A Chorded Compiler for Java

Fourth Year MEng Project Report

MEng Student:  Tim Wood

Supervisor:  Susan Eisenbach

2nd Marker:  Professor Jeff Magee

Address:  Department of Computing

180 Queen’s Gate

South Kensington Campus

Imperial College London

SW7 2AZ

Email:  tw00@doc.ic.ac.uk, tim@flamingpenguin.co.uk

Copyright © Tim Wood, 2004. All rights reserved.
Abstract

Polyphonic $C^\#$ is a language with modern concurrency primitives based on the Join-Calculus. We design and implement a compiler which adds Polyphonic $C^\#$ like extensions (called chords) to Java.

Our compiler uses Antlr, the parser-generator, and Velocity, the Apache project’s template engine for Java.

We develop a formal description of chorded-languages, and use this description to create a new algorithm for implementing chords. Our compiler provides both this algorithm and the Polyphonic $C^\#$ algorithm.

We prove that our new algorithm can find join-pattern matches in a single pass, whereas the Polyphonic $C^\#$ algorithm needs two passes in some situations — however, our new algorithm is worse in some other aspects.

We show how the formal description we have created can be used to reason about chorded-programs, and show that automated analysis of chorded programs might be possible by composing finite state transition graphs.

We discuss distributed programming with chords and RMI, and implement a distributed solution to the dining philosophers problem.
Acknowledgements

Many thanks to my supervisor Susan Eisenbach, who tolerated both my tendency to run off in random directions on a whim, and the fact that my eventual project bares hardly any resemblance to my original proposal. Her suggestion to look at Polyphonic C♯ set the project off along the road it took, and her help and advice through-out has been invaluable.

Professor Jeff Magee, my second marker, also provided invaluable input and insight. At a key stage in the project, it was his suggestion to change course from a concurrency-teaching tool to an implementation of chords for Java.

Matthew Sackman, Jacob Bower, Emma Crowe, Thomas Geike, Guido Arenstedt, James William Ascroft-Leigh and Paul Jolly for their time, conversation and inspiration. Honeycube and Motelhero for distracting me enough, but not too much, during the final few weeks of this project. Matthew Sackman generously proof-read much of the report (any mistakes that remain are mine), Paul Jolly helped out with some of my \LaTeX{} problems, and Mark Thomas showed me how to get “C♯” to appear properly.
# Contents

1 Introduction and Overview ....................................................... 1
   1.1 Background and Context .................................................. 1
   1.2 Objectives ........................................................................ 1
   1.3 Contributions .................................................................... 2
   1.4 Reading this Document ...................................................... 2

2 Concurrent programming background ........................................... 5
   2.1 Introduction ....................................................................... 5
   2.2 The Java Approach ........................................................... 5
      2.2.1 Threads in Java ......................................................... 5
      2.2.2 Locking and Mutual exclusion .................................... 7
      2.2.3 JSR166 [18] ............................................................. 8
   2.3 Ada Tasks [4] ..................................................................... 10
      2.3.1 Example of an Ada task [31] ....................................... 10
      2.3.2 Starting a task ......................................................... 11
3.4 Chord Programming Primer ........................................... 20
3.4.1 Starting a Thread .................................................. 21
3.4.2 Locking ................................................................. 21
3.4.3 Bounded Data Structures .......................................... 22

3.5 Chord Programming Idioms ........................................... 23
3.5.1 List of Idioms ......................................................... 23
3.5.2 The notion of state .................................................. 23
3.5.3 Initial state ............................................................ 25
3.5.4 Read/Write locks ..................................................... 25

3.6 Active Objects [17] ...................................................... 27
3.6.1 Polyphonic C♯ implementation [1] ............................... 27
3.6.2 Java Implementation ................................................ 28
3.6.3 Comparison ............................................................ 31

