hyperslices are integrated in hypermodules to build larger hyperslices or even complete systems.

The Hyper/J weaver performs composition at the bytecode level, which makes a natural decision to implement each hyperslice (concern module) as a package that can be compiled independently. However, an extra code is required to make a hyperslice declaratively complete so that it can be compiled. Additionally, the Hyper/J weaver requires stubs for non-default constructors. When the package is compiled, the references of these variables are bound to the definitions in the package. The extension of methods and constructors is realised by appending the code of their bodies one after the other.

**AHEAD**

AHEAD (Algebraic Hierarchical Equations for Application Design) is a feature modularization and composition technology based on step-wise development [BSR04]. It was created to address the issues of feature-based development of product-lines. AHEAD partitions feature into two categories: constants that modularize any number of classes and interfaces, and functions that modularize classes, interfaces and their extensions.

**2.4 Tools:** *abc*

The design goal of join point to separation of concurrent concern is to implement a language extension to modify an existing compiler. However, this is not always
the best approach, since existing compilers may not have been designed with extensibility as one of the main goals. Furthermore, they may be constrained to work with infrastructures which, themselves, are not easily extensible. In the case of AspectJ, the older pre-existing implementation is *ajc*, which is designed to support fast and incremental compilation and also to interact closely with the Eclipse toolset. A new kind of compiler implementation is needed for the design implement join point, which is *abc* [ACH+05], the Aspect-Bench Compiler, with extensibility as its primary design goal. *abc* also aims for an optimizing implementation of AspectJ.

In *abc*, to meet the extensibility approach, it uses existing, proven tools, namely the *Polyglot* extensible compiler framework to build on the frontend, and the *Soot* [VRCG+99] analysis and transformation framework to build on the backend. Indeed, Polyglot has been shown to meet the criteria of simplicity, modularity and proportionality on a wide variety of extensions to the syntax and type system of Java. By the same token, Soot has been shown to meet all the above criteria for code generation, analysis and optimization. Given the success of these building blocks, it is extremely important to design *abc* so that both are used as is, without any changes that are specific to *abc*. The section below dictates an architecture where the frontend separates the AspectJ program into a pure Java part and a part containing instructions for the weaver.

Soot is a Java bytecode analysis toolkit based around the *Jimple* [VRH98]. Jimple is low-level enough for pointcut matching, in that the granularity of any join point is at least one entire Jimple statement. It is high-level enough for weaving and easy analysis; in particular, during weaving there is no need to worry about implicit operations on the computation stack, because all operations are expressed in terms of explicit variables. Soot can produce Jimple from both bytecode and Java source code.

The framework of Soot for analysing and transforming Java bytecode is used as the back-end of *abc*. The most important advantage of using Soot as the backend is its intermediate representation, Jimple, both for developing *abc* itself and for extending the language. Soot provides modules to convert between Jimple, Java bytecode, and Java source code. It furthermore includes implementations of standard compiler optimizations, which *abc* applies after weaving. Significant speedups from these optimizations have already been observed alone. In addition to already implemented analyses and transformations, Soot has tools for writing
new ones, such as control flow graph builders, definition/use chains, a fixed-point flow analysis framework, and a method inliner. To create new pointcuts, these features are useful for implementing extensions that need to be aware of the inter-procedural behaviour of the program.

Because abc works with an unmodified Soot and Polyglot, it is easy for abc to update to the latest versions of Soot and Polyglot, which are released as the developers of abc itself. By the same token, it can also upgrade to new versions of AspectJ extensions abc without difficulty. This independence was achieved mainly by separating the AspectJ-specific features in the code being processed from standard Java code.

2.5 Related Work

There are several AOP systems in which a pointcut can specify a relationship between join points. AspectJ’s cflow [DT04] captures join points based on a control flow, which is useful when implementing access control policies. dflow [MK03] identifies joint points based on data-flow information, which is useful to enforce secrecy and integrity policies. However, both capture control data flow that has already occurred, rather than that which will happen in the future as in the general block join point, which will introduced in Chapter 4 in this thesis. Tracematches [DFS04], which are trace-based aspects, track join point relationships based on execution traces, which also have difficulty in implementing a proper around advice for the target code base.

Just before the completion of this thesis, transactional pointcut [SMH09] is a proposed approach for identifying temporal relationships between joint points, which describe related events by patterns comprised of several pointcuts. Transcut implemented in [SMH09] tracks relationships by selecting a region in the control flow graph of a program. It addresses the similar problem and provides an alternative solution for general block join point.

Also, java.util.concurrent [JGF] package provides standardized, efficient versions of utility classes commonly encountered in concurrent Java programming. But as this thesis mainly focuses on the refactoring of a concurrent program, this package is outside the scope of this thesis.
CHAPTER 2. ASPECT ORIENTED PROGRAMMING

2.6 Summary

This chapter has described the need to deal with crosscutting concerns using AOP. It has introduced the motivation for, and main concepts of, Aspect-Oriented Programming. It has also presented AspectJ, an aspect-oriented extension of Java, illustrated by a short example. Further information about AOP and related tools can be found on the AOSD website [ws] and in a special issue of the Communications of the ACM dated October 2001 [EFB01, EAK+01, LOO01, OT05, BA03, KHH+01, PC02, EZS+08, CSTW08, Net08, WBG+06, Sul01]. During the course of this project, several books or book chapters on AOSD have been published. Some are specific to AspectJ [CCHW04, Lad03], while others are more general, such as [FECA04, Mon05]. The ability to characterise and recognise certain behaviours lies at the root of AOP. Aspect-oriented implementations can be successful if the join points, at which the aspects intervene, can be clearly expressed and characterised. This characterisation is in terms of the behaviour and abstraction of the components, rather than by means of explicit mark-up annotations, which are tightly coupled to the implementation.

Explicit mark-up annotations or compiler directives are more a means of expressing complex concerns in compact abstractions rather than a means to separate concerns. Annotations neither entail the automatic recognition of a certain behaviour nor possess the flexibility of an aspect-oriented programme.

Therefore, this chapter has not only explained the basics of aspect-oriented programming, being the main technical background for the following chapters, but it has also provided further motivation as to why the techniques implemented in Chapters 3, 4 and 5 are worth considering, and how research into the separation of concerns can guide the development of new, more suitable techniques.

The loop join point presented in the next chapter comes from articles published in 2005 [HG06]. Since then, little work has been done on parallel optimization in the aspect-oriented software community. Concurrent computing and multi-threaded programs have been mentioned in several publications as an example of a type of aspect. However, most of the more recent work on concurrent computation consists of improving parallelisation at a coarser level, and has not been applied in any practical large-scale concurrent systems, for example by writing aspects to handle the caching of network transactions, or the caching of successive requests to complex methods [DHS+03]. The next chapter will review the coarse-grained separation of concerns.
Chapter 3

Concurrent Concerns

It is important that AOP languages are expressive enough to accomplish the goal of separating crosscutting concerns. The use of AOP should lead to better software because the localisation of crosscutting concerns makes programs easier to read, maintain and evolve. However, the original AOP languages often introduce tight coupling between the aspect and the points they crosscut. Such tight coupling harms the ability of an aspect-oriented program to evolve and should thus be avoided.

In the earliest AOP language, AspectJ, crosscuts were formed by explicitly enumerating join points by name. This clearly introduced tight coupling between the aspect and the modules it crosscut, and changing the program required a review of the crosscut enumerations, which conflicts with the idea of programs being oblivious to the aspects applied to them. A concurrent concern is a kind of pattern-based crosscutting, which can be found in most concurrent programs, which attempts to resolve the problems with parallel computing. This concern need to be identified by expressing crosscuts by stating the properties of join points instead of selecting them by name. The work presented in this chapter has led to the work for an extension of loop join points [HG04, HG06] and also led to the detailed investigation of synchronized block join point [Xi06]. [Xi06] is a project with solely implementation purpose for synchronized block join point, which is lack of examples and applications to analyze and support the usage of the join points.

Firstly, the schemes for using AspectJ to encapsulate parallel concerns are reviewed, while Section 3.1 reviews the implementation for the loop join point model [HG04, HG06] as well as its extension and how these join points work for
part of the separating concurrent concerns.

Secondly, section 3.2 provides a prototype for the aspect-oriented model for synchronization and also for the synchronized block join point [Xi06] for separating synchronization concerns in AspectJ.

Those two sections discuss the problem of separating concerns in concurrent programs using various general mechanisms in traditional OO languages. Each mechanism is given a sub-section in which its parallel features, paradigm and other characteristics will be discussed. These sections take a look at a number of concurrent concerns which may be beneficial in expressing better parallel crosscuts, and investigates a wide range of concurrent programs and related research which motivates the work described in this thesis. A driving example is used to show the problems associated with the separation of concerns in concurrent programs.

Finally, section 3.3 briefly introduces ideas for basic block join point and gives an idea of how the availability to target the basic block constructs could help to separate parallel concerns in the concurrent program.

### 3.1 Aspect-oriented model for loops

Concurrent programs often focus on two concerns: the algorithm used by the calculation itself related to some underlying scientific models, and a well-arranged parallelisation mechanism to achieve concurrency. These two separate concerns, in current procedural or object-oriented implementations, result in elements of code which usually interact within the same unit of the system design. This section reviews initial attempts to separate the parallel concerns from the base code in an object-oriented way and their resulting limitations.

Section 3.1.1 gives a quick introduction to loops functionality in Java. Section 3.1.2 presents an object-oriented model for describing loops and its shortcomings in terms of separating the associated parallel concerns. Section 3.1.3 presents the extended loops join point which is based on the loops join point [HG06]

#### 3.1.1 The loops model in Java

A loop is a sequence of statements which is specified once, but may be executed several times in succession. The code 'inside' the loop is obeyed a specified number of times, or once for each of a collection of items, or until some condition is met.
In some languages, such as Scheme, loops are expressed using a tail recursion rather than explicit looping constructs. Three typical types of loop in Java, as shown in Listing 3.1, are while loops, for loops and foreach loops.

A ‘while loop’ is a control flow statement which allows a code to be executed repeatedly based on a given boolean condition. The while loop can be thought of as a repeating if statement. The while loop construct consists of a block of code and a condition. The condition is evaluated, and if it is true, the code within the block is executed. This is repeated until the condition becomes false. Because the loops check the condition before the block is executed, the control structure is also known as a pre-test loop. This can be compared with the do-while loop, which tests the condition after the loop has been executed.

A ‘for loop’ is a statement which allows a code to be repeatedly executed. A for loop is classified as an iteration statement. Unlike other kinds of loop, such as the while loop, the ‘for’ loop is often distinguished by an explicit loop counter or loop variable. This allows the body of the ‘for’ loop (the code which is being repeatedly executed) to know about the sequencing of each iteration. For loops are typically used when the number of iterations is known before entering the loop.

Apart from the above, Java (since Java 5) also supports a ‘foreach’ loop. A ‘foreach’ loop is a computer language construct for traversing items in a collection. ‘foreach’ is usually used in place of a standard ‘for’ statement. However, unlike other for loop constructs, ‘foreach’ loops maintain no explicit counter. This can potentially avoid off-by-one errors and make a code simpler to read.

Listing 3.1: Example of three loops in Java.

```java
/* While loop and its condition */
int counter = 10;
while (counter > 1)
    //statements;
/* For loop and its counter */
for (int counter = 1; counter <= 5; counter++)
    //statements;
/* For each loop and its item set */
foreach (type item: set)
    // do something to item
```
3.1.2 Example to separate a loop concerns in OO

The following examples are taken from [HG06] and focus on rectangular double-nested loops, with fixed bounds, which would usually be written in the form shown in Listing 3.2. It is assumed that the different instances of the body of the loop can be executed in no particular order, and thus, executing the loop body with certain values of \(i\) and \(j\) does not impact upon the execution of the same loop body with other values of \(i\) and \(j\).

Listing 3.2: Rectangular double loop nest in Java.

```java
for (int i = minI; i <= maxI; i++) {
    for (int j = minJ; j <= maxJ; j++) {
        /* Loop body. Functions of i and j. */
    }
}
```

The separating model, called LoopA in Listing 3.3, consists of a delegation relationship between the following:

- the Runnable2DLoopBody interface to represent a double-nested loop body;
- the LoopA class which is in charge of executing the iterations.

In this model, the body of the loop appears in the `void run(int i, int j)` method of the class which implements the Runnable2DLoopBody interface. The \(i\) and \(j\) parameters of the method are the loop indices of the associated double-nested loop. An instance of LoopA holds an instance of Runnable2DLoopBody (its `loopBody` attribute) and the bounds (\(minI, maxI, minJ\) and \(maxJ\)) of the rectangle in the iteration space for which it is responsible. The `run()` method of LoopA is a double-nested loop which executes the `run(int i, int j)` method in its `loopBody` attribute over the part of the iteration space defined by the given bounds. As an example of this scheme, the loop described in Listing 3.2 would be refactored as shown in Listing 3.3.

The creation of an instance of LoopA, or a call to its `run()` method, together with the dynamic context (in this case, the arguments to the constructor), can be intercepted by an aspect in AspectJ. As a result, parallelism can be implemented transparently in the nested loop structure.

The LoopA implementation makes one call per iteration to a method belonging to an external object (`loopBody.run(i, j)`). The LoopA structure enforces a
better separation of concerns by using two classes to describe the boundaries and
the loop body.

The above model has been designed so that the iteration space and the loop
body of an embarrassingly parallelisable, double-nested for-loop can be encapsu-
lated into an object. The creation and manipulation of such objects can then be
recognised in AspectJ aspects. However, although \textit{LoopA} is expected to provide
better timing results, the programmer has to write the inner loop explicitly each
time. Thus, AspectJ can be used to define a tile of the iteration space originally
defined in a sequential implementation, and parallelise the loop accordingly.

\section*{3.1.3 An extended loops join point}

When parallelising code in order to improve performance, loops are the natural
places to make changes. There are sometimes several alternative ways of paral-
lelising the same loop, depending on various parameters, such as the nature of
the data being processed, or the architecture on which the application is going to
be executed as presented in \cite{HG06}. The model of loop join point presented thus
far takes an outside view of the loops; the point \textit{before} and \textit{after} the loop are not
within the loop itself. As a consequence, however many iterations there may be
for a given loop, \textit{before} and \textit{after}-advice will be executed only once. For some
applications, for example for inserting a piece of advice before each iteration, it
might be desirable to advise the loop body.

Section 3.1.3.1 presents an extended loop join point model which is based on
\cite{HG06} and the kind of loops it aims to recognise. Section 3.1.3.2 shows how to
write aspects for parallelisation using the extended loop join point.

3.1.3.1 The loop join point model

The basic loops join point presented in [HG06], which consists of defining: (a) the behaviour the model a loop aims to recognise, and (b) its dynamic (run-time) characteristics while execution, that is, to provide a join point with an execution context. Although this is a general purpose construct, the behaviour it defines is still not enough for parallelisation because this construct encapsulates the entire code which consists the iterative structure and the associated data to be processed. The “all in one” strategy cannot give enough flexibility while separating loop concerns. To make the loop join point more fine grained, an inner-loop join point is provided as an extension to the basic loop join point, which enable the concrete code refactoring inside loop.

Listing 3.4: Example of Java for-loops iterating over a Collection.

```java
/* Basic loop join point */
ExampleClass[] a ;
/* before the loop */
for (int i=0; i< a.length; i++) {
    /* Do something with obj */
}
/* after the loop*/

aspect BasicLoopJoinPoint{
    before(): loop(){...}
    before(int min, int max, int stride): loop()
        && args (min, max, stride) {...}
}

/* Extended loop join points */
ExampleClass[] a ;
for (int i=0; i< a.length; i++) {
    /* before the loop */
    /* Do something with obj */
    /* after the loop */
}
/* after the loop*/

aspect ExtendedLoopJoinPoint{
    before(): loop_body(){...}
    before(int min, int max, int stride): loop_body()
        && args (min, max, stride) {...}
}
```
The difficulties in defining the semantics of the inner loop body join lies in defining the structure of the loop. Even in the source-code, there is ambiguity about where to weave before and after advice in such a case. For example, is the termination condition in the loop-body or not? This question is even more pertinent for complex conditions that may include calls to methods. A basic-block control-flow approach here is to solve the problem. It defines that “before” the inner loop-body is the point at the beginning of the “header” [HG06], included in the loop, and that “after” the loop-body is a point inserted on the back edge of the natural loop. If there were several back edges in the corresponding combined loop, an equivalent of the “pre-header” could be inserted between the back edges and the header, in order to keep a single weaving point. In the case of a while-loop or a for-loop, “before” the loop-body would also be before the evaluation of the condition. Without the enhancement, such a model would not comprise any contextual information (or “arguments” to the loop-body).

As an illustration of the loop body, Listing 3.5 shows an example of code that involves inner loops join point and breaks (taken from [HG06]). Figure 3.1 shows the block-level control-flow graph for this example in which:

- the loop consists of blocks 1, 3, 4, 5, 6 and 7; its exit nodes are blocks 1 and 5; its successor node is block 9; and

- the loop body consists of blocks 3, 4, 5, 6 and 7; its exit nodes are blocks 5 and 7; its successor node is block 8, the injected tailers.

Listing 3.5: Two nested loops with a break statement jumping outside the outer loop

```java
int i = 0;
outerloop:
while (i < maxI) {
    int j = 0;
    while (j < maxJ) {
        if (c(i,j))
            break: outerloop ;
        j++ ;
    }
    i++ ;
}
/* A */
```
CHAPTER 3. CONCURRENT CONCERNS

In such cases, where there are several successor nodes, there are multiple points corresponding to the transition from blocks within the loop body to blocks outside the loop body but still within the loop. Thus weaving an after piece of advice would require replication of the woven code at ‘all’ back edges between exit nodes and their successor nodes.

3.1.3.2 Aspects for parallelisation

This section presents an application of the `loop()` pointcut, namely the parallelisation of loops. For loops iterating over a range of integers, the boundary values are passed via the `args` construct of AspectJ, to which `int` values are bound (for `min`, `max` and `stride`). Additionally, an extra argument will be bound to the originating array, if one has been found.

For loops iterating over an Iterator, the first argument of `args` will be bound to the corresponding instance of `Iterator`. Also, an extra argument will be bound to the originating Collection, if one has been found. In cases where the originating array or Collection do not matter, the use of a double-dot notation (“..”) [KHH+01] is recommended to make the argument optional. The following illustrates the usage of the loops pointcut:

- `loop.body() && args(min, max, stride)` will match only loop bodys iterating over an arithmetic sequence of integers for which the compiler was unable to find an array (although one may exist);
• loop_body() && args(min, max, stride, ..) will match all the loop bodys iterating over a particular arithmetic sequence of integers; and

• loop_body() && args(min, max, stride, array) will match all the loop bodys iterating over an array, a reference to which will be bound to pointcut parameter array.

The sample advice shown in Listing 3.6 executes (using Java threads with a cyclical loop scheduling) all of the loops contained in class LoopsAJTest in parallel, which are recognised as iterating over a range of integers. As shown, the loop() pointcut is ideally combined with the ’worker object creation pattern’ [Lad03], which creates new Runnables to execute join points on separate threads.