3.7 Conclusions ............................................................... 31

4 Programs, Classes and Objects ........................................ 33
4.1 Introduction ............................................................... 33
4.2 Program ................................................................. 33
4.3 Class ................................................................. 35
4.3.1 Ordering on TypeName ............................................. 36
4.4 Channels ............................................................... 36
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.4.1</td>
<td>Channel ordering</td>
<td>37</td>
</tr>
<tr>
<td>4.4.2</td>
<td>Indexing channels and bodies</td>
<td>37</td>
</tr>
<tr>
<td>4.5</td>
<td>Bodies</td>
<td>38</td>
</tr>
<tr>
<td>4.6</td>
<td>Inheritance</td>
<td>38</td>
</tr>
<tr>
<td>4.7</td>
<td>Operations on Code</td>
<td>38</td>
</tr>
<tr>
<td>4.7.1</td>
<td>Counting calls</td>
<td>39</td>
</tr>
<tr>
<td>4.8</td>
<td>Channel Protection</td>
<td>39</td>
</tr>
<tr>
<td>4.8.1</td>
<td>Visibility</td>
<td>39</td>
</tr>
<tr>
<td>4.9</td>
<td>Well Formedness</td>
<td>40</td>
</tr>
<tr>
<td>4.9.1</td>
<td>Patterns</td>
<td>40</td>
</tr>
<tr>
<td>4.9.2</td>
<td>Called Channels</td>
<td>40</td>
</tr>
<tr>
<td>4.9.3</td>
<td>Program Formedness Rules</td>
<td>40</td>
</tr>
<tr>
<td>4.10</td>
<td>Object</td>
<td>40</td>
</tr>
<tr>
<td>4.10.1</td>
<td>Mapping channels to queues</td>
<td>41</td>
</tr>
<tr>
<td>4.10.2</td>
<td>mQueue</td>
<td>41</td>
</tr>
<tr>
<td>4.10.3</td>
<td>Notation for queues</td>
<td>41</td>
</tr>
<tr>
<td>4.11</td>
<td>Object State</td>
<td>42</td>
</tr>
<tr>
<td>4.11.1</td>
<td>Message arrival</td>
<td>43</td>
</tr>
<tr>
<td>4.11.2</td>
<td>Chord dispatch</td>
<td>43</td>
</tr>
<tr>
<td>4.12</td>
<td>System State</td>
<td>43</td>
</tr>
<tr>
<td>4.13</td>
<td>Conclusion</td>
<td>44</td>
</tr>
</tbody>
</table>
5 Implementation Overview

5.1 Introduction................................................................. 45

5.2 Implementation Strategy............................................... 45

5.3 Objectives................................................................. 46

5.4 Target Java Version..................................................... 46

5.5 Translating Java.......................................................... 46

5.6 Parsing the chorded-Java................................................ 47

5.6.1 Selecting a parser-generator........................................ 47

5.6.2 Modifying the Grammar............................................. 48

5.6.3 Objectives met.......................................................... 51

5.7 Tree translation with Antlr............................................. 51

5.7.1 Collecting Information............................................... 51

5.7.2 Collecting the information........................................ 53

5.7.3 Creating Nodes........................................................ 54

5.8 Tree printing with Antlr................................................ 55

5.9 Type checking.............................................................. 56

5.10 Using Velocity to generate the auxiliary code..................... 57

5.10.1 What is Velocity?.................................................... 57

5.10.2 Alternatives........................................................... 57

5.10.3 How velocity solves this problem................................. 59
5.10.4 Velocity Version ................................................. 61
5.10.5 Objectives met .................................................... 61
5.11 Conclusions .......................................................... 63

6 Implementation Issues .................................................. 65

6.1 Introduction ......................................................... 65
6.2 Implementation Overview ........................................... 65
6.3 Implementing message queues in Java .......................... 66
  6.3.1 Null message queues ............................................. 66
  6.3.2 Null message queue complexity ................................. 66
  6.3.3 Non-null message queue ........................................... 67
  6.3.4 Non-null message queue complexity .......................... 69
6.4 Marshalling message arrival – implementing channels ........ 69
  6.4.1 Marshalling async channels ...................................... 70
  6.4.2 Marshalling sync channels ......................................... 70
6.5 Pattern Matching — identifying dispatchable chords ........... 73
  6.5.1 Atomicity of dispatch ............................................. 74
  6.5.2 Fairness ............................................................ 74
  6.5.3 Determinism ....................................................... 75
6.6 Encoding “Or” using “And” ............................................. 75
  6.6.1 Typing and naming in disjunction patterns ..................... 75

xii
6.7 Efficient encoding of subset operations ........................................ 76