Listing 3.6: Loop parallelisation inside loop body using Java Threads.

```java
void around(int min, int max, int step): within(LoopsBodyAJTest)
    && loop_body() && args (min, max, step, ..) {
    int numThreads = 4;
    Thread[] threads = new Thread[numThreads];
    for (int i = 0; i<numThreads; i++) {
        final int t_min = min+i;
        final int t_max = max;
        final int t_step = numThreads*step;
        Runnable r = new Runnable () {
            public void run() {
                proceed(t_min, t_max, t_step);
            }
        };
        threads[i] = new Thread(r);
    }
    for (int i = 1; i<numThreads; i++) {
        threads[i].start();
    }
    threads[0].run();
    try {
        for (int i = 1; i<numThreads; i++) {
            threads[i].join();
        }
    } catch (InterruptedException e) { }
}
```
3.2 Aspect-oriented model for synchronization

This section introduces a mechanism called synchronization which is used in concurrent programs for thread safety control. Section 3.2.1 introduces the basic functionality of synchronization in Java. Section 3.2.4 illustrates how synchronization as a concern is usually tangled and scattered across a concurrent program and, therefore, ideally should be separated from the base code. It presents an object-oriented model for synchronization, and it is found that the underlying abstractions for describing synchronization can cause significant problems when writing concurrent programs. Section 3.2.5

3.2.1 The synchronization model in Java

Synchronization is not simply shorthand for ‘critical section’ or ‘mutex’. While mutual exclusion is one element of the semantics of synchronization, there are two additional elements: namely the visibility and the ordering of each thread. Fortunately, this situation, in which there is a locked object inside a critical section, is similar to the data shared among computational nodes in a distributed environment. The statement is shown in Listing 3.7

Listing 3.7: Synchronized statement in Java 5.

```
Synchronized Statement

synchronized (Expression) Block
```

Two kinds of synchronization mechanisms in Java should be distinguished, namely, synchronized block (Section 3.2.2) and synchronized method (Section 3.2.3).

3.2.2 synchronized block

As a synchronized block, a lock is assigned to the object and ensures that only one thread at a time can access the code. Thus, before a thread starts to execute a synchronized block, it first grabs (locks) the lock on the associated object. Another thread will not be able to execute the code until the first thread has finished and has released the lock. Listing 3.8 is an example of synchronizing an object `t` of class `Test`. This example will deadlock if a single thread is not permitted to lock a lock more than once.
CHAPTER 3. CONCURRENT CONCERNS

Listing 3.8: Example for synchronized method.

class Test {
    public static void main(String[] args) {
        Test t = new Test();
        synchronized(t) {
            synchronized(t) {
                System.out.println("made it!");
            }
        }
    }
}

prints:
made it!

3.2.3 synchronized method

A synchronized non-static method locks the object to which the method belongs on entry to the method, and unlocks it on exit. A single thread may hold a lock more than once. If it already owns the lock, it increases the lock count on entry to the method, and decreases it on exiting. The lock blocks calls to all synchronized methods in the object, not simply other calls to that particular method. The lock acquired by synchronizing an object is the same as the lock which is implicitly acquired by synchronized methods. If method $m$ is synchronized, then an object executing it must be locked before transferring control. No further progress can be made until the current thread can obtain the lock. If there is a target reference, then the target must be locked; otherwise the Class object for class $C$, the class of the method $m$, must be locked. Control is then transferred to the body of method $m$ to be invoked. The object is automatically unlocked when execution of the body of the method has been completed, whether normally or abruptly. The locking and unlocking behaviour is exactly as if the body of the method was embedded in a synchronized statement.

The synchronized instance method in Listing 3.9 is syntactic shorthand for the code in Listing 3.10. The lock keeps calls to all synchronized methods out of the object generated from class Test, not simply calls to method $doit()$.

Adding or deleting a synchronized modifier to a method does not break the compatibility with existing binaries. If a method $m$ is not synchronized, control is transferred to the body of the method $m$ to be invoked. In the remaining part of the thesis, unless otherwise stated, both synchronization of an object and a synchronized method will be called a 'synchronized block' in the source code.
CHAPTER 3. CONCURRENT CONCERNS

Listing 3.9: Example for synchronized method.

class Test {
    ArrayList list = new ArrayList();
    public synchronized doit() {};
}

Listing 3.10: Example for equivalent synchronized block.

class Test {
    ArrayList list = new ArrayList();
    public doit() {
        synchronized (this) {}
    }
}

3.2.4 An object-oriented model for synchronization

Thread safety can be unexpectedly subtle because, in the absence of sufficient synchronizations, the ordering of operations in multiple threads is unpredictable and sometimes surprising. Unsafesequence in Listing 3.11, which is supposed to generate a sequence of unique integer values, offers a simple illustration of how the interleaving of actions in multiple threads can lead to undesirable results.

This behaves correctly in a single-threaded environment, but not necessarily in a multi-threaded environment.

Listing 3.11: Non-thread-safe sequence generator.

public class UnsafeSequence
{
    private int value;
    //Returns a unique value
    public int getNext() {
        return value++
    }
}

The problem with UnsafeSequence is that, with some unlucky timing, two threads could call getNext( ) and receive the same value. The increment notation, nextValue++, may appear to be a single operation, but will, in fact, be compiled by a sequence of three separate operations, namely, read the value, add one to it, and write the new value. Since operations in multiple threads may be arbitrarily interleaved by the runtime system, it is possible for two threads
CHAPTER 3. CONCURRENT CONCERNS

...to read the value roughly at the same time, both see the same value, and then both add one to it. The result is that the same sequence number is returned from multiple calls in different threads, whereas the presumed intent is that a sequence of distinct values will be created.

Within a program, the code segments which access the same object from separate, concurrent threads are called ‘critical sections’. A critical section can be a block or a method and is identified with the `synchronized` keyword. The Java platform associates a lock with every object and the lock is acquired when a critical section associated with that object is entered. `UnsafeSequence` can be fixed by making `getNext()` a synchronized method as shown in class `Sequence` in Listing 3.12, thus preventing an unfortunate interaction. The method declaration for `getNext()` contains the `synchronized` keyword. Other threads cannot call a synchronized method on the same object until the object is unlocked. The acquisition and release of a lock is done automatically by the Java runtime system. This ensures that race conditions cannot occur in the underlying implementation of the threads, thus ensuring the integrity of the data.

Listing 3.12: Thread-safe sequence generator.

```java
public class Sequence {
    private int nextvalue;
    //Guarded by keyword synchronized
    public synchronized int getNext() {
        return nextvalue++
    }
}
```

This section presents the new models, which are triggered by [HG06], for synchronization, which have a structure appropriate to aspect-oriented parallelisation using AspectJ. The models are designed in such a way that the points of the synchronization where the parallelisation control can happen are also valid AspectJ join points. The examples focus on nested synchronization, which may occur in a distributed system with fixed synchronized objects, which can be written in the form shown in Listing 3.13. It is assumed that the different instances of the body of the synchronization can be executed in no particular order; thus, executing the synchronized block body with certain objects of o and p does not impact the results of execution.

The new model `NestedSynchronizedBlock` consists of just one component,
as follows:

- **NestedSynchronizedBlock** super class is used to execute the synchronization, which contains a method `synchronizedDoSync` representing a nested synchronized block body.

In the model, the method `run()` represents the body of the nested synchronized block which implements the `NestedSynchronizedBlock` super class. An instance of `NestedSynchronizedBlock` holds a method `synchronizedDoSync()` to perform the actual synchronization.


```java
class Concretesynchronization extends RunnablesynchronizedBody {
    ...  
    public void run() {  
        synchronizedDoSync(o, p);
    }

    public void synchronizedDoSync(Object o, Object p) {
        ...  
        synchronized(o) {  
            synchronized(p) {  
                /* synchronized body. */
                * Concrete synchronization function. */
            }
        }
    }
}
...

/* Where the regular nested synchronized block would be */
RunnablesynchronizedBody sync =  
    new Concretesynchronization(objectA, objectB);
sync.run();
...
```

The models in the above example have been designed so that the nested synchronization function and synchronization block body can be encapsulated into an object. The creation and manipulation of such objects can then be recognised in AspectJ aspects. Thus, AspectJ can be used to define the synchronization strategy and apply the concurrent strategy accordingly.
3.2.5 synchronization join point

synchronization is a concern which developers encounter whenever guarded access to a resource is required. Devices, files and memory are all things which may require synchronization. synchronization is also an ideal way to deal with thread control and locking management, but the way to invoke interaction with synchronization suffers from the tangling problem, since it is not easy to express this crosscutting concern and encapsulate it for reuse in both a multi-threaded and distributed environment. Although join points for synchronized methods have been addressed, in [BK07], the synchronized block has not yet been treated in AspectJ [Asp] or in any other AOP framework [Bon05].

A synchronized method locks the monitor associated with the instance of the class (or the class, if it is a static method) and prevents others from doing so until the return from the method. A synchronized block can lock any monitor and can have a scope smaller than that of the enclosing method. A synchronized method can be picked out by the existing method called pointcut\(^1\) in AspectJ using the synchronized keyword. However, the selection of a synchronized block has not been supported in AspectJ to date. At bytecode level, unlike a synchronized method, a synchronized block is represented as a matched pair of entermonitor and exitmonitor bytecode instructions. Because one entermonitor can be statically (but not dynamically) paired with more-than-one exitmonitor, identifying the appropriate matched pair is non-trivial.

For the synchronization of an object, the locked object is passed via the arg construct of AspectJ, to which a var value is bound at the bytecode level. Additionally, an extra argument will be bound to the monitorenter and monitorexit, for the lock itself.

The pointcut definition of the model is:

- \(\textit{synchronized()} \land \textit{args(object)}\) matches all the executions of synchronized blocks with the named object as the synchronizing object.

The following is a possible way to express pointcuts and minimum advice codes:

- \(\textit{proceed(...)}\) inside \textit{around} advice for

\(^1\)This fragment of code can be found in AspectJ tutorial at http://dev.eclipse.org/viewcvs/indextech.cgi/aspectj-home
synchronized() & args(object) pointcut, which processes the executions of synchronized block, including its synchronization.

The implementation of the model, which is described in more detail in Chapter 4, uses the newly developed pointcut designator (PCD), namely synchronized, in conjunction with the args construct of AspectJ to expose the context of the synchronized block join point. The locked object is taken as the argument of the args construct of AspectJ.

The sample advice shown in Listing 3.14 executes in parallel. The list is originally executed in a synchronization block to avoid other threads reading or writing list while performing the method fooList(). While inside the synchronized block, it is thread-safe for list, as long as the method foo is disinterested in the order of execution. Thus the inner side of the synchronized block can be parallelised to enhance the performance or provide other implementation strategies. As shown, the synchronization() pointcut combines with the 'worker object creation pattern' [Lad03], which creates new Runnables to execute join points on separate threads. This piece of code is quite useful to re-split the original synchronized code such as when transforming it to transactional memory.

3.3 Miscellaneous

A more subtle problem may occur in the concurrent program pattern selection approach, which also affects the ability of aspect-oriented programs to evolve: the use of naming conventions or other ways of structuring code in patterns so that these can be captured by some sort of crosscut mechanism. Problems with evolution and obliviousness also occur in this case because programmers need to keep these conventions in mind. This problem can be distinguished as being an arranged patterns problem. It should be noted that this problem can also exist if the crosscut definition is separated from the aspect code (such as with AspectJ’s abstract pointcut separate composition rules, which only decouple the crosscut definition from the aspect naming convention). To completely avoid the situation in which aspect programmers rely on arranged patterns, crosscut languages need to be made sufficiently expressive. More expressive crosscut languages enable more complex patterns to be written for the concurrent programs. This eliminates the arranged pattern problem, leading to less coupling between aspects and the concurrent program.

class SynAJTest{
    private List list;

    public setList(List list) {
        this.list = list;
    }

    public void foo(Object obj){
        // do something with obj
    }

    public void fooList() {
        synchronized(list){
            Iterator i = list.iterator(); // Must be in synchronized block
            while (i.hasNext())
                foo(i.next());
        }
    }
}

void around(Object parentList): within(SynAJTest) && synchronized() && args (parentList, ..) {
    int numThreads = 4 ;

    Thread[] threads = new Thread[numThreads] ;
    ArrayList list = (ArrayList)parentList ;
    int step = list.size() / numThreads ;

    for (int i = 0 ; i<numThreads ; i++) {
        final int t_min = i*step ;
        final int t_max = t_min+step;
        Runnable r = new Runnable () {
            public void run() {
                proceed(((ArrayList)list.subList(t_min, t_max)) ;
            }
        };
        threads[i] = new Thread(r) ;
    }

    for (int i = 1 ; i<numThreads ; i++) {
        threads[i].start() ;
    }
    threads[0].run() ;
    try {
        for (int i = 1 ; i<numThreads ; i++) {
            threads[i].join() ;
        }
    } catch (InterruptedException e) { }
}
This section firstly introduces the basic block as a concern, which specifies the basic sequence control flow as a region; and then summarize Section 3.3.1 gives a scenario which the basic block is devised to handle, which comes under the consideration of (1) iterated behaviours in the control flow, or (2) transformation from a sequential to a parallel strategy. A use case is presented in Section 3.3.2 to illustrate the basic block.

3.3.1 User defined basic block

This section focuses on resolving the problem of computational region selection based only on method behaviour. The approach is to capture simplified sequences of method calls, since these sequenced methods are believed to map with a set of actions in a sequence diagram, which can hopefully be made to be parallel. This reasoning is effectively declarative, since each unit in the sequence diagram has a represents relationship with a corresponding method in the source code. The assumption is made that capturing regions of code is the key means of achieving the transformation from sequential to parallel program.

Thus, this kind of basic block is composed of a sequence of statements which can be picked out by specifying the beginning and end points, e.g.: capture (beginpoint A, endpoint B), where A is a point over the base model which denotes the start of the computational flow, and B is a point over the base model which denotes the end of the computational flow. Here, another assumption is that, although the point B is the behaviour next to pointcut A’s behaviour, they cannot form a nested situation. (In a nested situation, e.g. A and B are both inside a loop, and B is executed after A at some stage while the loop is executing. However, in a single round, B is never executed after A.) Given the fact that both the base model and the beginning and end behaviours are sets of concept definitions, it is possible to substitute every sequence behaviour in the mappings for its corresponding selection in the sequence control flow model. Some new concepts have been introduced here, i.e. the execution place of a block join point is composed by the start and end shadow which created the block and also the enclosing expressions. Thus, in this case, the argument of this block join point is simply the argument in which the start and end join point shadows are used as arguments. These join points could also be a block join point.
3.3.2 Brief draft of basic block

A user-defined basic block can be generated either from a user requirement or from a demand generated by the sequence diagram or another design level behaviour description. In the first attempt, the approach is simply to generate a user-defined join point from a user requirement, to demonstrate the feasibility of selecting a sequence control flow as a basic block. As shown in Figure 3.2, let TestA in (a) be a global class inside which run() is a public method and start(), finish() and doing() are private methods called inside run(). Let TestB in (b) be a global class, inside which run() is also a public method. However, in this class, the private methods called inside run() are changed to begin(), end() and during(). In the sequence diagram, both of the run() methods should be recognised as the same, or a similar, computational flow, as shown in (c); an ‘enter’ method followed by an ‘exit’ method with other process methods in between. Here a method call is said to be the local atomic unit but, in an ideal instance, apart from method calls, any statements inside method run() can be treated as being local atomic units, such as access field statement, loop clause, switch clause, etc. The purpose here is to pick out the computational flow model shown in (c) as a basic block.

```
class TestA {
    private int a;
    public void run() {
        start(a);
        doing(a);
        finish(a);
    }
}(a)
```

```
class TestB {
    private int b;
    public void run() {
        begin(b);
        during(b);
        end(b);
    }
}(b)
```

![Sequence diagram](image)

Figure 3.2: Examples of two block join points ((a),(b)) and corresponding computational flow model (c).

The two method calls, start() and finish(), are the requirements of the user. Apart from setting the beginning and end points of the block, the block join point can also add conditions in the middle. Although the aim is to recognise block join points with conditions, it is also important to make a detailed investigation of the various requirements to weave before, after, and around advice for a user-defined block.
3.4 Summary

This chapter has aimed to show the general parallel mechanisms and drawbacks of the design of these examples in terms of modularity. The examples indicate code-tangling, and this occurs as a direct consequence of scientific model concerns and parallel computing concerns crosscutting each other. The section has also reviewed the ability to characterise and recognise certain behaviour which lies at the root of AOP. Thus, the aspect-oriented implementation can be successful if the join points, at which the aspects intervene, can be clearly expressed and characterised. The next chapter will review two join points aiming to separate the above concerns in AOP.

The reason for introducing the expression for crosscutting concurrent concerns is to avoid limiting the expressiveness of the AOP language from its inception. By further combining this pattern-based concurrent concern with an extensive reification of static and dynamic program properties, a flexible join point model needs to be created to capture this concern. Further clarity and understanding of the aspects should also be provided by clearly splitting the what and where, and by possibly using specialised join point to specify the parallel place. The need for an understandable specification of crosscut behaviour in a concurrent program is the main motivation for the use of a concurrent concern.

This chapter thus presented the motivation for separating crosscutting concurrent concerns. It has investigated why the separation of concern in concurrent programs is important, what the limitation is if there is crosscutting parallel concern in concurrent programs, and how to circumvent this limitation to achieve a better system modularity. Section 3.1 reviewed the work in [HG06] to separate loops concern from the concurrent base code and showed the ability to rearrange parallelisation. It proposed the benefit and the possibility of separating these kinds of concerns in OO and also triggered the invitation of an extended inner loop join point model. Section 3.2 presented that the synchronization could be encapsulated with in an aspect in concurrent program and also brief presented synchronized block join point idea [Xi06], which is triggered by loops join point. The synchronization join point also has the ability to rearrange parallelisation in a concurrent program, which helps to implement new parallel strategies inside a synchronized block. The key idea is to encapsulate the handling of each parallel mechanism in a corresponding AspectJ join point, that is, to make this information visible and modifiable in the interface of the components, by refactoring the
application.

Crosscut definitions should avoid the tight coupling of an aspect to the base program. A concurrent program suffers from coupling through the arranged parallelisation problem. Advanced AOP languages should weaken the coupling of the aspect to the base program and hence, provide crosscuts which are more robust toward evolution. Avoiding this problem requires an expressive model which offers a powerful mechanism to describe the underlying concurrent points crosscut by an aspect, which will be introduced in Chapter 4.
Chapter 4

Fine grained concurrent join points

AOP shows improvements in modularity, despite having some shortcomings in terms of expressiveness. Some of these shortcomings are caused by a tight coupling between a concern and the program. An example is defining a crosscutting concern by enumerating the join points by name, or according to certain naming conventions. The join points, which can be exploited by AspectJ, (the most popular aspect-oriented extension to Java), are located at the interface of the components - either (packages, classes or methods) - which they mainly consist of method calls (and variations, such as calls to constructors) or field accesses. However, the constructs, which are usually required interesting for a concurrent program implementing an algorithm, are loops, synchronizations, if-then-else blocks and general blocks. Unfortunately, AspectJ does not recognise all of these constructs as being join points.

Based on Chapter 3, as more investigation are taken, more powerful and finer grained join points which support to separate concurrent crosscutting concerns are proposed and implemented. Therefore, this chapter proposes a novel generalisation of fine grained abstractions known as concurrent join point models suitable for the separation of parallel concerns in concurrent programs.

Firstly, Section 4.1 introduces the general idea of fine grained concurrent join point models, which can encapsulate the parallel concerns. Section 4.1.1 explains why it is crucial to extract structure unit mechanisms, such as loops, synchronized blocks, if-then-else blocks and general blocks, as join points. Section 4.1.2 explains why the approach is based on the bytecode and not on the source code, and also
explains the join point shadow.

Secondly, the extended synchronized block join point model is presented in Section 4.2. The work presented in this section has led to the publication of two articles presented respectively at the Foundations of Aspect-Oriented Languages workshop (FOAL’2008) and the Practices of Linking Aspect Technology and Evolution workshop (PLATE’2009). Section 4.2.1 introduces the extended work for a synchronized block join point [Xi06] and a related, synchronized block body join point, and the kind of synchronization these aim to recognise. Although the model is based on Java and AspectJ, it can potentially be applied to other languages. Section 4.2.2 enhances the join point model with a capability to remove the original synchronized block during advising. Section 4.2.3 introduces SynAJ and SynBodyAJ, two implementations, based on abc (aspect benchmark compiler) [ACH+05a], of a weaver capable of handling the synchronized block join point model. Section 4.2.4 embellishes the join point model with reflective capabilities, so that further information about the synchronization can be obtained by the aspect. Section 4.2.6 shows how to write aspects for concurrent programs using the enhanced synchronized block join point.