6.7.1 Why bit-vectors .............................................................. 77

6.7.2 Mapping sets to bit-vectors ............................................... 77

6.7.3 Object Pattern Vectors .................................................... 78

6.7.4 Implementing Bitvectors in Java ........................................ 78

6.8 Implementing the pattern matching algorithm ............................ 79

6.8.1 Selecting a match ........................................................... 79

6.8.2 Maintaining the Vector ................................................... 79

6.9 Conclusions ........................................................................ 81

7 Dispatch and synchronisation .................................................. 83

7.1 Introduction ...................................................................... 83

7.2 Simple Algorithm ................................................................ 83

7.2.1 Auxiliary Functions ....................................................... 85

7.2.2 One run is sufficient ....................................................... 85

7.2.3 No matches can only be followed by single dispatch .......... 85

7.2.4 There is never a match before a message arrival ............... 86

7.2.5 Complexity ................................................................... 87

7.2.6 Implementing the algorithm using threads ......................... 87

7.2.7 Memory Barriers ............................................................ 89

7.2.8 Optimising the Algorithm ............................................... 90
9 Reasoning about chords

9.1 Introduction .................................................. 105

9.2 Conjectures .................................................. 105

9.3 Transition Graphs .............................................. 106
  9.3.1 Constructing transition graphs ......................... 106

9.4 A simple lock ................................................. 107
  9.4.1 Formal Description of E ................................. 108
  9.4.2 Working out the transition graph of E ................ 109
  9.4.3 Showing the properties ................................ 110
  9.4.4 Accuracy ................................................ 110
  9.4.5 Liveness ................................................ 111

9.5 Detecting unreachable code ................................. 111
  9.5.1 Formal Description of F ................................. 112
  9.5.2 Initial States ............................................ 112
  9.5.3 Public Channels ......................................... 112
  9.5.4 Working out the transition graph of F ................ 113
  9.5.5 Finding the problem .................................. 113
  9.5.6 Accuracy and safety .................................. 114

9.6 Properties of infinite graphs ............................... 114
  9.6.1 Formal Description of G ............................... 115
E  γ augmentation 147

E.1 Syntactic Sugar .................................................. 147

E.2 Justification of γ augmentation .............................. 148

E.2.1 ⊔ ................................................................. 149

E.2.2 − ................................................................. 150

E.3 Usage of γ augmentation ....................................... 150

E.3.1 Implementing calledChannels ............................... 150

E.3.2 The analysis ..................................................... 151

F  Notes on Notation 153

F.1 Types ............................................................... 153

F.1.1 boolean ......................................................... 153

F.1.2 Infinite numbers ............................................... 153

G  Scenebeans 155

G.1 The pre-processor ............................................... 155

G.2 The macro-expander ............................................ 156

G.2.1 Variables ....................................................... 156

G.2.2 Multiple dereferences ....................................... 157

G.2.3 List size operation ........................................... 157

G.2.4 The re-write .................................................... 157

G.3 Updating from AWT to Swing ............................... 158

xviii
# List of Figures

1.1 Distributed dining philosophers ............................................. 3
4.1 Overview of a single class system ........................................... 34
5.1 Overview of software design .................................................... 47
5.2 Overview of the model API ..................................................... 53
5.3 Overview of implementation plugability .................................... 62
6.1 Message Arrival ................................................................. 68
6.2 Chord Dispatch ................................................................. 68
7.1 Flow-chart overview of simple algorithm .................................. 89
7.2 Flow-chart overview of polyphonic algorithm ............................ 93
8.1 RMI object reference proxy .................................................... 95
8.2 RMI Serialisation ............................................................... 96
8.3 Distributed dining philosophers ............................................. 98
8.4 Philosophic UML .............................................................. 99
9.1 Transition diagram of a lock .................................................. 110
9.2 Transition diagram of a lock .................................................. 114
9.3 Infinite Transition diagram ........................................... 118
9.4 Macro States of infinite graphs ...................................... 120
B.1 Reducing a graph using strong components ..................... 141
G.1 The Macro Expander ................................................... 158
G.2 The Original AWT Canvas ........................................... 159
G.3 The updated AWT Canvas, and Swing Panel ...................... 161
G.4 Network Visualisation API ........................................... 161
G.5 Nodes and links can easily be configured ......................... 162
G.6 A message being sent .................................................. 162
G.7 Connections can change colour ..................................... 162
G.8 Nodes can change colour and carry a status label .............. 163
Chapter 1

Introduction and Overview

A first principle not formally recognized by scientific methodologists: when you run into something interesting, drop everything else and study it. — B.F. Skinner

1.1 Background and Context

Multi-threaded programming is difficult. Even with modern language support for threads, for example in C♯ and Java, mistakes are easy to make. Concurrent programming in shared memory systems uses different models to concurrent programming in distributed systems; which adds to the problems.

Polyphonic C♯[1] is an extension of C♯. It adds high-level synchronisation primitives called chords. Chords are a join style synchronisation mechanism, heavily influenced by the Join Calculus[7]. The current implementation mechanism is a naive translation from Polyphonic C♯ into C♯; although it is detailed in several papers the compiler is not publicly available. Recently the polyphonic extensions to C♯ have been slated for inclusion in a new language (also based on C♯) called Cω. No compiler is yet publicly available for Cω.

The Join-Calculus does not map exactly to chorded-programming, and is in itself a programming language; rather than a formal description.

1.2 Objectives

- We hope to describe exactly what chorded-programming is and formally describe the properties of chorded programming.
- We will then use this description, and the published literature, to develop a compiler for a chorded variation of Java.
  We choose Java because it is a popular programming language in wide spread use and active development. It is also a fairly open standard and has a considerable body of research literature available.
- We hope to provide alternative algorithms for implementing chords, and use our formal description to show that our implementation of chords is correct.
• We will try to use our formal description framework to show how to reason about some properties of simple chord-programs, and to show that we can keep an analysis of charded programs finite. We hope that this work could lead to the creation of a compiler which can analyse and optimise charded-programs.

• We will also look briefly at how distribution effects our implementation and framework; including how to use chords with RMI, how the compiler can support the programmer in using this combination, and what effect charded programming may have on RMI.

1.3 Contributions

• The first publicly available charded compiler, which translates from a charded-Java variant into Java. See chapter 5.
  – An implementation of the polyphonic chord-implementation, which sometimes has to scan for matches twice on message arrival — but sometimes can dispatch a chord without a context switch. See chapter 7.
  – A new, simpler, algorithm for implementing chords, which we prove only needs to scan for matches once on each message arrival — but which sometimes context switches when the polyphonic algorithm would not. See chapter 7.

• A formal description of charded programs, and documentation of how that formal description maps to our implementation algorithm. See chapters 4 and 6.

• A discussion of provable properties of charded-programming, a demonstration that analysis of the states of a charded object can be preformed over a finite graph, and several walk throughs of reasoning on small charded-programs. See chapter 9.

• Support for RMI in our charded compiler, described in chapter 8.

• An implementation of a solution to the dining philosophers problem which runs distributed across several machines. See chapter 8, the solution is depicted running in figure 1.1.

1.4 Reading this Document

The first part of the document covers language support for multi-threading (chapter 2), what exactly the Join-Calculus and Polyphonic C♯ are (chapter 3), and how charded-programming can be formally described (chapter 4).

The second part covers the implementation of the compiler (chapter 5), including the use of Antlr and Velocity. Then it goes on to cover how the compiler implements charded-objects in Java (chapter 6) and the two implementation algorithms (chapter 7).

The third part covers distributed-programming with chords and RMI (chapter 8), and finally how to use our formalism to reason about charded-programming (chapter 9).

The appendixes cover some addition implementation information, several pieces of maths supporting the main sections, and a description of some improvements made to a library called scenebeans during the course of the project.
Figure 1.1: Distributed dining philosophers
Chapter 2

Concurrent programming background

We learn what we have said from those who listen to our speaking. — Kenneth Patton.

2.1 Introduction

There are several approaches to concurrent programming already in use. In this chapter we examine some of those systems and evaluate their benefits.

We look at the Java programming model and discuss some of its subtleties, describing Java locking and mutual exclusion. We talk very briefly about the Java memory model, and look in more depth at the JSR166 concurrency library.

Ada tasks and data protection are covered as an alternate approach to language design.

The Ruby language also provides language support for threading, we look at how it provides shared data and mutual exclusion.