Thirdly, a general block join point model is addressed. Section 4.4.1 presents the general block join point and shows how to define a block. Section 4.4.2 explains the join point model with data and context exposure. It then introduces BlockAJ, which is also an implementation based on abc [ACH+05a], and shows how to write aspects for concurrent programs using the general block join point.

Finally, the if-then-else block join point model is introduced in a similar way. The if-then-else block join point and its related context exposure are given in Section 4.5.1. Its implementation, IfThenElseAJ in abc [ACH+05a], and related usage is presented in Section 4.5.2.

4.1 Fine grained join point models for concurrent programs

4.1.1 Aspects for the Java-Grande Forum benchmarks

Parallellising Java applications can be achieved by various means, but often leads to code-tangling. The Java Grande Forum benchmark suite [JGF] comprises a set of applications which come in two versions:
• the sequential version, which is aimed at single-processor machines; and
• the multi-threaded version, which is parallelised using Java threads.

Some applications are provided in both versions. The only difference between
the two versions is the extra statements introduced in the parallelised versions.
These extra statements are interlaced in the code which describes what is to be
computed, which corresponds to the sequential version.

This section investigates how AspectJ may be used to encapsulate the means
of parallelising the application in each version. The aim is to provide aspects
which, if woven into a sequential version, will produce a multi-threaded version.
The amount of refactoring required in the sequential version varies across the
benchmark suite codes.

The chosen test case is the TM (Transactional Memory) [WKL07] benchmark.
This consists of 5 Java files: JGFTMBenchSizeA, JGFTMBench-SizeB, JGFTMBenchSizeC,
IDEATest.java and JGFTMBench.java, and uses classes of the provided jgfutil
package, such as JGFInstrumentor (which handles the timers and displays the
results). It requires little modification before aspects can be used for parallelisation.

The computationally intensive part of the program is the synchronization
in method void tm_idea(byte[] text1, byte[] text2, byte[] key) in class
IDEATest. This method takes the data in parameter text1, enciphers it with the
key in parameter key and stores the results in the array defined by parameter
text2. The method in the sequential version is written in the form shown in
Listing 4.1.

Listing 4.1: Implementation of tm_idea in the sequential version.

```java
private void tm_idea(byte[] text1, byte[] text2, int[] key) {
    /* Declaration of local variables and initialisations */
    ...
    synchronized (arrayList) {
        /* Body of the synchronization */
        ...
    }
}
```

Using the transform technique in [WKL07] to slide the code into transac-
tional memory, each part of this synchronized object could be independent from
the other (i.e. the synchronization part could be re-organized), it is possible to spread the computation across several processors by splitting the range of the synchronized object index $i$ into blocks. For example, the multi-threaded version of $TM$ is written so that there can be several instances of the synchronized object which start with $i = i_{\text{low}}$ and stop when $i > i_{\text{upper}}$, for appropriate (distinct) values of $i_{\text{low}}$ and $i_{\text{upper}}$, as shown in Listing 4.2.

In order to make it easier to write an aspect which can intercept the original calls to the tm_idea and partition the synchronized block, the sequential version, into which aspects of parallelism may be woven, is refactored to allow for both the original use and the use of blocks. The iteration space of the synchronized object is thus accessible and modifiable from the object interface, and can be partitioned by a parallelism aspect.

Starting from the sequential version as a basis, the parallelisation concern can be successfully encapsulated in aspects. This concern is no longer tangled within the computation code, and the application is more flexible.

The aspect for multi-threading, shown in Listing 4.3, uses an around piece of advice for intercepting calls to tm_idea(*, *, *, int, int) made from within the original tm_idea(*, *, *). These original calls are not executed, but the advice proceeds with the execution of the refactored tm_idea method via inner instances of Runnable, each run in a Java Thread.

### 4.1.2 Source code versus bytecode

Although this may seem to be an implementation decision, choosing whether the join point is recognised at source code level or at bytecode level may completely change the model. The way synchronizations are programmed in Java is not necessarily directly reflected in the generated bytecode. For example, instinctively, most Java programers would consider the body of a synchronized block to be the lines of code within the curly brackets following the synchronized() statement. However, a synchronized block can also be written in different ways with the same effect, for example as a synchronized method, or with some of the statements displaced, as shown in Listing 4.4.

In addition, the synchronized object of a synchronized block may encompass several instructions, particularly when it involves a call to a method or a complex expression, as shown in Listing 4.5. Although the object may not seem to be part of the synchronized block body, it could always be refactored to make it so
Listing 4.2: Implementation of tm_idea in the multi-threaded version.

```java
private void tm_idea(byte[] text1, byte[] text2, int[] key) {
    /* Declaration of local variables and initialisations */
    ...
    for (int i = 1; i < /* Total number of threads */; i++) {
        thobjects[i] = new IDEARunner(i, text1, text2, key);
        th[i] = new Thread(thobjects[i]);
        th[i].start();
    }
    thobjects[0] = new IDEARunner(0, text1, text2, key);
    thobjects[0].run();
}
...

The multi-threaded version uses class IDEARunner, implementing Runnable, to split the loop across several Threads. The content of this class is the same as that of the tm_idea method in the sequential version, except that the bounds of the synchronized object are defined according to the thread id given to the constructor. */

class IDEARunner implements Runnable {
    int id, key[];
    byte text1[], text2[];

    public IDEARunner(int id, byte[] text1, byte[] text2, int[] key) {
        this.id = id;
        this.text1 = text1;
        this.text2 = text2;
        this.key = key;
    }

    private void run () {
        /* Declaration of local variables and initialisations */
        ...
        ilow = id * slice;
        iupper = (id + 1) * slice;
        if (iupper > text1.length)
            iupper = text1.length;

        synchronized(ArrayList.subList(ilow, iupper)) {
            /* Body of the synchronization */
            ...
        }
    }
}
```
Listing 4.3: Example aspect for parallelising TM using multiple Java Threads.

```java
public privileged aspect MultiThreadsTM {
    private final int NUM_THREADS;

    public MultiThreadsTM () {
        NUM_THREADS = Integer.parseInt(System.getProperty("threads","1"));
    }

    void around(int ilow, int iupper) :
        call(void IDEATest.c_idea(...))
        && args(*, *, *, ilow, iupper)
        && withincode(void IDEATest.tm_idea(*, *, *)) {
            Runnable[] runnables = new Runnable[NUM_THREADS];
            Thread[] threads = new Thread[NUM_THREADS];

            int tslice = (iupper - ilow) / 8;
            int ttslice = (tslice + NUM_THREADS - 1) / NUM_THREADS;
            int slice = ttslice * 8;

            for (int k = 0; k < NUM_THREADS; k++) {
                final int localilow = k * slice;
                int iuppertemp = (k + 1) * slice;
                if (iuppertemp > iupper)
                    iuppertemp = iupper;
                final int localiupper = iuppertemp;

                runnables[k] = new Runnable() {
                    public void run() {
                        proceed(arrayList.subList(localilow, localiupper));
                    }
                };
            }

            for (int k = 1; k < NUM_THREADS; k++) {
                threads[k] = new Thread(runnables[k]);
                threads[k].start();
            }

            runnables[0].run();

            for (int k = 1; k < NUM_THREADS; k++) {
                try {
                    threads[k].join();
                } catch (InterruptedException e) {
                }
            }
        }
    }
```
CHAPTER 4. FINE GRAINED CONCURRENT JOIN POINTS

Listing 4.4: Simple examples of equivalent synchronization.

```java
public void someMethod() {
    synchronized (obj.getUnit()) {
        /* A */
    }
}

public void someMethod() {
    Object unit = obj.getUnit();
    synchronized (unit){
        /* A */
    }
}
```

(for example through a temporary object). Moreover, the compiled code does not necessarily reflect the way a complex expression has been written in the source code.

Listing 4.5: synchronization with a call to a method.

```java
public void someMethod() {
    synchronized (this) {
        /* A */
    }
}

public void someMethod() {
    synchronized (Object obj = this) {
        /* A */
    }
}

public synchronized void someMethod() {
    /* A */
}
```

Since the main concern is to recognise the behaviour of the code, rather than the way it was written, the choice has been made to base the representation of the synchronized block at the bytecode level, rather than the source code level. As a result, the representation is more robust to variations in programming style. However, this choice introduces limitations regarding (a) the potential specific handling of an abrupt exit, and (b) the nature of the control-flow graphs. Indeed, the synchronized block join point model expects a reducible (or well-structured
[ASU86, Muc97]) graph. The Java source-code produces a bytecode with reducible control-flow graphs, but this is not necessarily the case for a bytecode produced by other means. A bytecode approach also follows the evolution of AspectJ, which, since version 1.1, does the shadow-matching and weaving only at bytecode level [HH04]. The theoretical definition of a join point shadow is as follows: "Every dynamic join point has a corresponding static shadow in the source code or bytecode of the programme. The AspectJ compiler inserts code at these static shadows in order to modify the dynamic behaviour of the programme" [BB04]. The detailed requirements of various kinds of concurrent applications in order to form a basis for the fine-grained join point model are as follows:

1. the ability to weave before advice, after advice and around advice for a synchronized block, genearal block and if-then-else block including the lock;

2. the ability to weave before advice, after advice and around advice for a synchronized block, genearal block and if-then-else block body excluding the lock; and

3. the ability to extract the context of execution at the join point.

4.2 The extended synchronized block join point model

This section presents the objective of the extended synchronized block join point model (SBJP). The extended SBJP is based on the SBJP as presented in [Xi06], but with more flexibility to meet the various kinds of requirements of concurrent applications. This model could be applied to various aspect-oriented systems, but the presentation focuses on AspectJ.

According to the different weaving requirements, it is necessary, not only to recognise the synchronized block, but also the synchronized block body, namely, the code inside the synchronized block brackets. Thus, a supporting synchronized block body join point (SBBJP) is also provided.
4.2.1 synchronized block in general case

In order to find a synchronized block, the initial approach is to sequentially pair the monitor instructions from the bytecode, which needs to construct the complete control flow graph of the code following the method described in [ASU86, Lad03]. This technique is based on finding dominators and successors. Given that a dominator contains a entermonitor, the synchronized blocks are defined as the region from the dominator to two of its branches, both of which end with a successor node containing a exitmonitor. One of the branches denotes the normal exit, and the other the exception exit. Figure 4.1 represents the control flow graph for the simple synchronized block.

![Figure 4.1: Control flow graph of simple synchronized block.](image)

In this example, the successors of the dominator node 1 are: (a) nodes 2, 3 and 4, which is a normal exit branch; and (b) nodes 5, 6 and 7, which contain an exception handler as an exception exit branch. Typically, a doubly-nested synchronized block is not recommended, but since it is not forbidden by the Java grammar, distinguishing nested synchronized blocks are necessary for detecting lock re-entrance or deadlock problems. Listing 4.6 shows the source code, and Figure 4.2 the corresponding control flow graph, for a doubly-nested synchronized block in which:

- the inner synchronized block consists of nodes 2, 3, 4, 5 and 6; its exception handler is 4, 5 and 6; and
- the outer synchronized block consists of nodes from 1 to 10; its exception handler is 7, 8 and 9.
Once the name of the locked objects, \( r_0 \) and \( r_1 \) in Figure 4.2, are registered, these two synchronized blocks can be recognised.

Listing 4.6: Doubly nested synchronized blocks.

```java
ArrayList r0 = new ArrayList();
ArrayList r1 = new ArrayList();

public void run() {
    synchronized (r0) {
        synchronized (r1) {
            //do something with r0,r1...
        }
    }
}
```

4.2.2 synchronized block with extra exit nodes

The above two branches of synchronization can become confusing when there are `return` or `break` statements in the body of a synchronized block, because there would then be multiple normal exit nodes from the synchronized block. As shown in Listing 4.7 and 4.8, what appears to be a simple synchronized block actually contains three exit nodes sharing the same dominator (see control flow graph in Figure 4.4 for a multi-exit synchronized block). In such a case, defining the points immediately after the synchronized block would be ambiguous. Therefore, instead of using two exit nodes for the join point model, the union of all branches which contain an exit node sharing the same dominator is considered as a single synchronized block.
CHAPTER 4. FINE GRAINED CONCURRENT JOIN POINTS

Listing 4.7: Example synchronized block containing a `return` statement.

```java
synchronized (Object obj) {
    if(obj.getStatus())
        return ;
}
```

Listing 4.8: Example synchronized block containing a `break` statement.

```java
while(true) {
    synchronized (Object obj) {
        if(obj.getStatus()) break ;
    }
}
```

However, some of the “extra” exit nodes may be illegal, even though they are consistent with Java grammar. In the case shown in Listing 4.8, the code can be compiled and run without error as a single threaded program. However, this could cause a severe unreleased lock problem in a multi-threaded program because of the misuse of the `break` statement, which exits a synchronized block without safely releasing the lock. If the program exits via the `break`, there will be no `exitmonitor` paired with the dominating `entermonitor`, and the object will be locked forever. The implementation of the SBJP could find this kind of improper use of synchronized blocks, and report it in the form of a misuse warning.

4.2.3 synchronized block body join point

According to the approach taken by [Bon05], there is a requirement to distinguish between picking out the whole synchronized block and just the body of the block. For example, to insert a piece of advice just after obtaining the lock, the application catches the body of the synchronized block as the shadow, so as to use proper `before` advice to get the time of acquiring, or releasing, the lock. To distinguish these two kinds of join points, a new SBBJP is implemented and `synchronized_body` is used as the keyword for the pointcut.

Catching the synchronized block body does not consider the verbose exception handling associated with synchronization. As shown in Figure 4.1, the shadow is just the code covering node 2. For a synchronized block with more than two exits, the shadow of the SBBJP will be a combination of several non-continuous
areas. Moreover, any \texttt{return} statement inside the \textit{around} advice for a block body should be followed by a \texttt{exitmonitor} statement to ensure that the lock is properly released by the block body.

### 4.2.4 Context exposure

Although synchronized blocks do not have arguments in the same way as other join points, such as method calls, they often depend on contextual information, particularly the locked object, to which programmers may require access. In addition, unlike synchronized methods, synchronized blocks expose the Java monitor bytecode instructions to the programmer, which could be regarded as another kind of context exposure to allow the SBJP to perform a flexible adaptation, at least including:

- eliminating unnecessary synchronization; and

- re-inserting synchronization that has previously been removed.

Knowing that a synchronized block is represented as a region of code between paired \texttt{entermonitor} and \texttt{exitmonitor} bytecode instructions enables the determination of the execution behaviour of the synchronized block in some detail at compile-time. The monitor instruction pairs can be considered as a first set of arguments for the synchronized block. In order to make this compile-time meaningful, only synchronized blocks, including the lock, are considered for this context exposure. This exempts SBBJP which is not able to have unnecessary synchronization removed. Moreover, the synchronized object inside the block may be of interest for applications as another contextual argument.

**Eliminating synchronization**

In order to reduce overheads, synchronization actions may sometimes be safely removed without compromising program semantics, thus improving performance. This elimination could be done manually, or by some algorithm. Ruf [Ruf00] provides an algorithm for removing unnecessary synchronization operations from statically compiled Java programs. One of the approaches is simply to prove that the program spawns no threads, making contention impossible, upon which all synchronization maybe removed. Unnecessary synchronization can also occur
when a JMM (Java Memory Model) lock is replaced by some other kind of lock (such as a distributed lock) when the program becomes distributed.

The removal of synchronization in Java corresponds to the removal of paired `entermonitor-exitmonitor` instructions and their associated exception handler at bytecode level. In order to handle possible exceptions caused by multi-threaded execution, the synchronized block actually contains at least one `try-catch` block just after it acquires the lock, and this `try-catch` block will end by releasing the lock. If there is any `return` inside the synchronized block, an extra `exitmonitor` is added before the `return` to ensure that the lock is released safely.

![Figure 4.3: Synchronized block (a) and associated code without synchronization (b).](image)

![Figure 4.4: Control flow graph for synchronization with multiple exits.](image)

Figures 4.3(a) and 4.3(b) respectively represent a traditional Synchronized block containing a `return`, and the associated code which eliminates the synchronization. The elimination of the synchronization removes the associated
try-catch block and the potential jump from synchronized block body to the exception handler. Figures 4.4 and 4.5 depict the control-flow graph of the bytecode corresponding to Figures 4.3(a) and 4.3(b), respectively. The dashed line in Figure 4.4 represents the potential exception flow. Inside the second block, the expression of Object obj is optional, which depends on whether the first instruction in the synchronized block is a local variable declaration.

Re-inserting synchronization

Since synchronization could be removed at compile-time by the advice of a SBJP, it is also necessary to be able to undo this modification. In order to simplify the re-insertion of the synchronization, as well as comply with the semantics of AspectJ, the advice of SBJP does not really remove the monitor instructions and associated exception handler at compile-time; instead, it creates a new method, rm.proceed(), which contains the code inside the synchronized block without synchronization. The original synchronized block including synchronization is contained in method proceed(). Thus any eliminated synchronization can be recovered simply by replacing method rm.proceed() by method proceed().

synchronized object

Both forms of SBJP and SBBJP must hold a locked object. The characteristics of this object dominate the behaviour of the synchronized block. Through analysing the attributes of the object, the execution details of a synchronized block could be re-arranged. Again, exposing this extra information could be useful, for example,
for certain synchronization schemes which require the program explicitly to check the legality of a thread in acquiring a lock, or for thread rescheduling.

4.2.5 Implementation in abc: SynAJ and SynBodyAJ

Due to the focus of the SBJP model being put on a join point model integral to AspectJ, the implementation uses abc [ACH+05a], an alternative AspectJ compiler, not only for the extensibility, which is the core design of abc, but also for the Soot [VRCG+99] framework, which provides most of the infrastructure to perform the analyses.

This section describes two extensions to abc [ACH+05a], known as SynAJ and SynBodyAJ, which implement SBJP and SBBJP and provide synchronized() and synchronized_body() pointcuts, respectively. The former picks out the synchronized block, including the lock, provides contextual information, and offers a flexible way to remove unwanted synchronization. The latter picks out the synchronized block body excluding the lock.

Shadow matching

In the Soot [VRCG+99] framework, as well as in abc [ACH+05a], a three-address representation of the bytecode, Jimple [VRH98] is used by which finding a synchronized block at bytecode level is made possible. Both SynAJ and SynBodyAJ extend the class which finds the shadows in each method so that it looks for both synchronized blocks and synchronized block bodies. For each method processed, the control flow graph and its corresponding dominator tree are built using the Soot [VRCG+99] framework toolkit. Then each synchronized block and associated body is identified.

abc [ACH+05a] provides two kinds of classes representing a shadow, namely StmtShadowMatch and BodyShadowMatch, both of which extend ShadowMatch. The former is used for pinpointing a specific statement or group of statements in the method, for example, when a method call pointcut is used. The latter is used when the shadow is the whole method body, for example, when a method execution pointcut is used. LoopsAJ, an extension of abc [ACH+05a] which implements a loop join point model [HG06], provides GroupShadowMatch when the shadow is a group of statements; this is also used in SynAJ and SynBodyAJ.

When dealing with before and after advice for both SynAJ and SynBodyAJ,
one of the requirements of abc [ACH+05a] is to insert nop operators in the shadow, at points where before and after advice might be woven. Given this, most of the abc infrastructure can already handle synchronized shadows for before and after advice.

Handling around advice requires a modification of the around-weaver, as in LoopsAJ [HG06]. However, the requirements for synchronized blocks concerning exception handling have to be satisfied in a more complicated way than those for loops. Also, due to the requirement to distinguish the synchronized block from its body, further modification inside the around-weaver is required. Handling of the new type of shadows and, more importantly, keeping the control flow graph and the synchronized join point structures consistent with the method content are performed using synchronizedAdviceApplication, a run-time controller class in abc. The controller can signal the weaver to do the specific matching work, before or after the original matching, or even replace the way of matching.

Both SBJP and SBBJP structures are created during shadow-matching, and different weaving operations modify the set of instructions in the methods. Weaving a around advice is implemented by placing the statements which form the shadow into a separate method and replacing them by an invocation of that method. Because the SBJP structure corresponds to the instructions which form the synchronized block, it has to be updated to take this operation into account.

Since the synchronized block has to deal with possible exceptions during the execution of around advice, Soot[VRCG+99] uses the class Trap to describe the exception handler in a method. Its methods setHead(), setTail() and setHandler() are utilised for all transformations of exception handlers. It is thus possible to implement a consistent exception handler behaviour for proceed() inside a around advice and other aspects.

**Context exposure**

Both SynAJ and SynBodyAJ expose the locked object as contextual information. In order to ensure that the transformation caused by the exposed context will not change the meaning of the synchronization, related field declarations and assignments are made to properly acquire and release the lock.

There are several potential ways to use the exposed context in SynAJ, as shown in Listing 4.9.

In case 1, an attribute of the object is changed before proceed().
Listing 4.9: Context exposure examples.

```java
ArrayList list = new ArrayList();
public void run() {
    synchronized(list) {
        Thread t = (Thread)list.get(0);
        //do something with list...
    }
}
//--------------------------------------------------
/* 1. change an attribute of the object */
void around(ArrayList list): synchronized()
    && within(*.run()) && args(list) {
    list.ensureCapacity(10);
    proceed(list);
}
//--------------------------------------------------
/* 2. add or remove n field of the object */
void around(ArrayList list): synchronized()
    && within(*.run()) && args(list) {
    list.clear();
    list.add("hello");
    proceed(list);
}
//--------------------------------------------------
/* 3. change the object with which to proceed */
void around(ArrayList list): synchronized()
    && within(*.run()) && args(list) {
    new Thread(new Runnable() {
        public void run() {
            ArrayList sublist = list.clone().subList();
            proceed(sublist);
        }
    }).start();
}
In case 2, a field of the object is removed, and then a new one is added before `proceed()`, which may cause compatibility problems for execution inside the synchronized block. For example, this occurs if the original synchronized block cast the object in `ArrayList list` to `Thread` type, but, during the `around` advice in case 2, the object in `ArrayList list` is `String` type. The programer of the aspect should avoid this incompatible misuse.

In case 3, the object passed to `proceed()` is changed to another new object. In order to make the `around` advice make sense, not only should compatibility be dealt with, as in case 2, but also the locking associated with the new object should be done properly. Due to weaving limitations inside `abc`, the original `around` advice just passes the object as an argument which is like the rewritten Java code shown in Listing 4.10. Although `sublist` can be accessed and used inside the synchronized block as an argument, it is not guaranteed that `sublist` is locked safely, even though `list` is. Thus further modifications are made in order to unify the name of the lock and its associated object.

Listing 4.10: Unlocked object by improper transformation.

```java
list = subList;
synchronized(subList){
    // Do something with subList as list...
}
```

`SynBodyAJ` has almost the same way of using the exposed context but, since `SynBodyAJ` does not take the lock into account, the modification for thread safety need not be made.

**Eliminating synchronization**

The exposed monitor can make use of the synchronized block more flexibly under the control of `SynAJ`, as described in Section 4.2.4.

In its around-weaver, `abc` provides the static class `ProceedMethod` which represents each `proceed()` method. `SynAJ` extends the method which uses the around-weaver to weave the join point so that the `proceed()` method can be used inside the `around` advice.

In order to implement the `rm.proceed()` method, to remove the synchronization of a synchronized block inside the `around` advice, `ProceedMethod` has been
extended by the RemoveProceedMethod. Since the removal operation can be undone, the original ProceedMethod is kept. When weaving the around advice, the extension is used by calling both of the following methods:

\[
\text{proceedMethod.doWeave()} \quad \text{and} \quad \text{removeProceedMethod.doWeave()}
\]

Thus there are two different methods for the around-weaver to call during around advice.

### 4.2.6 Aspects for synchronization

This section discusses some characteristics of the SBJP based on the implemented model. Firstly, the pointcut expression is given. Secondly, the re-entrant checking method is studied, and finally, a comparison is made between the functionality obtained from SBJP and that of another implementation using [BK07] as related work.

#### Writing pointcuts

For synchronization over an object, the locked object is passed via the arg construct of AspectJ, to which a var value is bound at the bytecode level. Additionally, an extra argument will be bound to the monitorenter and monitorexit, for the lock itself.

The pointcut definitions of the two models are:

- \(\text{synchronized()} \&\& \text{args(object)}\) matches all the executions of synchronized blocks with the named object as the synchronizing object;
- \(\text{synchronized\_body()} \&\& \text{args(object)}\) matches all the synchronized block bodies with the named object as the synchronizing object;

The following are possible ways to express pointcuts and minimum advice codes:

- \(\text{proceed(...)}\) inside around advice for \(\text{synchronized()} \&\& \text{args(object)}\) pointcut, which processes the executions of the synchronized block, including its synchronization;
- \(\text{rm\_proceed(...)}\) inside around advice for \(\text{synchronized()} \&\& \text{args(object)}\) pointcut, which processes the synchronized block but without synchronization; and
• proceed(...) inside around advice for
synchronized_body() && args(object) pointcut, which processes the synchronized block body.

The implementation of these two models uses the newly developed pointcut designators (PCDs), namely synchronized and synchronized_body, in conjunction with the args construct of AspectJ to expose the context of the synchronized block join point and its body join point. For both join points, the locked object is taken as the argument of the args construct of AspectJ. The extra mechanism used to remove unwanted synchronization is bound to the rm_proceed() construct, similar to proceed() inside the around advice.

For example, the following around advice would recognise the synchronized block shown in Listing 4.11, and process the code without synchronization.

```java
void around(Object obj): synchronized() && args(obj) && within(* *.run()) {
    rm_proceed(obj) ;
}
```

Listing 4.11: Simple synchronized block in Java.

```
ArrayList list ;
void run() {
    /* before synchronized block */
    synchronized(list) {
        /* before synchronized block body */
        /* do something with list ... */
        /* after synchronized block body */
    }
    /* after synchronized block */
}
```

## 4.3 Related work

Pairs of lock and unlock pointcuts have been recently proposed and used to pick out synchronized blocks [BK07]. These pointcuts match the entermonitor and exitmonitor instructions, respectively. Because pairs of lock and unlock pointcuts are used to catch synchronized blocks, the weaving capability is limited to
Table 4.1: Comparison of weaving capabilities.

<table>
<thead>
<tr>
<th></th>
<th>SBJP &amp; SBBJP</th>
<th>LOCK &amp; UNLOCK</th>
</tr>
</thead>
<tbody>
<tr>
<td>before() acquire and release lock</td>
<td>√</td>
<td>√</td>
</tr>
<tr>
<td>after() acquire and release lock</td>
<td>√</td>
<td>√</td>
</tr>
<tr>
<td>around() synchronized block</td>
<td>√</td>
<td>X</td>
</tr>
<tr>
<td>around() synchronized block body</td>
<td>√</td>
<td>X</td>
</tr>
<tr>
<td>context exposure</td>
<td>√</td>
<td>X</td>
</tr>
<tr>
<td>remove synchronization</td>
<td>√</td>
<td>X</td>
</tr>
</tbody>
</table>

*before* and *after* kinds of advice for the synchronized block. The paper [BK07] does not state whether the lock and unlock pointcuts have the capability of removing the synchronization. The system obviously cannot weave *around* advice or any proceed() for converting the synchronized block into transactions. Moreover, it seems dangerous to use just half of the monitor as the pointcut, in case the other half of the pointcut is misused.

The above situation, together with the context exposure capabilities, is summarised in Table 4.1, where *LOCK & UNLOCK* refers to the work presented in [BK07].

### 4.4 General block

This section presents the objective of the general block join point model. This model could be applied to various aspect-oriented systems, but the presentation here focuses on *AspectJ*. The sequence of the execution of a program is controlled by *statements*, which are executed for their effect and do not have values. Some statements *contain* other statements as part of their structure; such other statements are sub-statements of the statement. Statement *S immediately contains* statement *U* if there is no statement *T* different from *S* and *U* such that *S* contains *T* and *T* contains *U*. In the same manner, some statements contain
expressions as part of their structure.

A block is a sequence of statements. It can be used in both functional pro-
gramming and object oriented programing for directing the control flow. It can be
seen as local class declarations and local variable declaration statements within
braces. Listing 4.12 shows the definition of a block in Java.

Listing 4.12: Definition of block in Java.

```
Block: { BlockStatements }
BlockStatements: BlockStatement BlockStatements BlockStatement
BlockStatement: LocalVariableDeclarationStatement ClassDeclaration Statement
```

A block is executed by executing each of the local variable declaration state-
ments and other statements in order from first to last (left to right). If all of these
block statements complete normally, then the block completes normally. If any
of these block statements complete abruptly for any reason, the block completes
abruptly for the same reason. Because the start point and end point of the block
can either be a variable declaration or a method or even a block itself, identifying
the appropriate matched pair is non-trivial.

Just as for synchronized blocks, according to the different weaving require-
ments, it is necessary not only to recognise the block, but also recognise the block
body, namely the code inside the block brackets. Thus a supporting block body
join point is also provided.

### 4.4.1 General block join point

In order to define a general block, the initial approach is to define the start and
end statements and the statements between them sequentially from the bytecode,
which needs to construct the complete control-flow graph of the code following
the method described in [ASU86, Lad03]. As with the way to find a synchronized
block, the technique is based on finding dominators and successors. Given that
a dominator contains a start statement, the block is defined as being the region
from the dominator to a successor node containing an end statement. Figure 4.6
represents the control-flow graph for a simple block. In this example, node 1 is
a start statement and node 4 is an end statement; the successors of dominator node 1 are nodes 2, 3, 4 and 5.

\section*{4.4.2 Aspects for general block}

For a general block join point over a region, the start and end points for the shadowed area are defined by the start and end pointcut expression. As an aspect oriented language instruction, the block join points face a problem of how to define start and end points. As stated in Section 4.4.1, the bytecode level mechanism are the target points we are looking for. Since the core concern is to recognise the behaviour of the block code, rather than the way it was written, the choice has been made to base the representation of the block at the bytecode level. As a result, the presentation of the start and end points use pointcuts to express, which is also more robust to variations in the programming style. The object captured by those pointcuts can be passed via the \texttt{arg} construct of AspectJ, to which a \texttt{var} value is bound at the bytecode level.

The technique used to capture shadow of block join point is to some extent similar to capture shadow of synchronization join point. The approach of preparing join point shadows is to find the code between the “start” pointcut and “end” pointcut. Thus, first pick up the first statement which meet the requirement of the start pointcut and the last statement of the end pointcut, and then to weave advice in an “inside-out” order- that is, before advice that should run “closest” to the original code of general block join point is woven first.

The pointcut definitions of the general block pointcut models are:
• capture(pointcut1; pointcut2) && args(object) matches all the executions of a general block with the region which starts with the pointcut1 expression and ends with the pointcut2 expression; any arguments passed through pointcut1 and pointcut2 can be treated as arguments to the general block pointcut.

• capture_body(pointcut1; pointcut2) && args(object) matches the executions inside a general block with the region which starts with the pointcut1 expression and ends with the pointcut2 expression, exclusively; any arguments passed through pointcut1 and pointcut2 can be treated as arguments to the general block body pointcuts.

The following are possible ways to express pointcuts and minimum advice codes:

• proceed(...) inside around advice for capture() pointcut, which processes the executions of the general block, including its start and end pointcuts;

• rm_proceed(...) inside around advice for capture() && args(object) pointcut, which processes the general block, but without the start and end pointcuts; and

• proceed(...) inside around advice for capture_body() pointcut, which processes the general block body.

The implementation of these two models uses the newly-developed pointcut designators (PCDs), namely capture and capture_body, in conjunction with the args construct of AspectJ to expose the context of the general block join point and its body join point. For both join points, since there is no particular object associated with the block. The extra mechanism used to remove unwanted start and end pointcuts is bound to the rm_proceed() construct, similar to rm_proceed() inside the around advice for the synchronized block join point.

4.5 If-Then-Else block

This section presents the objective of the if-then-else block join point model. This model could be applied to various aspect-oriented systems to control parallelism. There are three types of if block, namely, if only, if-then and if-then-else.
The if statement enables the conditional execution of a statement or a conditional choice of two statements, executing one or the other but not both. Listing 4.13 shows the Java definition of a if statement.

Listing 4.13: Definition of if statement in Java.

```java
IfThenStatement:
  if ( Expression ) Statement

IfThenElseStatement:
  if ( Expression ) StatementNoShortIf else Statement

IfThenElseStatementNoShortIf:
  if ( Expression ) StatementNoShortIf else StatementNoShortIf
```

The Expression must have type boolean, or a compile-time error occurs.

A if-then statement is executed by first evaluating the Expression. If the evaluation of the Expression completes abruptly for some reason, the if-then statement completes abruptly for the same reason. Otherwise, the execution continues by making a choice based on the resulting value:

- If the value is true, then the contained Statement is executed; the if-then statement completes normally if, and only if, the execution of the Statement completes normally.

- If the value is false, no further action is taken and the if-then statement completes normally.

A if-then-else statement is executed by first evaluating the Expression. If evaluation of the Expression completes abruptly for some reason, then the if-then-else statement completes abruptly for the same reason. Otherwise, the execution continues by making a choice based on the resulting value:

- If the value is true, then the first contained Statement (the one before the else keyword) is executed; the if-then-else statement completes normally if and only if, the execution of that statement completes normally.

- If the value is false, then the second contained Statement (the one after the else keyword) is executed; the if-then-else statement completes normally if and only if execution of that statement completes normally.
4.5.1 If-Then-Else join point

In order to define an if-then-else block and avoid an unreachable statement, the initial approach is to identify the if-then-else block sequentially from the bytecode. This technique is also based on finding the if statement and all of the possible branches. Given that a dominator contains an if statement, the if-then-else blocks are defined as the region from the dominator to all of the successor nodes. Figure 4.7 represents the control flow graph for the simple if-then-else statement. In this example, the successors of dominator node 1 are: (a) nodes 2, 3 and 4, which is one of the branches; and (b) nodes 5 and 6, which is the other branch.

4.5.2 Aspects for If-Then-Else block

For the if-then-else block join point over a region, the start statement should contain an if statement, and the end point for the shadowed area is defined by all of the possible successors of the start statement. The boolean contained in the expression can be passed via the arg construct of AspectJ, to which a var value is bound at bytecode level.

The pointcut definitions for the if-then-else block pointcut models are:

- **ifelse(boolean bool) && args(bool)** matches all of the executions of the if-then-else blocks with the named boolean as the argument;
- **ifelse_body(boolean) && args(bool)** currently only matches all of the executions of the inner code of a if block without associated else counterpart, which also excludes the if statement itself. This can be extended to
match the execution of a full if-then-else blocks body, but the behaviour of its *before* and *after* advice should be discussed.

The following is a possible way to express pointcuts and minimum advice codes:

- proceed inside *around* advice for
  *ifelse()* pointcut, which processes the executions of if-then-else block, including all of its branches.

The implementation of these two models also uses new pointcut designators (PCDs), namely *ifelse* and *ifelse_body*, in conjunction with the *args* construct of *AspectJ* to expose the context of the if-then-else block join point and its body join point. Actually, in terms of the body join point, it is meaningful only with the *if* statement without the *else* block.

### 4.6 Summary

This chapter has demonstrated the synchronized block join point and its associated synchronized block body join point in *AspectJ*. The model achieves the reusability of the synchronized code and thread control management in Java to such an extent that the concurrency can be fully handled by a single aspect. More generally, the chapter has shown that *around* advice for join points is not limited to executing the *proceed()* method with given arguments, but can also address more complex and flexible behaviours, such as the *rm_proceed()* method, to remove synchronization overhead.

As discussed in Chapter 3, the code-tangling in concurrent programs can be resolved by providing mechanisms to separate cross-cutting parallel concerns. This chapter has suggested that parallelisation in concurrent programs can be separated and encapsulated into modular units to form a novel generalisation of parallel concerns. It studies the semantics of the set of parallelism mechanisms, and abstracts them as fine grained join point models. This chapter first demonstrates the extended synchronized block join point and its associated synchronized block body join point in *AspectJ*. Then the general block join point, and the if-then-else block join point are also presented.

Chapter 5 will present applications and evaluations for these fine grained join points, which shows the flexibility of using these mechanism to enable crosscutting
concurrent concerns to be encapsulated into a single aspect and achieve better system modularity.
Chapter 5

Applications and evaluation

This chapter investigates applications of the techniques presented in Chapters 3 and 4. It provides examples of the flexibility of using the aspects for concurrent programs and evaluates the system modularity attained using the techniques which enable concurrent programs to be synchronized using aspects.

Although some of the theory presented in Chapters 3 and 4 is not limited to a Java environment, all the examples used here are specifically fit for AspectJ and Java class files.

There are two main achievements which can be expected to have a substantial influence on concurrent programming:

1. Concurrent defects, such as unintentional race conditions or deadlocks are difficult and expensive to uncover and analyze, and such faults often escape detection before software is released. In the implementation of a concurrent system, there will be concerns that inherently crosscut the natural modularity of the rest of the implementation. The new set of join point models provides language mechanisms that explicitly capture such crosscutting structures. This makes it possible to program crosscutting concerns in a modular way, and achieve the usual benefits of improved modularity. The join point models provided in Chapters 3 and 4 can instrument concurrent applications with high quality concurrent testing and debugging tools, in the domains of data race detection, coverage analysis, performance monitoring, trace analysis, concurrent noise making and network load simulation. The original testing and debugging tools can be re-implemented and expressed both neatly and flexibly.
2. A systematic comparison between object-oriented and aspect-oriented versions of the same concurrent applications is given in order to assess to what extent each solution provides maintainable software decompositions. Measurable improvements can be achieved by the new set of join point models in terms of appropriate AOP metrics which are derived from an established suite of OO modularity metrics. The analysis is driven by fundamental modularity attributes, such as coupling, cohesion, conciseness, and separation of concerns. The aspect-oriented design of the join point models has exhibited superior stability and reusability through the changes for concurrent programming, as it has resulted in fewer lines of code, improved separation of concerns, weaker coupling, and lower intra-component complexity.

Section 5.1 demonstrates the flexibility of aspects by showing example aspects for synchronization schemes, which avoid the object-oriented code tangling and which can also be reused in many applications and woven as demanded.

Sections 5.2, 5.3, 5.4 and 5.5 give four scenarios in which the new join point models can be used to enhance flexibility in concurrent program, covering the areas of distributed environment [Bon05, KG07, BK07], IT security [ABD07], performance monitoring [Bon05] and thread control [Ruf00], showing the weaving capability of the model.

Section 5.6 demonstrates how to instrument a concurrent testing and debugging tool (Contest) using the new join point models. The main advantage is that the instrumentation part of the tool creating method, which usually contains little scientific contribution but consumes most of the work, becomes much easier to perform [CU05] and can achieve better results.

Before the remaining case studies, an evaluation is undertaken through a quantitative and qualitative comparison between Java and AspectJ. The measurement metric suites are introduced in Section 5.7, which take account of both the object oriented and aspect oriented requirement.

Two case studies are presented in Section 5.8 and 5.9; these sections each describe one application and present evaluation results for it. The evaluation results are obtained for both the Java and AspectJ implementation.

The chapter and its results are summarised in Section 5.10.
5.1 Aspects for flexibility in implementing synchronization strategies

Chapters 3 and 4 have shown that it is possible to use aspects for choosing a synchronization scheme. The examples showed a multi-threaded aspect, which would parallelise the original synchronized application using multiple Java threads. This section aims to demonstrate the flexibility introduced by the use of aspects by showing how other synchronization or parallelization strategies may be selected.

The example in this section describe several aspects that can intercept the original synchronization and apply parallelization in two similar applications differently. These aspects are made abstract, in a manner similar to abstract classes in Java. In abstract aspects, the advice is fixed but the pointcut descriptor is abstract and has to be concretised in non-abstract subaspects.

In the following example, the abstract pointcut $\text{synchronizedToParallelise}$ is expected to select the synchronization to re-parallelise and takes a synchronized object of class $\text{List}$ as its argument. Aspect $\text{ThreadBlockScheduling}$, shown in Listing 5.1, parallelises the selected synchronized block using block scheduling, each block being executed in a Thread that is managed in the advice.

Other parallelisation aspects could be implemented, for example using a distributed Fork/Join framework [LMGF05]. This aspect can be plugged into or removed from the original code at will, since their presence in the application is optional. The flexibility introduced makes it possible to choose and adapt a new parallelisation strategy, compared with the original synchronization scheme, depending on the application, without introducing any code-tangling in the computational units.

The example chosen to illustrate the use of these conversion of synchronization to parallelisation aspects consists of a Grid Route Finder, shown in Listing 5.2. However, these aspects could be applied to a large number of applications that contain synchronized blocks. All of the four abstract aspects presented above could be used, but only one concrete aspect is presented (shown in Listing 5.3), using the thread block scheduling. This concrete aspect uses $\text{SynAJ}$.
Listing 5.1: Aspect for converting synchronization to parallelization using block scheduling.