2.2 The Java Approach

Java uses an approach to concurrency called Threads\[11\] (hence multi-threading, a program which contains multiple threads of execution).

2.2.1 Threads in Java

A lot has been written on Threading in Java, [27], [16]. Threads are represented by the class Thread, and are started by creating an instance of thread and telling it to run. Here is a simple example:

```java
final Thread thread = new Thread(new Runnable ()
{
    public void run ()
    {
        someObject . someComplexOperation () ;
    }
});
```
Unfortunately, this code is a little difficult to understand. Here it is necessary to understand objects and classes, methods, scope, inner-classes, anonymous inner-classes, inheritance and interfaces; just to be able to work out that a thread is being started in order to run “someObject.someComplexOperation()”. This isn’t ideal for teaching, learning, nor maintenance — which often depends on the legibility of the code[3]. The example could be somewhat simplified by writing it in a more verbose manner, but it is still not easy to explain.

There are also subtleties to the Java multi-threading system; for instance, threads should not be started in the constructor of an object. If they are (which is allowed by the language and runtime) then the finalness of object members may not be correctly presented. Another example is the immutability (or not[26]) of strings.

### 2.2.1.1 Java Threads and Strings

A String in Java has several items of data associated with it. As an example, to save memory, Java strings can consist of a pointer to an array of characters and an offset into the array.

```java
public class String {
    final char[] m_data;
    final int m_offset;
    public String(char[] data, int offset) {
        m_data = data;
        m_offset = offset;
    }
    [...]
}
```

Listing 2.2: Simplified Java String

Assume data contains data="/home/tim/tmp"; if the memory write at lines 8 and 9 are not seen consistently (for instance when being flushed from a multi-processors cache) then we could have a problem.

```java
char[] data = "/home/tim/tmp".toCharArray();
String str = new String(data, 9);
passToSomeOtherThread(str);
```

Listing 2.3: Simplified Java String Use

The second thread should see str which contains the String "/tmp". However, if the write at line 9 has not been flushed to main memory when the second thread reads the string it will find that m_offset is still equal to its initial value (that is 0).
This is a severe problem – if the second thread is doing a remove all on the directory that it
is passed, then the wrong data will be deleted (/home/tim/tmp, instead of /tmp)! The JSR133
proposes an extension to the semantics of final which should fix this problem.

2.2.2 Locking and Mutual exclusion

Java locking is done on a per-object basis. Each Object has its own lock, which can be obtained
by one thread at a time. Not having the lock will stop a thread from entering any critical sec-
tions dependent on the lock; this can be used to protect the data in an object from inconsistent
modification and reading. The object can be considered as a monitor. Synchronisation is also the
mechanism for propagating memory changes between threads in the Java Memory Model[11].

In this example, threads cannot enter the synchronised methods unless they first obtain a lock on
the object. Only one thread at a time can obtain the lock — threads without the lock are queued
by the Java runtime until the lock becomes available.

```java
public class ConsistentObject
{
    private int m_x = 2;
    private int m_y = 3;
    private int m_z = 4;

    public synchronized void updateX(final int x)
    {
        m_x = x;
        m_z = m_x * m_y;
    }

    public synchronized void updateY(final int y)
    {
        m_y = y;
        m_z = m_x * m_y;
    }

    public synchronized int z()
    {
        return m_z;
    }

    public synchronized int y()
    {
        return m_y;
    }

    public synchronized int x()
    {
        return m_x;
    }
}
```

Listing 2.4: A Thread-Safe Object
You may notice that we have synchronised the read methods as well as the ones that do the updates. This may seem unnecessary at first, however it is the only way to make sure changes in the variables are immediately and consistently seen by the other threads (if the readers were not synchronised then the results written by one thread may not be seen by other threads for an arbitrary length of time). Synchronised here can be understood in terms of memory barriers.

Java provides another syntax for obtaining locks. For example, if one wants to read the values of x,y and z consistently – ensuring they are not updated between method calls – then one must obtain and release the lock “manually”.

```java
public int someMethod(final ConsistentObject theObject) {
    int polynomial;
    synchronized (theObject) {
        polynomial = theObject.x() * theObject.y() * theObject.z();
    }
    return polynomial;
}
```

Listing 2.5: Inline Method Synchronisation

This code gets the lock before the calculation, performs the calculation, then releases the lock before returning the result (it could be simplified, but we wanted to illustrate that code can be placed before and after the synchronised block). This synchronised also has effects on the memory view between different threads.