```java
public abstract aspect ThreadBlockScheduling {

    abstract pointcut synchronizedToParallelise(Map map);

    public final int THREADS_COUNT;
    public final int STRIDE;
    public ThreadBlockScheduling() {
        THREADS_COUNT=Integer.parseInt(System.getProperty("threads","1"));
    }

    void around(Map map): synchronizedToParallelise(map) {
        Thread[] threads = new Thread[THREADS_COUNT];
        int chunk_length = (map.size()/(THREADS_COUNT*STRIDE))*STRIDE;
        if ((map.length%THREADS_COUNT)!=0) chunk_length += STRIDE;

        for (int k = 0 ; k < THREADS_COUNT ; k++) {
            final int slice_min = k*chunk_length;
            int temp_max = (k+1)*chunk_length;
            if (temp_max>map.size()) temp_max = map.size();
            final int slice_max = temp_max;
            thread[k] = new Thread(new Runnable() {
                public void run() {
                    proceed(getSubMap(slice_min, slice_max, map));
                }
            });
        }

        try {
            for (int k = 1 ; k < THREADS_COUNT ; k++) {
                thread[k].start();
            }
            thread[0].run();
            for (int k = 1 ; k < THREADS_COUNT ; k++) {
                thread[k].join();
            }
        } catch (InterruptedException ie) { /* ... */ }
    }

    public Map getSubMap(int min, int max, Map map) {
        //Using some strategy to get un-conflicted sub-map...
    }
}
```
Listing 5.2: Route finder for a grid.

```java
public class Grid {
    public Map gridMap;
    public void run() {
        synchronized (gridMap) {
            gridMap.getRoute();
        }
    }
}
```

Listing 5.3: Aspects for grid route finder in parallel using block scheduling.

```java
public aspect GridMapBlockScheduling extends ThreadBlockScheduling {
    pointcut withinrun() : withcode (void Grid.run(..));
    pointcut mapSynchronization(): synchronized() && args(Map);

    pointcut synchronizedToParallelise(Map map):
        withinrun() &&
        mapSynchronization() &&
        args(map);
}
```
CHAPTER 5. APPLICATIONS AND EVALUATION

5.2 Applications for logging locked time

The model of synchronized block join point presented thus far takes an external view of the synchronized block; the points before and after the synchronized block are not within the synchronized block itself. As a consequence, however many iterations there may be for a given synchronized block, before and after advice will be executed only once. For some applications, for example for inserting a piece of advice just after entering the lock, it might be desirable to advise the synchronized block body.

The potential application uses the synchronized block join point (SBJP) and the synchronized block body join point (SBBJP) to capture various times associated with a lock; the concrete use cases in [Bon05] can all be implemented by before and after advice using these two models. It is assumed that high precision timer methods are available.

For example, to obtain the total time spent in the locked state, the inserted code needs to check the time via two pieces of advice, one before the lock is acquired and one just after the lock is released. In this simple example, an around advice can be used, as shown in Listing 5.4. The semantics depends on what getCurrentTime() does, and that most implementations would include time spent by other threads that happen to pre-empt the current thread (and that pre-emption can happen at any time).

Listing 5.4: Example to obtain time the lock remains locked.

```
void around(): synchronized() && within(*.run()) {
    time t = getCurrentTime();
    proceed();
    t = getCurrentTime()-t;
}
```

To obtain the time taken to acquire the lock, one advice is woven just before the lock is acquired and another advice is woven just after the lock is acquired, as shown in Listing 5.5. The SBBJP model is required here in order to specify the start of the body of the synchronized block as the weaving point. Two separate advices are necessary in this case, and the integer field t_start must be shared between them so that its value can be passed from one to the other.

The interleaving semantics for synchronized block join point need to be discussed for this example to get time spent for acquiring a lock. A pure lock join
Listing 5.5: Example to obtain time to acquire the lock.

```java
// Example to obtain time to acquire the lock.

time t_start, t_end;
before(): synchronized() && within(*.run()) {
    t_start = getCurrentTime();
}
before(): synchronized_body() && within(*.run()) {
    t_end = getCurrentTime();
}
```

point[BK07] could be a better approach to get time spent for locking. This is also one of the reason to introduce `synchronized_body` join point to explicitly pick out the “outer” or “inner” part of synchronization code base.

### 5.3 Aspects for synchronized block selection

As described in Section 4.7, writing pointcuts to select specific synchronized blocks can be difficult. The way pointcuts are written can also have impact on the performance obtained, in particular when the `cflow` and related `cflowbelow` constructs are used.

Synchronized block selection can be used to avoid re-entrant behaviour, which can be caused by doubly nested synchronized blocks. Since synchronized blocks cannot be named, it is impossible to use a name-based pattern to write a pointcut that would select a particular synchronized block. It is thus proposed that selection of synchronized blocks is made to rely on the data being processed.

For example, consider the synchronized blocks shown in Listing 5.6. The pointcuts to select the outer synchronization can be of two forms:

1. **data-based**:

   ```
   pointcut synchronizedList (ArrayList list):
   synchronized() && args(list); or
   
   pointcut outerCflowsynchronized():
   synchronize() && !cflowbelow(synchronized());
   ```

   In Listing 5.6, the pointcuts to select the inner synchronized block are written in three distinct forms using a mixture of the `thisJoinPoint`, `args` or `cflowbelow` [DT04] constructs in `AspectJ`. 
Listing 5.6: Example for nested synchronized blocks.

```java
Collection list1 = collection;
ArrayList list2 = arrayList;
synchronized(list1) {
    synchronized(list2) {
        /* ... */
    }
}

before(): synchronized() {
    Object[] args = thisJoinPoint.getArgs(0);
    if(args.length>=1)
        if(args[0] instanceof ArrayList)
            /* ... */
}

synchronizedList(ArrayList list): synchronized() && args(list);
befor(e(ArrayList list): synchronizedList(list)
    { /* ... */ }

outerCflowsynchronized(): synchronize() && !cflowbelow(synchronized());
befor(e(): outerCflowsynchronized();
    { /* ... */ }
```

These three approaches pick out the same inner synchronized block, but the resulting performance differs. The first two turn out to be identical; both rely on the processed data and make a selection based on an `instanceof` test. They generally perform better than the third approach, which uses `counter` based selection.

The first pointcut makes the selection according to the data type handled by the synchronized block. Although the data type check is performed at runtime (via `instanceof`), this could be optimised and determined at compile time. Indeed, `ArrayList` has no subtype, and its supertype is `Object` and `Collection`, thus, declaring the variable to be of type `ArrayList` at compile-time guarantees the same type at runtime. No additional runtime checks are required to check whether the pointcut matches the inner synchronized block.

The third pointcut makes the selection according to the state of the control flow. “`synchronized() && !cflowbelow(synchronized())`” selects all the synchronized blocks that are not below the control flow of any synchronized block, which is exactly what is required to match the outer synchronized block. However, the implementation of `cflow` and `cflowbelow` relies on a counter, or a stack, which is incremented and decremented on entry and exit, respectively, of
the join point described within \texttt{cflow()}, that is, in this case, all the synchronized block. A test to check the value of this counter is performed before entering each synchronized block.

As the results obtained with these three pointcuts in Figure 5.1 demonstrate, choosing either one of these ways of writing the pointcut descriptors can have an impact on the performance obtained, depending on the optimisation strategies of the JVM and on the test-case. Extensive work on optimising \texttt{cflow} related implementations [ACH+05b] has been done for \texttt{abc} and has been integrated subsequently into \texttt{ajc3}.

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{figure5_1.png}
\caption{Performance results for different selections schemas.}
\end{figure}

However, performance is not always paramount; if the processed data are not available, the \texttt{cflowbelow} pointcut can be used to pick out all of the inner blocks without knowing the processed data for nested synchronized blocks, which is helpful for anonymous selection.

The machine which is used for running the experiments has the configuration as follows:

\begin{itemize}
  \item a PC with a single Intel (R) Pentium (R) D processor at 3.00 GHz, with 1.00GB of RAM running Windows XP SP 2.
\end{itemize}
5.4 Thread control and rescheduling

More generally, if the synchronization cannot be eliminated at compile time, there are still ways to release the bottleneck presented by synchronization in multi-threaded programs. For example, in the pseudo-code shown in Listing 5.7, if the locked object `ArrayList sublist` could be split during the `around` advice, the original thread could be split into two sub-threads, each of which contains and proceeds a sub-list of the original `ArrayList sublist`. If the program is executing on a distributed system, during the `around` advice, the original thread could even ask remote nodes in a distributed dynamic aspect machine [KG07] to finish each subtask, namely each `proceed()` in Listing 5.7. The method `canBeSplit()` here is a dummy method indicate the situation that the list can be processed concurrently by two threads without racing condition.

Listing 5.7: Example to control and reschedule thread.

```java
ArrayList al = new ArrayList();
public void run() {
    synchronized(al) {
        if(al.size() > 0)
            Object obj = al.get(0);
            //do something with obj...
    }
}
void around(ArrayList list): synchronized()
    && within(*.run()) && args(list) {
    if(canBeSplit(list)){
        int size = list.size();
        new Thread(new Runnable() {
            public void run() {
                ArrayList sublist = list.clone().subList(0, size/2-1);
                proceed(sublist) ;
            }
        }).start() ;

        new Thread(new Runnable() {
            public void run() {
                list = list.subList(size/2,size-1) ;
                proceed(list) ;
            }
        }).start() ;
    }
    else
        proceed(list) ;
}
```
5.5 Converting synchronized blocks into transactions

The application presented in this section involves changing code so that it uses transactions instead of locks. Transactional Memory (TM) has become an active research area as it promises to simplify the development of highly scalable parallel programs. It aims to provide the scalability of fine-grain locking, but with the programming ease of coarse-grained locking such as synchronized locks. TM has seen a rise in research activity as the demand for scalable software increases in order to take advantage of future chip multiprocessors [McD05].

TM requires a programmer to mark code blocks which access shared data as transactions. Whenever a transaction executes, a runtime system records the transaction’s data accesses into a readset and a writeset. These sets are compared with the sets of other concurrently executing transactions for access conflicts, for example, write/write or read/write. If conflicting accesses are detected then one of the conflicting transactions is aborted and then restarted. A contention manager [HLMW03], decides which transaction to abort. A transaction that completes execution of its code block without being aborted can commit its writeset. TM implementations exist in a variety of flavours, including software-based (STM), hardware-based (HTM), and hardware/software hybrids (HyTM); refer to Larus and Rajwar [LR06] for details.

As discussed in [LR06, WKL07, ATKS07], transactions provide strong atomicity semantics for all referenced objects, providing a natural replacement for the critical sections defined using Java synchronized. Optimistic execution of transactions provides good parallel performance in the common case of non-conflicting object accesses, without the need for fine-grain locking mechanisms that further complicate correctness and introduce significant performance overhead. An easy way to provide these benefits for an existing parallel program is simply to replace each lock with a new construct, such as atomic{B}, that executes the statements in block B as a transaction [ATKS07]. The implementation that provides atomicity and isolation depends on a transactional memory model that is different from the Java Memory Model (JMM), but the general idea of this conversion can be readily implemented using the synchronized block join point.

Consider the simple string interning example shown in Listing 5.8. With
CHAPTER 5. APPLICATIONS AND EVALUATION

transactional execution, there is no need to use anything other than the non-locking map since the caller specifies its atomicity requirements, creating a single logical operation out of the get() and put() operations. Concurrent reads to the map happen in parallel due to speculation and the speculation is handled automatically by the system. The detailed implementation can be found in [LR06, WKL07, ATKS07].

Listing 5.8: Converting synchronized blocks into transactional memory.

```java
String intern() {
    synchronized(map) {
        Object o = map.get(this);
        if(o!=null)
            return (String)o;
        map.put(this, this);
        return this;
    }
}

aspect TestAspect {
    void around(Object map): synchronize(
        && withincode(* *.intern(..))
        && args(map){
            atomic{
                rm_proceed(map);
            }
        }
    )
}
```

Another example is from [HLMW03]. Consider a linked list whose values are stored in increasing order, a value is required to add into the list if it is not already in the list, and meanwhile, keep the same increasing order. This list is going to implement an integer set method by transactional memory which used to be implemented by synchronization. The original code in Listing 5.9 works fine when the size of list is not huge. But when the size is increased to large number and need more multiple thread to process the set method, the synchronization part is quite time consuming and become the bottleneck.

The fine-grained locks synchronization is guaranteeing correctness and avoiding deadlock become complex and error-prone. But synchronization also cause vulnerability to thread. The implementation of the insertion using synchronization is to avoid duplicated insert from different thread. But the disadvantage is its a bit time consuming especially when there are huge number of thread trying to do the insertion. As introduced in [HLMW03], various contention management
Listing 5.9: Example of set value in a linked list using synchronization

```java
public class IntSet {
    private List first;
    class List implements Cloneable {
        Integer value;
        List next;
        List(Integer v) {
            this.value = v;
        }
        List getNext() {
            return next;
        }
        public Object clone() {
            List newList = new List(this.value);
            newList.next = this.next;
            return newList;
        }
    }
    public IntSet() {
        List firstList = new List(Integer.MIN_VALUE);
        this.first = new List(firstList);
        firstList.next = new List(Integer.MAX_VALUE);
    }
    public boolean insert(int v) {
        List newList = new List(v);
        List firstList = this.first;
        synchronized(firstList) {
            List prevList = firstList;
            List currList = prevList.getNext();
            while (currList.value < v) {
                prevList = currList;
                currList = (List)currList.getNext();
            }
            if (currList.value == v) {
                result = false;
            } else {
                result = true;
                newList.next = prevList.getNext();
                prevList.next = newList;
            }
            return result;
        }
    }
    ...
}
```
schemes can be implemented and plugged in without affecting the correctness of the transaction code. Thus we can design, implement and verify an obstruction-free data structure once, and then vary the contention managers to provide the desired progress guarantees and transaction prioritization. Take the linked list insertion implementation as example, the transactional memory way’s of solution is depending on the value of the current node, the transaction either detects a duplicate or inserts the new node between the previous and current nodes, and then tries to commit. If the commit succeeds, the method returns; otherwise, it resumes the loop to retry the transaction. Here transactional objects are implemented by the \texttt{TMObject} class as in [HLMW03]. A transaction is a short-lived, single-threaded computation that either commits or aborts. A transactional object is a container for a regular Java object. A transaction can access the contained object by opening the transactional object, and then reading or modifying the regular object. Changes to objects opened by a transaction are not seen outside the transaction until the transaction commits. If the transaction commits, then these changes take effect; otherwise, they are discarded. Transactional objects can be created dynamically at any time. The creation and initialization of a transactional object is not performed as part of any transaction. A thread calls \texttt{beginTransaction()} to start a transaction. Once it is started, a transaction is active until it is either committed or aborted. In large scaled multiple thread program, this approach could perform better than synchronization. Thus, synchronization join point could be the best place to take the advantage of transformation for those code originally written with synchronization to transactional memory. As shown in Listing 5.10, the \texttt{TMObject} class is introduced to original code base which implemented in synchronization seamlessly. The detail of how the transactional memory work can be referred in [HLMW03]. Synchronization join point is a bridge connect the old and new implementation. Aspect with two pointcuts transfer Listing 5.9 into transactional memory. The original code is oblivious to the aspect which woven it into transactional memory or not.
Listing 5.10: Example of aspect to convert synchronization block into transactional memory.

```java
public aspect synToTM{
    List around(List list): call(List *.getNext()) {
        TMObject newNode = new TMObject(this);
        try {
            newNode.open(WRITE);
            proceed();
        } catch (Exception e) {
            if (canThrough(e))
                throw new Denied(e);
        }
    }

    boolean around(Integer value, List list): synchronized() && withincode(*.insert(value)) && args(list) {
        List newList = new List(value);
        TMThread thread = (TMThread)Thread.currentThread();
        while (true) {
            thread.beginTransaction();
            boolean result = true;
            try {
                result = rm_proceed(first);
            } catch (Denied d){}
            if (thread.commitTransaction())
                return result;
        }
    }
}
```
5.6 ConTest reimplementation for concurrent program debugging

Like concurrent program testing, concurrent program debugging, for events such as unintentional race conditions or deadlocks, is difficult. It is expensive to uncover and analyze such faults and they often escape detection. One reason for this difficulty is that the set of possible interleavings is huge, and it is not practical to try all of them. Only a few of the interleavings actually produce concurrent faults; thus, the probability of producing one is very low. Since the scheduler is deterministic, executing the same tests many times will not help, because the same interleaving is usually created. The problem of debugging multi-threaded programs is compounded by the fact that tests that reveal a concurrent fault in the field or in a stress test are usually long and run under different environmental conditions.

ConTest [EFN+01] is a tool used by more than fifty testing and developer teams in IBM for finding bugs caused by concurrency. It alleviates the need to create a complex testing environment with many processors and applications, and works by instrumenting the bytecode of the application with heuristically controlled conditional sleep and yield instructions. It has been shown in [CU05] that using AOP to find concurrent program bugs is both feasible and easy and that the benefits of the higher level of abstraction are significant. However, without the ability to instrument synchronization blocks, it is not easy to find certain concurrent bugs and therefore impossible to completely reimplement ConTest with AspectJ.

As shown in Listing 5.11, an aspect is implemented to demonstrate the capabilities of an AOP language using a SBJP. The aspect alters the class files to increase the likelihood of catching concurrent bugs, using ideas already implemented in ConTest. Special emphasis is placed on the instrumentation capabilities.

SynchronizeSleepNoise is an aspect based on a single pointcut and a single advice. The pointcut defines where the instrumentation is being done. The advice is a call to `sleep()` with a random parameter in the range $[0, 50]$ with a probability of $1\%$ for invoking the `sleep` method. This adds noise to the instrumented application as done by ConTest’s instrumentor. The difference, however, is that this aspect inlines the noise, whereas ConTest inserts new methods, which add
Listing 5.11: Sleep noise with SBJP.

```java
public aspect SynchronizeSleepNoise extends Thread{
private static Random rand = new Random();
pointcut noiseVictem(): synchronize()
  && cflowbelow(synchronize())

before(): noiseVictem() {
  try{ // noise
      if (rand.nextInt(100) == 1){ // activation
          sleep(rand.nextInt(50)); // type
      }
  } catch (Exception e) {};
}

after(): noiseVictem() {
  try{ // noise
      if (rand.nextInt(100) == 1){ // activation
          sleep(rand.nextInt(50)); // type
      }
  } catch (Exception e) {};
}
}
```

some runtime overhead. This example could easily be expanded to instrument special concurrent related methods, such as `sleep`, `yield`, `notify`, `notifyAll`, and so on. In addition, the type of noise could be altered to other types of noise that affect the interleaving of the program, all creating different kinds of heuristics.

The aspect `SynchronizeSleepNoise` is tested against several programs with documented bugs. These programs received a single parameter: the number of threads running simultaneously, and are categorized as low (2 threads), medium (7 threads), or high (15 threads). The tests are run 10 times for each category and for each configuration: once as the uninstrumented program, called “original” in the figure, then using ConTest with simple noise(“GetSetNoise”), and finally with the `SynchronizeSleepNoise` aspect(“SynchronizeNoise”). Four different programs were involved.

The first program is a concurrent version of the bubble sort algorithm. The bug in this program is that the programmer assumed the threads would finish their work without interruption and accessed the shared resources without enough synchronization. Only the getter method is synchronized but not the setter method. Figure 5.2 shows the results for this program. It clearly shows
that using \textit{SynchronizeSleepNoise} increases the chance of concurrent bugs being manifested.

![Bugs Revealed](image1)

Figure 5.2: Bugs revealed for different implementations of ConTest: Bubble Sort.

The second program is also a bubble sort program, but with a different bug. The programmer used a \texttt{sleep} statement to initialize all threads before they started working. The results in Figure 5.3 show the benefit in using this type of testing. By inserting the \textit{SynchronizeSleepNoise} in the original multi-thread test case, it shows AOP, especially the synchronized join point helped in increasing the chance of finding bugs.

![Bugs Revealed](image2)

Figure 5.3: Bugs revealed for different implementations of ConTest: 2’d Bubble Sort.

The third program is one that issues IDs for users. Each user requests and receives a unique ID. The bug is that the programmer assumed that incrementing the ID counter was an atomic operation and did not protect it. Only the getter method is synchronized but not the setter method. In Figure 5.4, \textit{SynchronizeSleepNoise} surpassed ConTest’s performance for medium concurrency, which could be the result of a low activation parameter used for ConTest.

In the last program, the programmer wanted to obtain the locks for two files in different places, but did not maintain a global partial order on the way she
obtained these locks. This introduced a deadlock to the program. The initial results shown in Figure 5.5 indicate the benefit of using this type of testing, by adding a \textit{SynchronizeSleepNoise} exactly after the first lock was obtained in one of the two accesses. As shown in results, \textit{SynchronizeSleepNoise} increased the chance of finding bugs in low and medium cases. Note that the heuristic is very simple and was not modified to suit the specific program. By tuning the frequency of adding noise and the type of noise, better results may be achieved.

Through the test case, it shows that with concurrent aspects, a useful testing tool could be implemented for exposing multi-threaded bugs with very little effort. It has been considered that AOP and AspectJ are important for implementing high quality open source and academic testing tools, in the domains of data race detection, coverage analysis, performance monitoring, trace analysis, concurrent noise making, network load simulation and many others [CU05]. If more tools follow this approach, AspectJ together with the use of concurrent aspects will create an incentive for additional research in the testing community.
5.7 Evaluation Metrics

The evaluation [HCN98, CK91] for the system aims at investigating the facets of the use of aspects for scientific software especially for numerical computing: their impact on flexibility and their impact on performance. The prospective framework should provide scientific applications with a flexible design. This flexibility is gained from two perspectives:

- the re-usability of the base code, which could be incorporated into many applications; and
- the decoupling of the concurrent concerns from the design and implementation of the base application.

This section describes the evaluation strategy for the proposed research; Section 5.7.1 gives the evaluation criteria for modularity measures from the system viewpoint of code re-useability, based on coupling, cohesion, and conciseness metrics, and Section 5.7.2 gives the evaluation criteria for decoupling modularity measures with respect to separation of concurrent concerns.

5.7.1 Evaluation criteria for coupling, cohesion, and conciseness

The quantitative assessment is based on the application of a metrics suite, which includes metrics for separation of concerns, coupling, cohesion, and size [CK91, KKG07]. These metrics are selected because they are the metrics which have mostly been applied in a number of empirical studies [CSF+06, FSG+08, FCS+08]. The coupling, cohesion, and size metrics are based on classic OO metrics [CK91]. The original OO metrics were extended to be applied in a paradigm-independent way, supporting the generation of comparable results. Furthermore, the metrics suite introduces four new metrics for quantifying separation of concerns. They measure the degree to which a single concern in the system maps to the design components (classes and aspects), operations (methods and advice), and lines of code. For all the employed metrics, a lower value implies a better result. The following subsection presents a brief definition of each metric, and associates them with the attributes measured by each one. The concern metrics require manual “shadowing” of the code, i.e. identifying which segment of code contributes to each concurrent concern in the application.
The C&K metrics [CK91] are used in [KKG07] to evaluate a proposed weaver because they provide the most comprehensive and best validated set of measures [HCN98]. The calculation of each metric for use with aspects is adapted by [KKG07], which is actually the guideline for the proposed AO weaver. The C&K suite involves measurement against several metrics, including number of children and number of methods per class. These individual results are then used in combination to assess system properties, such as testability and maintainability. For each metric, it will be shown how the value is calculated, and the individual results, in terms of the change in value from OO program to AO program or from original AO program to improved AO program.

The following values are proposed to evaluate the design and implementation of the AO weaver. These values are:

**Coupling between components**

Coupling between components [CK91] (classes, interface and aspects)(CBC) is a count of the number of other components from which elements are used i.e. calls or attribute accesses between components. CBC relates to the notion that an object is coupled to another object if one of them acts on the other. To apply this to aspects, aspects are considered coupled to classes only if the aspects explicitly name the classes. For instance, if the join point is `call(* *(..))`, then the aspect is not coupled to any classes. However, if the join point is `call(example.Test.methodName(..))`, then the aspect is coupled to `Test`.

Excessive coupling between objects is detrimental to modular design and prevents reuse. The more independent a component is, the easier it is to reuse it in another application. In order to improve modularity and promote encapsulation, inter-object class couplings should be kept to a minimum. The larger the number of couplings, the higher the sensitivity to changes in other parts of the design, and therefore, maintenance is more difficult. A measure of coupling is useful to determine how complex the testing of various parts of a design are likely to be. The higher the inter-object component coupling, the more rigorous the testing needs to be.

**Depth of inheritance tree**

Depth of inheritance tree (DIT) [CK91] is the maximum distance from a class node to the root of the tree. DIT is a measure of how many ancestor classes can
potentially affect this class.

The deeper a class is in the hierarchy, the greater the number of methods it is likely to inherit, making it more complex to predict its behaviour. Deeper trees constitute a greater design complexity, since more methods and classes are involved. The deeper a particular class is in the hierarchy, the greater the potential reuse of inherited methods.

**Lack of cohesion in operations**

Lack of cohesion in operations (methods and advices) (LCIO) [CK91] is the degree to which operations within a component are related to one another in terms of shared variables.

Pointcuts and advice blocks are considered to be methods to adapt this for AO evaluation. Only the synchronization and resource sharing real-time areas show a change in the LCIO.

**Weighted operations per component**

Weighted operations (methods and advices) per component (classes, interfaces and aspects) (WOC) [CK91] is a measure of the number of operations implemented within a component.

The number of operations and the complexity of operations involved is a predictor of how much time and effort is required to develop and maintain the component. The larger the number of operations in a component, the greater the potential impact on children, since children will inherit all of the operations defined in the component. Components with large numbers of operations are likely to be more application-specific, limiting the possibility of reuse.

**Vocabulary size**

Vocabulary size counts the number of components (classes, interfaces and aspects) of the system.

**Lines of code**

Lines of code is a software metric used to measure the size of a software programme by counting the number of lines in the text of the programme’s source code. It is typically used to predict the amount of effort required to develop a
programme, as well as to estimate programming productivity or effort once the software is produced.

**Number of attributes**

Number of attributes counts the number of attributes of each class or aspect.

### 5.7.2 Evaluation criteria for separation of concerns

The separation of concerns metrics measure the degree to which a single concern in the system maps to the design components (classes and aspects), operations (methods and advices), and lines of code.

**Concern diffusion over components**

Concern diffusion over components [KKG07] counts the number of classes and aspects whose main purpose is to contribute to the implementation of a concern and the number of other classes and aspects that access them.

**Concern diffusion over operations**

Concern diffusion over operations [KKG07] counts the number of methods and advices whose main purpose is to contribute to the implementation of a concern and the number of other methods and advices that access them.

**Concern lines of code**

Concern lines of code counts the number of lines of code whose main purpose is to contribute to the implementation of a concern.

**Concern Diffusion over LOC**

The metric Concern Diffusion over lines of code [KKG07] smears the number of transition points for each concern through the lines of code. A transition point is the point in the code where there is a “concern switch”. It is measured by shadowing lines of code in the application classes related to the specific concern that you are interested to investigate. After that, it is necessary to count the number of transitions points through the source code of every shadowed class.
5.7.3 Summary

These metrics suite in Sections 5.7.1 and 5.7.2 have been defined based on the reuse and refinement of some classical and OO metrics [CK91, HCN98]. The original definitions of the OO metrics were extended to be applied in a paradigm independent way, which supports the generation of comparable results.

The metrics suites also include new metrics for measuring separation of concerns [KG07, CFGF08]. The separation of concerns metrics measure the degree to which a single concern in the system maps to the design components (classes and aspects), operations (methods and advices), and lines of code.

5.8 Distributed lock migration

In the context of distributed Java programs, three distinct synchronization styles are encountered, namely, sharing data among multiple threads, sharing among clusters of JVMs and sharing among clusters of physical computers. The former is the logical concern of sharing data between concurrent threads; the latter two are practical concerns about the particular ‘distributed environment’ in which the code is executed. The problem is, how to migrate the logically necessary Java threads transparently across the physical vagaries (numbers of JVMs, numbers and configurations of physical computers, etc.) of the distributed environment.

In the context of Java, the JMM determines what values can be read in multithreaded environments. It also allows complete prediction of the values that are seen by each thread. Thus a lock acquired by a synchronized block needs to be thought of as a JMM lock.

To migrate a multithreaded program from one JVM to a cluster of JVMs, there would need to be another kind of cluster-wide lock, as seen in [BK07], for example. This is used to secure competition among the threads, possibly with conditional logic, to bypass the JMM lock and to use an alternative locking implementation, such as distributed locks or cluster locks. Meanwhile, the converted multithreaded program would be necessary to remove the original JMM lock by calling \texttt{rm.proceed()}, which executes the code inside the lock. Pseudo-code for this is shown in Listing 5.12.

It is worth mentioning that the new \texttt{ClusterManager} allows implementation of certain optimizations. For example, the lock manager can be coordinated with the object manager and can use synchronization boundaries to create a Unit
of Work. Then the object manager can batch all the field modifications that happen within the specific Unit of Work and propagate them to the other nodes in the cluster when the Unit of Work is completed upon distributed and local lock release.

Unlike the original synchronized block, which uses a JMM lock, the ClusterManager can define several lock types with different semantics, as follows:

- **Write lock** has the same semantics as regular Java synchronization and allows at most one single thread to acquire a lock.

- **Read lock** reconciles object changes but does not require blocking of the execution, e.g. multiple threads can acquire the same lock.

- **Concurrent lock** is a looser form of write lock. Multiple threads can be inside a concurrent lock at the same time, and can all write to the data, but, only the changes made by the last thread to exit the block actually take effect. This is nondeterministic and is thus generally used only with considerable caution.

The ClusterManager could apply several optimizations, such as escape analysis and reentrant locking. In a distributed system, greedy locking can give significant performance advantages. In this case, the lock is owned by the particular node until it is requested by another node and then it is transferred to that node. This reduces the number of calls to the lock manager.

This strategy could also be used in a cluster of physical computers. The code inside `rm_proceed()` could even be sent to a remote computational node during the execution of the `around` advice by a distributed dynamic aspect machine, such as that described in [KG07].
The use of AOP technology allows a distributed system to focus on implementation details for the services used in the clustering runtime system and transparently weave in the code into the application in order to get high-availability, and scalability. This approach to clustering provides great benefits to the end users, who can focus solely on implementing their business logic and still get the benefits of clustering without polluting their code with traditional API-based clustering.

A comparison of the Java and SBJP-instrumented implementations for the distributed lock migration case study has been undertaken, based on the suite of modularity metrics presented in Section 5.8.2. Modularity measures with respect to separation of concurrency concerns are based on concern diffusion over components, concern diffusion over operations, concern lines of code [CFGF08], and the concern diffusion over LOC [KG07]. Figure 5.6 presents the results. It shows how the distributed lock migration program was affected by the four concern measures. All four of the metrics are evidently reduced after the code re-programmed to use the synchronized block join point SynAJ.

![Figure 5.6: Concern measure for distributed lock migration.](image)
5.9 Record and Playback (R&P) for concurrent program testing

Testing and debugging of concurrent programs presents additional difficulties to the testing and debugging of sequential programs. One of such difficulty is non-determinism, which makes it harder to reproduce the behaviour of a program execution. Even if the concurrent processes are deterministic, non-determinism can arise from the possible different interleavings of process scheduling.

Most modern operating systems support kernel threads. This means that the operating system is in control of scheduling of threads and will swap out the current running thread after it has been running for a given period of time determined by the operating system or if it has been blocked on a system call. Clearly, any thread will only be able to execute a particular number of instructions during the time it was given to run.

Unfortunately, most operating systems and runtime systems – including the diverse versions of the Java Virtual Machine – do not offer any support to the tester facing this problem. Thus, the tester has to find alternative approaches in order to be able to adequately exercise his/her programs.

In this case study an approach to reproduce the behaviour of a Java multi-threaded program is presented. The term “record and playback” (R&P) [Del04] has been used to describe techniques and tools to register the input and output of a program execution and to use such data to automatically re-execute the same (or other) program and compare the results. The program under analysis (under testing or debugging) is instrumented with synchronized block body join points in such a way that each of the synchronization points (where the program accesses shared objects) are recorded. In a second execution the instrumented program can use this synchronization sequence to force the same sequence of accesses to the shared objects, in order to produce the same behaviour. In addition, the analysis of a given synchronization sequence created in a particular execution can help to create new synchronization sequences and possibly new behaviours.

The approach used in R&P to capture thread schedule is based on this kernel threading model and is very intuitive: count and record the number of instructions executed by each traced thread during the record stage and during the replay stage only allow traced threads to run for a number of instruction that was previously recorded for them.
During the record stage, when one of the traced threads starts running, the performance counter is configured such that it gets incremented for each retired instruction, thus providing a count for a number of instructions that get executed by a traced process. When a thread is preempted, the value in the counter is read and stored in the timing log file. The counter is then reset to start counting instructions executed by a next thread that is scheduled to run.

For example, in the code in Listing 5.13, the thread must obtain the monitor of myObject before executing doSomeOtherThing, in method myOtherMethod.

```
public class MyClass {
    myOtherClass myObject = new myOtherClass();

    public void synchronized myMethod() {
        doSomeThing();
    }

    public void myOtherMethod() {
        synchronized (myObject) {
            doSomeOtherThing();
        }
    }

    public void myUnsyncMethod() {
        doSomeOtherThing();
    }
}
```

Listing 5.13: Example of synchronized method/block.

The instrumented code for recording is shown in Listing 5.14. The call to beginRegisterAccess is responsible for inserting an event in the synchronization sequence, i.e., the fact that a thread has locked an object. Although not necessary for the characterization of the synchronization sequence, the call to endRegisterAccess inserts the opposite event, i.e., the release of an object’s lock. Such an event may be useful for synchronization sequence generation.

The replaying phase requires a different instrumentation. The first point to note is that before entering a synchronized code, the thread should consult the synchronization sequence and check whether the next event is the one to be executed. For example, if thread T1 is about to lock object O1, there must be an statement that consults the synchronization sequence to check if the next event is “T1 locks O1”. The instrumentation for replaying a synchronization sequence for Listing 5.13 is shown in Listing 5.15.
Listing 5.14: Example of recording instrumentation.

```java
public class MyClass {
    myOtherClass myObject = new myOtherClass();

    public void myMethod() {
        beginRegisterAccess(Thread.currentThread(), this);
        try {
            doSomething();
        } finally {
            endRegisterAccess(this);
        }
    }

    public void myOtherMethod() {
        Object sinc;
        synchronized (sinc = myObject) {
            beginRegisterAccess(Thread.currentThread(), sinc);
            try {
                doSomeOtherThing();
            } finally {
                endRegisterAccess(sinc);
            }
        }
    }
}
```

Listing 5.15: Example of playback instrumentation.