### 2.2.3 JSR166 [18]

JSR166 is the concurrency library and support scheduled to be added to JSR176 (J2SE 1.5). The JSR adds several new features (and alters the Java runtime slightly); for instance Atomic Variables, Futures, Thread Pools, Semaphores, Locks and barriers.

Callables are function indirections (normally function calls wrapped with inner-classes). Callable is an interface with a single method (call) which returns a generic return type. Futures can be used to indirect the return value of the wrapped function (see below).

Several queues (including blocking queues) are supported. Some of the queues are not thread safe (such as java.util.PriorityQueue), but others are — PriorityBlockingQueue for example.

One of the “improvements” in JSR166 is the addition of unstructured locking. Java currently allows per object locking by way of structured synchronised blocks, although it does not allow multiple locks to be obtained atomically. In structured locking, the lock is obtained on entry to the block and released on exit; this is quite different from unstructured locking, for example:

```java
lock.lock();
try {
    action();
} finally {
    lock.unlock();
}
```
Listing 2.6: Unstructured Locking

Here the lock is obtained and released in an unstructured way, this has advantages and disadvantages — locks can be held between methods, and released at different points in different code paths. But, it is possible to make a mistake and leave a code path that does not release the lock (or possibly does not obtain the lock). This new API also allows custom implementations of lock to be used, as well as providing read/write locks.

Atomic variables that support test and set operations have been added. They do not replace synchronisation, but can help to improve the performance of highly contended operations.

The correctness of the implementation depends on JSR133; which is the corrected Java memory model [28], [25], [26].

2.2.3.1 Why JSR166?

Why have Sun decided to incorporate JSR166 into Java 1.5?

2.2.3.1.1 Summary of the official reasons[18] for JSR166

- To produce a standardised, and small library to do for concurrency what the JDK1.2 collections framework does for Abstract Data Types.
- To provide high quality implementations of what, it is believed, a lot of developers do anyway (or would if the Java Memory Model allowed it).

2.2.3.1.2 And what it means for the author In the past I have needed to implement some of the things in the specification myself. Thread pools, Executors and Futures are a common pattern for reducing complexity and lock contention which I have programmed for two separate systems. Blocking and priority queue structures are also things I have needed (and written) in the past. The ConcurrentHashMap looks very nice, and is something that could not be implemented without the new memory model, (although I have previously written a version with per-bucket locking).

I am more ambiguous about the utility of the other components. I have never like unstructured locking and semaphores (except at the very lowest levels, and in compiler output where code paths can be verified); it is very easy to make mistakes with these unstructured primitives.

The atomic variables are very low level and, in my opinion, not as useful as the corrections to the semantics of final and volatile. But I can see a few, mostly performance related, places where I would have used them — particularly to prevent blocking on a lock which is already taken. I can’t see atomic variables becoming something I use regularly (because, unlike synchronised, they do not have easily thought about effects on the memory visibility ordering of the surrounding instructions).

The CyclicBarrier is cool, and can perform a function like a rendezvous — it stops any threads from passing until a threshold number of threads arrive. Then (optionally) runs a command before allowing the threads to continue.
The SynchronousQueue (where each put must be matched by a take before either thread can continue) is similar to rendezvous channels used in CSP and Ada. It is something which is nice to have (although I can't think of any occasions I would have used it).

The setDefaultUncaughtExceptionHandler() which has been added to Thread is fantastic and will make thread programming much prettier; no more messy try-catch blocks surrounding the statements in the run() method of each thread.

2.2.3.1.3 Why is it there? I think it is there for two reasons. Firstly that Java is more and more being used on servers and large systems, and the current primitives are not sufficient; both in terms of performance and the breadth of programming styles that can be used. The second reason is that there seems to be an element of, “look what we can do with our shiny new memory model and improved JVM”.

2.3 Ada Tasks [4]

Ada has language support for multi-threading in the form of Tasks. Each Ada task is an object. Ada tasks progress through several states, they are either inactive, blocked or ready to run. Tasks which are ready to run compete to be scheduled. Tasks which have finished go into the terminated state[21].