```java
public class MyClass {
    myOtherClass myObject = new myOtherClass();

    public void myMethod() {
        checkAccess(Thread.currentThread());
        synchronized (this) {
            nextEvent();
            doSomething();
        }
    }

    public void myOtherMethod() {
        checkAccess(Thread.currentThread());
        synchronized (myObject) {
            nextEvent();
            doSomeOtherThing();
        }
    }
}
```
Using synchronization join points, the tracing process can be encapsulated in a single aspect. As shown in Listing 5.16, the call to \textit{beginRegisterAccess} is responsible for inserting an event in the synchronization sequence, i.e., the fact that a thread has locked an object. Although not necessary for the characterization of the synchronization sequence, the call to \textit{endRegisterAccess} inserts the opposite event, i.e., the release of an object’s lock. Such an event is useful for synchronization sequence generation.

Listing 5.16: Recording with synchronized block body join point.

\begin{verbatim}
before(Object obj): synchronized_body(obj)
   && within(*.myMethod()){
      beginRegisterAccess(Thread.currentThread(), obj);
   }

after(Object obj) returning : synchronized_body(obj)
   && within(*.myMethod()){
      endRegisterAccess(obj);}

after(Object obj) throwing : synchronized_body(obj)
   && within(*.myMethod()){
      endRegisterAccess(obj);}
\end{verbatim}

The playback phase, shown in Listing 5.17, requires a synchronized block join point. The first point to note is that, before entering a synchronized code, the thread should consult the synchronization sequence and check whether the next event is the one to be executed.

Listing 5.17: Playback with synchronized block join point.

\begin{verbatim}
before(): synchronized() && within(*.myMethod()){
      checkAccess(Thread.currentThread());}

before(): synchronized_body() && within(*.myMethod()){
      nextEvent();}
\end{verbatim}

The call to \textit{checkAccess} does not use the synchronizing object to consult the synchronization sequence. This is because, considering that the thread is deterministic and followed the synchronization sequence until that point, it would not be accessing a different object. It has only to check if it is its turn to execute. The call to \textit{nextEvent} inside the block ensures that the object has been locked before the event is removed from the synchronization sequence, thereby liberating other threads to follow their execution (if possible).
CHAPTER 5. APPLICATIONS AND EVALUATION

The code presented in Listing 5.13, 5.14 and 5.15 are just simplified version of implementation for original R&P instrumentation. The UML of the full implementation for R&P are given in Appendix B.

Because coupling is already a complex software attribute in object-oriented systems (e.g., there are many different mechanisms that can constitute coupling) and there has been no attempt to provide a structured synthesis, the understanding of the state-of-the-art for aspect-oriented systems is derived from the concept in object oriented programs, as presented in Section 5.7.

The majority of the evaluation for R&P on coupling measurement in abject-oriented systems focuses on usage dependencies between classes and aspects, which can be derived from a static analysis of design documents or source code. The dynamic aspects of coupling between objects and aspects at run-time have barely been investigated and are not yet considered in practice.

With respect to the coupling metrics suggested in Section 5.7, the metrics suite includes some common coupling metrics which have been included in several existing coupling metric frameworks in our coupling metrics suite.

The evaluation results presented next compare the Java and synchronized block join point instrumented implementations for the Record and Playback case study [Del04], based on the suite of modularity metrics presented in Section 5.8. Modularity measures with respect to separation of concurrency concerns are based on concern diffusion over components, concern diffusion over operation, concern lines of code(LOC), concern diffusion over LOC [CFGF08].

5.9.1 Separation of Concerns Measures

Figure 5.7 presents an analysis of separation of concerns focusing on concurrency concerns. It shows how the Record and Playback (R&P) program was affected by the four concern measures. The concern diffusion over components and the Concern Diffusion over lines of code are evidently reduced after the code is reprogramed to use the synchronized block join point implementation, SynAJ.

In the AO version of R&P, code related to the implementation of the synchronization handling concern was moved to aspects. Therefore, for all the metrics, the number of classes implementing synchronization record was zero, instead, there are 8 classes/aspects used for playback. The reduced values for Concern Diffusion over Components in the OO and AO versions of R&P are due to the design choice of creating one synchronized block aspect for each public class.
implemented synchronization record and playback in the OO version. Other possible design choices would be to put the exception handling code in a single aspect or, for each synchronization, create an aspect that encapsulates the possible playback strategies for a given synchronization.

The AO version exhibited a better Concern Diffusion over Operations (19% lower than the OO version). For most components, the AO solution was either equivalent or superior to the OO one. The AO versions of these components had higher values in Concern Diffusion over Operations because the OO version had operations with more than one synchronized block. When these record and playback steps were moved to aspects, each one had to be put in a separate advice. Moreover, record reuse was low for these components, since they implement record step for every synchronized block. When moving record step to aspects, we synchronized block advice as much as possible.

Concern Diffusion over LOC was the metric where aspects performed best. The AO version of R&P had less than 20% of the number of concern switches of the OO version.

5.9.2 Size Measures

Experimental data presented in Figure 5.8 supports the claim that a global view of synchronized join points facilitates code reuse. Synchronizations with the same
strategy are handled similarly in most of the methods. Hence, it becomes easier for the developer to identify which synchronization can be reused.

![Figure 5.8: Size measure for Record and Playback.](image)

The 8% increase in the vocabulary size of the AO version was entirely due to the aspects. No new names were introduced or removed. Similarly to Concern Diffusion over Components, Vocabulary Size depends heavily on how the implementation of the exception handling concern is partitioned among the aspects. The number of operations in the AO version of R&P was 17% bigger than in the OO version. The main reason for this increase was the creation of advice implementing record and playback in synchronization blocks. Another reason for the increase in the Number of Operations was the refactoring of methods to expose join points that AspectJ can capture.

The reduced values for LOC were expected. Reusing synchronized aspects in R&P could reduce the lines of code in one side, but the use of AspectJ incurred a slight implementation overhead because it was necessary to specify join points of interest in order to associate synchronized block to pieces of code. In the end, the economy in LOC achieved although there is slight overhead of using AspectJ.

### 5.9.3 Coupling and Cohesion Measures

The coupling between normal implementation and re-implementation with the synchronized join point models is also reduced in R&P, as shown in Figure 5.9.
For the most part, this happens because methods using the same synchronization strategy do not need to explicitly list the synchronizations that they signal in their interfaces. Since many of these methods do not do anything with these synchronizations, the number of classes on which these methods depend will inevitably decrease.

The OO and AO versions of R&P exhibited very similar measures for the coupling metrics. The Depth of the Inheritance Tree increased by less than 3% in the AO version. This was expected, since the use of aspects alone does not interfere with this metric. The increase of 1 in the value of the measure was due to the creation of a new aspect.

Coupling between Components in the two versions was almost identical. Couplings were reduced by removing the class related to record step but also increased when aspects had to capture contextual information from synchronized blocks. In these cases, at most one coupling is reduced per synchronized block, due to a reference from the aspect to its corresponding class.

Lack of cohesion in the operations was the one for which the AO version of R&P presented the worst results. The AO version was more than 25% higher than in the OO version. This is due to the large number of operations that were created to expose join points that AspectJ can capture. These new operations are not part of the implementation of the synchronization concern (and therefore do not affect Concern Diffusion over Operations), but are a direct consequence
of using aspects to modularize this concern. Refactoring to expose join points is a common activity in aspect-oriented software development, since current aspect languages do not provide means to precisely capture every join point of interest. It is interesting to note that the goal of the Lack of Cohesion in Operations metric is to capture a partial view of cohesion: it considers only the explicit relationships between the attributes and operations. It does not consider direct inter-operation relationships and the semantic closeness between elements of a component. Moreover, even though cohesion was worse in the AO version, the aspects had very good measures for Lack of Cohesion in Operations. This happened because none of the synchronized advice accesses fields of the classes they refer to and the aspects do not define new fields.

5.10 Summary

This chapter has investigated two facets of the use of aspects for concurrent programming: their impact on flexibility and their impact on system modularity.

The novel concurrent join point models presented throughout this thesis leverage AOP techniques to promote improved separation between normal and concurrent code. It can be claimed that the use of the proposed model brings two main advantages:

1. it makes concurrent structure explicit and understandable in a localized way, without the need to examine other parts of the program; it thereby allows high quality concurrent testing and debugging tools to be applied to concurrent applications, in the domains of data race detection, coverage analysis, performance monitoring, trace analysis, concurrent noise making and network load simulation.

2. it promotes better maintainability of concurrent code by separating the parallel concerns and eliminating annoying parallel mechanism declarations such as synchronization. The implementation of the join point models SynAJ, BlockAJ and IfThenElseAJ are all available in AspectJ and also form part of the abc project.

As demonstrated throughout this chapter, aspects can provide a concurrent program with a flexible design. This flexibility is gained from the reuseability of
the concurrency aspects, which could be incorporated into many applications, to such an extent that the concurrency can be fully handled by a single aspect. It also decouples the concurrency concerns from the implementation of the parallel program. It has been shown that *around* advice for join points is not limited to performing only the `proceed()` method with given arguments, but can also address more complex and flexible behaviours, such as the `rm.proceed()` method to remove synchronization overhead.

However, this flexibility comes at a cost which depends heavily on the JVM used (because the examples are based on Java) and on the selection mechanism used. The performance of JVMs is a general problem addressed by the major vendors and virtual-machine researchers. It is not specific to aspects for high-performance computing, nor even to AOP.

In terms of system modularity, the concurrency aspects enhance program modularization by improving concurrent code reuse; and they promote better maintainability of concurrent code by separating the parallelisation concerns. The contributions concerning system modularization will be summarised in the next chapter.
Chapter 6

Conclusions

6.1 Contributions

This thesis has addressed problems of improving the programming model in concurrent programs, in order to decouple the various concerns that must be dealt with in parallel computing and has described techniques that assist the separation process. These techniques are important because for many scientific software development and maintenance tasks such as debugging and testing it is essential that code tangling in concurrent programs is avoided; however, existing tools encounter difficulties in encapsulating the concurrent concern separately from the base code.

The purpose of using concurrent programs is to perform the functional calculation and at the same time benefit from multithreaded techniques. The two primary concerns of parallel computing are the functionality, which is what the application is intended to calculate, and the arrangement for parallelisation, which means many calculations are carried out simultaneously. Using programming languages such as Fortran or Java, Section 1.3 has identified that codes for these two concerns become tangled with one another. This thesis has presented a novel step to solve the tensions between clarity of the description of computing models and details of the parallel implementation for concurrent programs.

Through techniques enabling crosscutting concerns to be separated, insights into the problems have been obtained in Chapter 2, which introduces AOP in detail. The main contribution of Chapter 2 is a survey of existing techniques that assist separating concerns, combined with an assessment of their suitability for aspect-oriented programming. As mentioned in Chapter 2, the first examples
of AOP in the literature involve encapsulating the logging concern separately from the functional concern, in scientific software applications programmed in Java. However, further developments in AOP to use AspectJ to intervene at fundamentally procedural elements in parallel computing have been suspended, leaving a gap in the software industry to embrace the language to make it more appealing for a wider audience.

On the basis of these insights, Chapter 3 described mechanisms for using AspectJ to encapsulate the parallelisation concern in an aspect to separate it from the computation code. The primary mechanism upon which AOP is based is the concept of the join point. The challenge for this thesis has been to formulate this concept as an abstraction that can make the link between the computing model and the implementation of parallelisation in concurrent programs. For concurrent programming, the concept of synchronization has been found to be crucial for preserving the integrity of resources shared. It is simple but effective and fast to use. The LoopsAJ and basic SynAJ presented in Chapter 3 implements basic loops and synchronization join points for use in aspect-oriented software development and enables the separation of concurrent concerns in AOP. However, the implemented join point models have also shown that, in general, concurrent concerns have not been fully separated owing to a lack of a support mechanism, and the use of those join point models is limited to instrument AspectJ with a more functional pointcut. This in itself contributes to the area of AOP, because there is no similar functional pointcut, despite the fact that it has not been used in real cases.

Studies have shown that separation of concurrency concerns can benefit software development and maintainers. The set of separation tools are potentially highly powerful. The group of fine-grained join point models presented in Chapter 4 makes significant improvements towards mechanisms that are more powerful and yet more usable and adds new instrumentation in AspectJ. The disadvantage of LoopsAJ and SynAJ is that they can cope only with the outer body of the join point, which can be limiting. The enhanced LoopsAJ and SynAJ have no restrictions in this respect, which makes them applicable in more complex situations. For example, when a concurrent program wants to convert to a transactional program, which has an atomic mechanism to ensure consistency when using a shared object or memory, it may simply remove the original synchronization or replace the synchronization with another type of “lock”.
CHAPTER 6. CONCLUSIONS

The contribution of this thesis lies in the design and evaluation of a set of case studies in concurrent programs, as presented in Chapter 5. In particular, the enhanced fine-grained join point models can be adapted in various concurrent situations to provide novel flexible usage and enhancement in system modularity. These case studies demonstrate that the set of fine-grained join points make an important contribution that enables advice to bypass existing synchronizations.

6.1.1 Contributions to concurrent programming

The major contribution of this thesis is enabling the AOP language to become more flexible for the possibility of using aspects for decoupling consummately the implementation of parallelisation from that of computation models. As shown throughout Chapters 3, 4, and 5, the example applications of the techniques presented in this thesis make it possible to use the same base computational components with different parallelisation aspects and the same aspects with different computational components. In such systems, the computational code that forms the base components is totally decoupled from the code implementing the parallelisation. The benefits in terms of reusability are immediate. Readability is also improved compared with implementations that are not aspect-oriented, since the computational concern and the parallelisation concern are not at all tangled. Reasoning with aspects might incur a certain learning curve, but this can be eased by the use of tools that assist the development process.

6.1.2 Contributions to aspect-oriented programming

As far as the AOP community is concerned, the thesis completes the need for, and covers the possibility of making, aspects capable of handling complex behaviours that are not necessarily limited to acting upon actions occurring at the named interface of the objects, traditional modules, or join point for loops. The various join points presented in Chapter 4 are a group that could not be targeted by any combination of AspectJ pointcuts. Although synchronization, if-then-else block, and general block are usually part of the basic programming toolkit, it is worth providing mechanisms for AspectJ such as languages to enable the handling of synchronization that requires in-depth analysis and novel usage of the base concurrent program. Providing AspectJ with the ability to intervene at block level cannot be addressed solely by enhancing pointcut expressiveness, that is,
by improving the mechanisms for expressing complex pointcuts by combining primitive pointcut descriptors. This thesis makes the case for providing aspect-oriented tools with fine-grained join points, corresponding to primitive pointcut descriptors on their own, and encompassing complex behaviours.

*SynAJ*, *SynBodyAJ*, *IfThenElseAJ*, and *BlockAJ* – the extensions to *abc* [ACH+05a] that provide AspectJ with join points for synchronization, if-then-else block, and general block, as presented in Chapter 4 – have been released under an open source licence. This both demonstrates the practical implementation realised for this thesis and gives other researchers willing to explore new join points a comprehensive example of a complex extension.

### 6.1.3 System modularity evaluation

The system modularity results obtained in Chapter 5 show that the concurrent aspects enhance program modularisation by improving concurrent code reuse; it also promotes better maintainability of concurrent code by separating the parallelisation concerns.

To calculate system modularity attributes, we selected existing object-oriented metrics and applied them to both aspect-oriented designs and object-oriented designs. These metrics were immediately applicable to the component-based object-oriented design. However, in some cases some of the metrics had to be reformulated to make them applicable to the aspect-oriented design.

This thesis has presented a far-reaching study in which aspect-oriented and object-oriented implementations are compared with several concurrent programs with respect to primary system modularity attributes. The results show that the number of operations and components decreases slightly with the use of AOP and that the overall quality of the aspect-oriented system is significantly superior at the system and component levels. The use of AOP requires fewer lines of code, helps to achieve an improved separation of concerns, and exhibits components with weaker coupling and lower internal complexity. However, a higher cohesion is a side effect in the aspect-oriented solution mainly because some aspects are not aggregating interrelated behaviours. Apart from this, architectural stability is clearly superior in the aspect-oriented architectural design of the target system.
CHAPTER 6. CONCLUSIONS

6.2 Critique

The join points presented in this thesis such as that for general blocks provide aspect-orientation with more expressiveness regarding fine-grained and complex behaviours. However, some of the practical problems related to the general block or if-then-else block selections show that having fine-grained join points is not sufficient unless these are accompanied by pointcut description mechanisms that are more expressive.

The main limitations of the concurrency aspects are caused by their reliance on the byte-code for recognising the synchronized block. This design decision has been made for the same reasons as in AspectJ, which aims to make the aspect applicable to wider code bases. The information prepared for shadow matching and weaving can be collected only at byte-code level.

Another limitation is the inability to call a specific proceed() outside the scope of a pointcut. A possible solution is to expose the aspect as a closure object, that is, a self-contained executable object (typically implementing the Runnable interface) that will be able to execute proceed with all the required execution context.

Another major disadvantage of the synchronization aspect is related to the size of the synchronized block. A synchronized block might encompass more code than is minimally required in the critical section for the safety of the synchronization. This may cause extra overhead. However, the current SBJP model is not capable of resizing the region of the synchronized block.

A final limitation relates to converting synchronized blocks into transactions. Transactional memory can be implemented in various systems, in either software or hardware, and for different purposes. Therefore, code written to work on one JVM may fail to work on another, since some of the atomic constructs use their own hardware-specific library specifications to achieve implementation.

As identified in AOP [KLM+97], pointcut expressiveness is a problem because it relies on name patterns for writing pointcuts, which is convenient but unsatisfactory. For example, matching the methods that “get” some values in a class, such as getX(int X), getY(int Y), and getXY(int X, int Y) in a Point class, can be done using a pointcut descriptor based on the regular expression “get*”; however, this mechanism relies on compliance with naming conventions for the components of an application. Even without taking foreign languages into consideration, pattern matching may have undesired side effects. The expression “get*”
might also match methods such as \texttt{getup()}, which may not be intended.

Since the general-block join point is not associated with a named signature, pointcut expressiveness becomes particularly important for selecting different blocks. For practical reasons, in particular to enable execution of test programs, the base model used in this thesis has been that of AspectJ. The weaving model of AspectJ used since version 1.1 is entirely based on the byte-code. As a result, this model works even on third-party classes for which the source code is not available. However, this constraint also implies a loss of the abstraction used by the programmer in the source code. In particular, it suggests that potential block labels or other forms of loosely coupled annotations are lost, whereas they might have helped improve the selection mechanisms if source-based weaving had been available. A source-based approach has also been investigated, but the related models and tools are not sufficiently mature to have been used as a basis for the final work. This problem may have a hybrid solution that could retain more information about the source-code as the byte-code representation (or in the intermediate representation used within the compiler), and limit the expense of the possibility of using aspects on arbitrary byte-codes. More expressiveness could also be gained by using genericity and, in particular, fine-grained genericity [KM05].

6.3 Related and future work

Although the applications presented in this thesis focus on system modularity improvement through parallelisation, the join points for synchronization, if-then-else block, and general block presented in Chapter 4 could be applied to other concerns such as performance monitoring and profiling, or to checkpoint mechanisms, which may require other types of substantial aspects. Moreover, simple timers are trivial but useful to implement using aspects.

This thesis has provided the basis for the aspect-oriented handling of synchronization at application level. However, the synchronization of an concurrent program also depends on other layers of the system. A possible next step in this line of research is to attempt to address concerns that crosscut all the elements of the system, from hardware to application, via operating system and libraries.

The general-block join point presented in this thesis could also be used to propose another level of join point selector such as join point patterns. As discussed
in [CSF+06], the join point pattern approach is used to identify higher-level join points and to decouple aspects’ definition and base-code syntax and structure. The join point patterns could be expressed based on the application design information. More precisely, a join point pattern is a template on the application behaviour identifying the join points in their context. Join points are captured when the pattern matches the portion of the application behaviour. Compared with current general-block approaches, a number of advantages can be observed. The pointcuts definition is more behavioural. In the join point pattern definition, the context of the computational flow can be identified and matched. Moreover, a graphical definition of join point patterns is more intuitive and comprehensible for programmers. Perhaps in AspectJ 2020, the UML may be able to define a fairly complex but better-understood join point?
Bibliography


[CSF06] Nélio Cacho, Cláudio Sant’Anna, Eduardo Figueiredo, Alessandro F.


Appendix A

AspectJ syntax guide

AspectJ is an aspect-oriented extension to Java. The language is fully compatible with pure Java. However, it introduces new kinds of structures and new keywords to write aspects. This appendix presents a summary of the syntax of AspectJ version 1.2.

“AspectJ adds to Java just one new concept, a join point - and that’s really just a name for an existing Java concept. It adds to Java only a few new constructs: pointcuts, advice, inter-type declarations and aspects. Pointcuts and advice dynamically affect program flow, inter-type declarations statically affect a program’s class hierarchy, and aspects encapsulate these new constructs.

A join point is a well-defined point in the program flow. A pointcut picks out certain join points and values at those points. A piece of advice is code that is executed when a join point is reached. These are the dynamic parts of AspectJ. AspectJ also has different kinds of inter-type declarations that allow the programmer to modify a program’s static structure, namely, the members of its classes and the relationship between classes.

AspectJ’s aspects are the unit of modularity for crosscutting concerns. They behave somewhat like Java classes, but may also include pointcuts, advice and inter-type declarations” [Asp].

A.1 General structure of aspects

In AspectJ, aspects are syntactically similar to Java classes. Aspects are defined via the “aspect” keyword, where “class” would have been used to define a class Java. Aspects can contain several categories of members:
• Java classes, methods, and fields, in the same way as they would be contained in a class;

• inter-type declarations (ITD) (also known as introductions) make it possible to intervene in the structure of other classes or aspects, by adding new members;

• pointcut descriptors: these can be named and are formed by combinations of conjunctions and disjunctions of pointcut expressions - including primitive pointcuts;

• pieces of advice: before, after or around pieces of advice can be considered as the aspect equivalents of methods. They do not have names, but they contain a pointcut (named or anonymous). They contain the Java instructions to execute when encountering join points matched by their pointcut; and

• declarations (which are beyond the scope of this appendix).

A.2 Inter-type declarations

Inter-type declarations make it possible to add new members (fields and methods) to classes, via an aspect. A basic example is shown in Listing A.1. Without aspect `ToStringAspect`, class `Test` does not override method `toString()` defined in `Object`. Aspect `ToStringAspect` introduces method `Test.toString()` into class `Test`. This is a modification of the structure of the class that is visible throughout the system.

A.3 Pointcut descriptors

The following is an extract from the AspectJ programming guide [Asp].

A pointcut is a program element that picks out join points and exposes data from the execution context of those join points. Pointcuts are used primarily by advice. They can be composed with boolean operators to build up other pointcuts. The primitive pointcuts and combinators provided by the language are:
Listing A.1: Inter-type declaration example.