Ada tasks can:

- Wait for other tasks to complete.
- Send messages to each other (a rendezvous).
- Use Protected-Objects to provide both shared and exclusive access to data.

2.3.1 Example of an Ada task [31]

Ada tasks (in this case a task type) consist of a series of entry statements defining which messages the task accepts.

Examples adapted from [31].

```ada
task type Babbler is
  entry Start (Message : Unbounded_String; Count : Natural)
end Babbler
```

Listing 2.7: Task Type

This task has a single entry called Start which takes a number and a string.

Tasks are implemented like this;

```ada
task body Babbler is
  Babble : Unbounded_String;
  Maximum_Count : Natural;
begin
```

10
accept Start(Message : Unbounded_String; Count : Natural) do
    Babble := Message;
    Maximum_Count := Count;
end Start;
for I in 1 .. Maximum_Count loop
    Put_Line(Babble);
    delay 1.0;
end loop;
end Babbler;

Listing 2.8: Task Body

The accept makes the task wait for the start message. When it receives the start message, the task runs the code between the “do” and “end start”. The sending task cannot continue until this code block has finished executing. This is a rendezvous, and is used here to make local copies of the message parameters. Once the rendezvous has finished both tasks continue.

2.3.2 Starting a task

Procedure Noise is
  Babble_1 : Babbler;
  Babble_2 : Babbler;
begin
  Babble_1.Start(U("Hi, I'm Babble_1"), 10);
  Babble_2.Start(U("Hi, I'm Babble_2"), 6);
end Noise;

Listing 2.9: Starting a Task

This procedure starts two tasks, then waits for them both to end before it exits.

2.3.3 Data Protection

Ada provides Protected Types, which contain three kinds of operation;

1. Protected functions, allowing multiple tasks simultaneous read only access to internal data.
2. Protected procedures, which allow one task exclusive read-write access to the internal data.
3. Protected entries, which are like protected procedures but add a boolean barrier expression. The barrier must be true before a task is allowed to proceed past it. If not true the task is placed in a queue until the barrier becomes true.

protected type Resource is
  entry Seize;
  procedure Release;
private
  Busy : Boolean := False;
end Resource;

Listing 2.10: Protected Type
In this example we see a simple binary semaphore. Seize is a protected entry, it is guarded by the boolean expression “not Busy”; where “Busy” is a boolean variable. Tasks calling Seize are queued until “not Busy” becomes true. Only a single task is allowed into the procedures Seize and Release at any particular time.

2.4 Ruby Threads [30]

Ruby has language support for threads. For example (examples in this section adapted from [30]);

```ruby
for page in pages
threads << Thread.new(page) { |myPage|
  h = Net::HTTP.new(myPage, 80)
  puts "Fetching: #{myPage}"
  resp, data = h.get('/', nil)
  puts "Got #{myPage}: #{resp.message}"
}
end
```

Listing 2.12: Ruby Threads

Ugly. In ruby, all of the variables in scope when the threads are created remain available to the threads. Variables created inside the thread execution block are created per-thread.

2.4.1 Synchronisation of Thread Death

Ruby supports synchronisation of thread termination, it is possible to make the current thread wait for a group of other threads to terminate before continuing.

```ruby
threads.each { |aThread| aThread.join }
```

Listing 2.13: Join Ruby Threads
2.4.2 Shared variables

Although ruby supports shared variables via them being in scope when the thread is created, it also provides a per thread hash which contains shared variables referenced by name. These variables are not automatically synchronised.

2.4.3 Controlling Threads

Ruby allows the priority of threads to be set, but also has more primitive thread control. Ruby allows the programmer to explicitly stop, run, deschedule and wait for a thread. However, the ruby manual does not recommend the use of these controls.