```java
public class Test {
    int value = 10;
    public static void main(String[] args) {
        Test t = new Test();
        System.out.println("Test: "+t);
    }
}

public aspect ToStringAspect {
    public String Test.toString() {
        return Integer.toString(value);
    }
}
```

call(MethodPattern) Picks out each method call join point whose signature matches MethodPattern.

execution(MethodPattern) Picks out each method execution join point whose signature matches MethodPattern.

get(FieldPattern) Picks out each field reference join point whose signature matches FieldPattern. [Note that references to constant fields (static final fields bound to a constant string object or primitive value) are not join points, since Java requires them to be inlined.]

set(FieldPattern) Picks out each field set join point whose signature matches FieldPattern. [Note that the initializations of constant fields (static final fields where the initializer is a constant string object or primitive value) are not join points, since Java requires their references to be inlined.]

call(ConstructorPattern) Picks out each constructor call join point whose signature matches ConstructorPattern.

execution(ConstructorPattern) Picks out each constructor execution join point whose signature matches ConstructorPattern.

initialization(ConstructorPattern) Picks out each object initialization join point whose signature matches ConstructorPattern.

preinitialization(ConstructorPattern) Picks out each object pre-initialization join point whose signature matches ConstructorPattern.

staticinitialization(TypePattern) Picks out each static initializer execution join point whose signature matches TypePattern.

handler(TypePattern) Picks out each exception handler join point whose signature matches TypePattern.
adviceexecution()  Picks out all advice execution join points.

within(TypePattern)  Picks out each join point where the executing code is defined in a type matched by TypePattern.

withincode(MethodPattern)  Picks out each join point where the executing code is defined in a method whose signature matches MethodPattern.

withincode(ConstructorPattern)  Picks out each join point where the executing code is defined in a constructor whose signature matches ConstructorPattern.

cflow(Pointcut)  Picks out each join point in the control flow of any join point P picked out by Pointcut, including P itself.

cflowbelow(Pointcut)  Picks out each join point in the control flow of any join point P picked out by Pointcut, but not P itself.

this(Type or Id)  Picks out each join point where the currently executing object (the object bound to this) is an instance of Type, or of the type of the identifier Id (which must be bound in the enclosing advice or pointcut definition). Will not match any join points from static contexts.

target(Type or Id)  Picks out each join point where the target object (the object on which a call or field operation is applied to) is an instance of Type, or of the type of the identifier Id (which must be bound in the enclosing advice or pointcut definition). Will not match any calls, gets, or sets of static members.

args(Type or Id, ...)  Picks out each join point where the arguments are instances of a type of the appropriate type pattern or identifier.

PointcutId(TypePattern or Id, ...)  Picks out each join point that is picked out by the user-defined pointcut designator named by PointcutId.

if(BooleanExpression)  Picks out each join point where the boolean expression evaluates to true. The boolean expression used can only access static members, parameters exposed by the enclosing pointcut or advice, and thisJoinPoint forms. In particular, it cannot call non-static methods on the aspect or use return values or exceptions exposed by after advice.

!  Pointcut  Picks out each join point that is not picked out by Pointcut.

Pointcut0 & Pointcut1  Picks out each join point that is picked out by both Pointcut0 and Pointcut1.

Pointcut0 || Pointcut1  Picks out each join point that is picked out by either pointcuts. Pointcut0 or Pointcut1.

(  Pointcut )  Picks out each join point picked out by Pointcut.
A.3.1 Pointcut definition

Pointcuts are defined and named by the programmer with the pointcut declaration.

```java
pointcut publicIntCall(int i):
  call(public * *(int)) && args(i);
```

A named pointcut may be defined in either a class or aspect, and is treated as a member of the class or aspect where it is found. As a member, it may have an access modifier such as public or private.

```java
class C {
  pointcut publicCall(int i):
    call(public * *(int)) && args(i);
}
class D {
  pointcut myPublicCall(int i):
    C.publicCall(i) && within(SomeType);
}
```

Pointcuts that are not final may be declared abstract, and defined without a body. Abstract pointcuts may only be declared within abstract aspects. In such a case, an extending aspect may override the abstract pointcut.

For completeness, a pointcut with a declaration may be declared final.

Though named pointcut declarations appear somewhat like method declarations, and can be overridden in subaspects, they cannot be overloaded. It is an error for two pointcuts to be named with the same name in the same class or aspect declaration.

The scope of a named pointcut is the enclosing class declaration. This is different than the scope of other members; the scope of other members is the enclosing class body. This means that the following code is legal:
aspect B extends A {
    pointcut publicCall(int i): call(public Foo.m(int)) && args(i);
}

aspect B percflow(publicCall()) {
    pointcut publicCall(): call(public Foo.m(int));
}

A.3.2 Context exposure

Pointcuts have an interface; they expose some parts of the execution context of the join points they pick out. For example, the PublicIntCall above exposes the first argument from the receptions of all public unary integer methods. This context is exposed by providing typed formal parameters to named pointcuts and advice, like the formal parameters of a Java method. These formal parameters are bound by name matching.

On the right-hand side of advice or pointcut declarations, in certain pointcut designators, a Java identifier is allowed in place of a type or collection of types. The pointcut designators that allow this are this, target, and args. In all such cases, using an identifier rather than a type does two things. First, it selects join points as based on the type of the formal parameter. So the pointcut

pointcut intArg(int i): args(i);

picks out join points where an int (or a byte, short, or char; anything assignable to an int) is being passed as an argument. Second, though, it makes the value of that argument available to the enclosing advice or pointcut.

Values can be exposed from named pointcuts as well, so is a legal way to pick out all calls to public methods accepting an int argument, and exposing that argument.

There is one special case for this kind of exposure. Exposing an argument of type Object will also match primitive typed arguments, and expose a “boxed” version of the primitive. So, pointcut will pick out all unary methods that take, as their only argument, subtypes of Object (i.e., not primitive types like int), but will pick out all unary methods that take any argument: And if the argument was an int, then the value passed to advice will be of type java.lang.Integer. The “boxing” of the primitive value is based on the original primitive type.
So in the following program The pointcut will match and expose the integer argument, but it will expose it as an `Integer`, not a `Long`.

### A.3.3 Primitive pointcuts

**Method-related pointcuts**

AspectJ provides two primitive pointcut designators designed to capture method call and execution join points.

- `call(MethodPattern)`
- `execution(MethodPattern)`

**Field-related pointcuts**

AspectJ provides two primitive pointcut designators designed to capture field reference and set join points:

- `get(FieldPattern)`
- `set(FieldPattern)`

All set join points are treated as having one argument, the value the field is being set to, so at a set join point, that value can be accessed with an `args` pointcut. So an aspect guarding a static integer variable `x` declared in type `T` might be written as

**Object creation-related pointcuts**

AspectJ provides primitive pointcut designators designed to capture the initializer execution join points of objects.

pointcut publicCall(Object o): call(public *(*(..))) \&\& args(o);
public class InstanceOf {
    public static void main(String[] args) {
        doInt(5);
    }
    static void doInt(int i) {} } }

aspect IntToLong {
    pointcut el(long l) :
        execution(* doInt(..)) && args(l);
    before(Object o) : el(o) {
        System.out.println(o.getClass());
    }
}

aspect GuardedX {
    static final int MAX_CHANGE = 100;
    before(int newval): set(static int T.x) && args(newval) {
        if (Math.abs(newval - T.x) > MAX_CHANGE)
            throw new RuntimeException();
    }
}

• call(ConstructorPattern)
• execution(ConstructorPattern)
• initialization(ConstructorPattern)
• preinitialization(ConstructorPattern)

Class initialization-related pointcuts

AspectJ provides one primitive pointcut designator to pick out static initializer execution join points.

• staticinitialization(TypePattern)

Exception handler execution-related pointcuts

AspectJ provides one primitive pointcut designator to capture execution of exception handlers:

• handler(TypePattern)
All handler join points are treated as having one argument, the value of the exception being handled. That value can be accessed with an args pointcut. So an aspect used to put FooException objects into some normal form before they are handled could be written as

```java
aspect NormalizeFooException {
  before(FooException e): handler(FooException) && args(e) {
    e.normalize();
  }
}
```

Advice execution-related pointcuts

AspectJ provides one primitive pointcut designator to capture execution of advice

- `adviceexecution()`

This can be used, for example, to filter out any join point in the control flow of advice from a particular aspect.

```java
aspect TraceStuff {
  pointcut myAdvice(): adviceexecution() && within(TraceStuff);
  before(): call(* *(..)) && !cfow(myAdvice) {
    // do something
  }
}
```

State-based pointcuts

Many concerns cut across the dynamic times when an object of a particular type is executing, being operated on, or being passed around. AspectJ provides primitive pointcuts that capture join points at these times. These pointcuts use the dynamic types of their objects to pick out join points. They may also be used to expose the objects used for discrimination.

- `this(Type or Id)`
- `target(Type or Id)`
The this pointcut picks out each join point where the currently executing object (the object bound to this) is an instance of a particular type. The target pointcut picks out each join point where the target object (the object on which a method is called or a field is accessed) is an instance of a particular type. Note that target should be understood to be the object the current join point is transferring control to. This means that the target object is the same as the current object at a method execution join point, for example, but may be different at a method call join point.

- args(Type or Id or ‘‘..”, ..)
- target(Type or Id)

The args pointcut picks out each join point where the arguments are instances of some types. Each element in the comma-separated list is one of four things. If it is a type name, then the argument in that position must be an instance of that type. If it is an identifier, then that identifier must be bound in the enclosing advice or pointcut declaration, and so the argument in that position must be an instance of the type of the identifier (or of any type if the identifier is typed to Object). If it is the “*” wildcard, then any argument will match, and if it is the special wildcard “..”, then any number of arguments will match, just like in signature patterns. So the pointcut

- args(int, .., String)

will pick out all join points where the first argument is an int and the last is a String.

Control flow-based pointcuts

Some concerns cut across the control flow of the program. The cflow and cflowbelow primitive pointcut designators capture join points based on control flow.

- cflow(Pointcut)
- cflowbelow(Pointcut)

The cflow pointcut picks out all join points that occur between entry and exit of each join point P picked out by Pointcut, including P itself. Hence, it picks out the join points in the control flow of the join points picked out by Pointcut.
The cflowbelow pointcut picks out all join points that occur between entry and exit of each join point \( P \) picked out by Pointcut, but not including \( P \) itself. Hence, it picks out the join points below the control flow of the join points picked out by Pointcut.

**Context exposure from control flows**  The \( cflow \) and \( cflowbelow \) pointcuts may expose context state through enclosed \textit{this}, \textit{target}, and \textit{args} pointcuts. Anytime such state is accessed, it is accessed through the most recent control flow that matched. So the “current arg” that would be printed by the following program is zero, even though it is in many control flows.

```java
class Test {
    public static void main(String[] args) {
        fact(5);
    }
    static int fact(int x) {
        if (x == 0) {
            System.err.println("bottoming out");
            return 1;
        } else return x * fact(x - 1);
    }
}
aspect A {
    pointcut entry(int i): call(int fact(int)) && args(i);
    pointcut writing(): call(void println(String)) && ! within(A);
    before(int i): writing() && cflow(entry(i)) {
        System.err.println("Current arg is " + i);
    }
}
```

It is an error to expose such state through negated control flow pointcuts, such as within !cflowbelow\((P)\).

**Program text-based pointcuts**

While many concerns cut across the runtime structure of the program, some must deal with the lexical structure. AspectJ allows aspects to pick out join points based on where their associated code is defined.

- \texttt{within}(TypePattern)
- \texttt{withincode}(MethodPattern)
• `withincode(ConstructorPattern)`

The `within` pointcut picks out each join point where the code executing is defined in the declaration of one of the types in `TypePattern`. This includes the class initialization, object initialization, and method and constructor execution join points for the type, as well as any join points associated with the statements and expressions of the type. It also includes any join points that are associated with code in a type’s nested types, and that type’s default constructor, if there is one.

The `withincode` pointcuts picks out each join point where the code executing is defined in the declaration of a particular method or constructor. This includes the method or constructor execution join point as well as any join points associated with the statements and expressions of the method or constructor. It also includes any join points that are associated with code in a method or constructor’s local or anonymous types.

Expression-based pointcuts

• `if(BooleanExpression)`

The `if` pointcut picks out join points based on a dynamic property. It’s syntax takes an expression, which must evaluate to a boolean true or false. Within this expression, the `thisJoinPoint` object is available. So one (extremely inefficient) way of picking out all call join points would be to use the pointcut

• `if(thisJoinPoint.getKind().equals('call'))`

Note that the order of evaluation for pointcut expression components at a join point is undefined. Writing if pointcuts that have side-effects is considered bad style and may also lead to potentially confusing or even changing behavior with regard to when or if the test code will run.

Signatures

One very important property of a join point is its signature, which is used by many of AspectJ’s pointcut designators to select particular join points.

Methods

Join points associated with methods typically have method signatures, consisting of a method name, parameter types, return type, the types of the declared
(checked) exceptions, and some type that the method could be called on (below called the “qualifying type”).

At a method call join point, the signature is a method signature whose qualifying type is the static type used to access the method. This means that the signature for the join point created from the call `((Integer)i).toString()` is different than that for the call `((Object)i).toString()`, even if i is the same variable.

At a method execution join point, the signature is a method signature whose qualifying type is the declaring type of the method.

**Fields**

Join points associated with fields typically have field signatures, consisting of a field name and a field type. A field reference join point has such a signature, and no parameters. A field set join point has such a signature, but has a single parameter whose type is the same as the field type.

**Constructors**

Join points associated with constructors typically have constructor signatures, consisting of a parameter types, the types of the declared (checked) exceptions, and the declaring type.

At a constructor call join point, the signature is the constructor signature of the called constructor. At a constructor execution join point, the signature is the constructor signature of the currently executing constructor.

At object initialization and pre-initialization join points, the signature is the constructor signature for the constructor that started this initialization: the first constructor entered during this type’s initialization of this object.

**Others**

At a handler execution join point, the signature is composed of the exception type that the handler handles. At an advice execution join point, the signature is composed of the aspect type, the parameter types of the advice, the return type (void for all but around advice) and the types of the declared (checked) exceptions.
A.3.4 Matching

The withincode, call, execution, get, and set primitive pointcut designators all use signature patterns to determine the join points they describe. A signature pattern is an abstract description of one or more join-point signatures. Signature patterns are intended to match very closely the same kind of things one would write when declaring individual members and constructors.

Method declarations in Java include method names, method parameters, return types, modifiers like static or private, and throws clauses, while constructor declarations omit the return type and replace the method name with the class name. The start of a particular method declaration, in class Test, for example, might be

```java
class C {
    public final void foo() throws ArrayOutOfBoundsException { ... }
}
```

In AspectJ, method signature patterns have all these, but most elements can be replaced by wildcards. So

- `call(public final void C.foo() throws ArrayOutOfBoundsException)` picks out call join points to that method, and the pointcut

- `call(public final void *.*() throws ArrayOutOfBoundsException)` picks out all call join points to methods, regardless of their name name or which class they are defined on, so long as they take no arguments, return no value, are both public and final, and are declared to throw ArrayOutOfBoundsException exceptions.

The defining type name, if not present, defaults to *, so another way of writing that pointcut would be

- `call(public final void *() throws ArrayOutOfBoundsException)`

Formal parameter lists can use the wildcard .. to indicate zero or more arguments, so

- `execution(void m(..))`
picks out execution join points for void methods named m, of any number of arguments, while

- \texttt{execution(void m(\ldots, \text{int}))}

picks out execution join points for void methods named m whose last parameter is of type int.

The modifiers also form part of the signature pattern. If an AspectJ signature pattern should match methods without a particular modifier, such as all non-public methods, the appropriate modifier should be negated with the ! operator. So,

- \texttt{withincode(!public void foo())}

picks out all join points associated with code in null non-public void methods named foo, while

- \texttt{withincode(void foo())}

picks out all join points associated with code in null void methods named \texttt{foo}, regardless of access modifier.

Method names may contain the * wildcard, indicating any number of characters in the method name. So

- \texttt{call(int *())}

picks out all call join points to int methods regardless of name, but

- \texttt{call(int get*())}

picks out all call join points to int methods where the method name starts with the characters “get”.

AspectJ uses the new keyword for constructor signature patterns rather than using a particular class name. So the execution join points of private null constructor of a class \texttt{C} defined to throw an \texttt{ArithmeticException} can be picked out with

- \texttt{execution(private C.new() throws ArithmeticException)}
Matching based on the declaring type

The signature-matching pointcuts all specify a declaring type, but the meaning varies slightly for each join point signature, in line with Java semantics. When matching for pointcuts withincode, get, and set, the declaring type is the class that contains the declaration.

When matching method-call join points, the declaring type is the static type used to access the method. A common mistake is to specify a declaring type for the call pointcut that is a subtype of the originally-declaring type. For example, given the class

```java
class Service implements Runnable {
    public void run() { ... }
}
```

the following pointcut `call(void Service.run())` would fail to pick out the join point for the code `((Runnable) new Service()).run()`. Specifying the originally-declaring type is correct, but would pick out any such call (here, calls to the `run()` method of any `Runnable`). In this situation, consider instead picking out the target type: `call(void run()) && target(Service)`. When matching method-execution join points, if the execution pointcut method signature specifies a declaring type, the pointcut will only match methods declared in that type, or methods that override methods declared in or inherited by that type. So the pointcut `execution(public void Middle.*())` picks out all method executions for public methods returning void and having no arguments that are either declared in, or inherited by, Middle, even if those methods are overridden in a subclass of Middle. So the pointcut would pick out the `methodexecution` join point for `Sub.m()` in this code:

```java
class Super {
    protected void m() { ... }
}
class Middle extends Super {
}
class Sub extends Middle {
    public void m() { ... }
}
```
A.4 Advice

Advice defines pieces of aspect implementation that execute at well-defined points in the execution of the program. Those points can be given either by named pointcuts (like the ones you’ve seen above) or by anonymous pointcuts. Here is an example of an advice on a named pointcut:

```java
pointcut setter(Point p1, int newval): target(p1) && args(newval)
    (call(void setX(int)) || call(void setY(int)));

before(Point p1, int newval): setter(p1, newval) {
    System.out.println("About to set something in " + p1 + " to the new value " + newval);
}
```

And here is exactly the same example, but using an anonymous pointcut:

```java
before(Point p1, int newval): target(p1) && args(newval)
    (call(void setX(int)) || call(void setY(int))) {
    System.out.println("About to set something in " + p1 + " to the new value " + newval);
}
```

Here are examples of the different advice:

This after advice runs just after each join point picked out by the (anonymous) pointcut, regardless of whether it returns normally or throws an exception:

```java
after(Point p, int x): target(p) && args(x) && call(void setX(int)) {
    if (!p.assertX(x)) throw new PostConditionViolation();
}
```

This after throwing advice runs just after each join point picked out by the (anonymous) pointcut, but only when it throws an exception of type `Exception`. Here the exception value can be accessed with the name `e`. The advice re-raises the exception after it’s done.

This after returning advice runs just after each join point picked out by the (anonymous) pointcut, but only if it returns normally. The return value can be accessed, and is named `x` here. After the advice runs, the return value is returned:
This around advice traps the execution of the join point; it runs instead of the join point. The original action associated with the join point can be invoked through the special proceed call:

```java
void around(Point p, int x): target(p) && args(x) && call(void setX(int)) {
    if (p.assertX(x)) proceed(p, x);
    p.releaseResources();
}
```
Appendix B

UML for Original R&P

Figure B.1: Package for Original R&P Instrumentation