2.4.4 Mutual Exclusion

Instead of explicitly controlling thread scheduling, ruby supports a few mutual exclusion and synchronisation primitives. The first is the Mutex class, which uses a semaphore to provide exclusive access to a section of code.

```ruby
mutex = Mutex.new

count1 = count2 = 0
difference = 0
counter = Thread.new do
  loop do
    mutex.synchronize do
      count1 += 1
      count2 += 1
    end
  end
end

spy = Thread.new do
  loop do
    mutex.synchronize do
      difference += (count1 - count2).abs
    end
  end
end

Listing 2.14: Ruby Mutex
```

These mutex do not provide any wait or notify mechanism. To do this the “condition variable” is provided.

```ruby
mutex = Mutex.new
cv = ConditionVariable.new

a = Thread.new { }
mutex.synchronize { }
  puts "A: I have critical section, but will wait for cv"
  cv.wait(mutex)
```

13
puts "A: I have critical section again"
}
}

Listing 2.15: Ruby Condition Variable

The condition variable allows a mutex to be temporarily released, until it receives a signal.

b = Thread.new { 
  mutex.synchronize { 
    puts "B: Now I am critical, but am done with cv"
    cv.signal
    puts "B: I am still critical, finishing up"
  }
}

Listing 2.16: Ruby Signal

The interface Condition will perform a similar function in Java 1.5[18].

2.5 Conclusion

We looked at Java, Ada and Ruby support for concurrency and saw how they differed.

Language support for threading has existed for sometime (for instance in Ada), however even with language support it is not always easy to create simple to understand syntax (for instance in Ruby). Java, with its structured locking and language support for wait-notify mechanisms does offer a clean usable programming interface; but still has some subtleties. All of these mechanisms still require relatively low level control of thread synchronisation conditions and Ruby and Java require explicit use of locks and monitors. Ada provides some more implicit synchronisation primitives, but these are still essentially the same as the monitor approach taken by Java.
Chapter 3

Chords

Programs must be written for people to read, and only incidentally for machines to execute. — H. Abelson and G. Sussman (in “The Structure and Interpretation of Computer Programs”)

3.1 Introduction

Chords are a high-level programing language structure for synchronisation of process execution. They use Join patterns (as described in the Join-Calculus) to express complex synchronisation conditions in an object-orientated (message passing) manner.

Chorded programming is heavily based around the idea of asynchronous methods — in particular asynchronous message passing on named channels. It uses this asynchronous idea to present a unified model of both thread control (starting and stopping threads), and to pass information from one thread of control to another.

Message passing is implemented using method call. Chords require groups of methods to all have been called before the code associated with them is allowed to execute. At most one of these method calls can be synchronous; the rest must be asynchronous.

Chorded programming has been implemented in Polyphonic $C^\#$ and is described in [1]. Polyphonic $C^\#$, is now included in a language called $C^\omega$. Since a compiler is not available for either language, this evaluation, and the subsequent work in this project, is based on descriptions in papers published on the languages.

In this chapter we look at the Join Calculus, then describe chords as they appear in Polyphonic $C^\#$. After that we look at how a programmer can use chords, and what some common chorded-programming idioms are. We then make a brief comparison between a Java and a Chorded implementation of the active-object pattern.

3.2 The Join Calculus

The Join Calculus[7] is an experimental language dealing with concurrency, synchronisation, distribution and mobile agents[5]. In this report we are primarily interested in the first two parts, although we touch on the second two.
The join calculus is a way of describing processes. Processes send and receive messages on channels. A process makes transitions by sending or receiving messages.

For instance a simple buffer can be described as

\[
\text{def get()}|\text{set}(s, v) = \text{return } s, v \text{ to get}
\]

The buffer matches a pair of get/set messages returning the argument of the set method on the get channel.

### 3.2.1 Basic Join Calculus programming

The examples in this section are adapted from [5]. The introduction to the Join-Calculus on that site heavily influenced the structure and content of this section.

Join-Calculus programs consist of two concepts; processes which execute asynchronously and cannot return a value, and expressions which execute synchronously and can return a value. Information is passed between processes by sending and receiving message on channels.

These messages are considered values (rather than references, or processes), and consist of zero or more other values. Channels and listening processes are described by a single language construct, this is designed to allow them to be implemented as functions.

### 3.2.2 Processes

Processes are implemented as top-level phrases terminated by “;”. For example, this process sends “1” on the channel “echo”:

```
spawn echo(1)
;;
```

Listing 3.1: Simple Join process

Spawn starts a process. Processes can be executed in parallel by composing them with the “|” operator. Here is a process that sends “1” and “2” to the channel echo in arbitrary order.

```
spawn echo(1) | echo(2)
;;
```

Listing 3.2: Parallel processes

Composite processes with local variables and conditionals use the “{|}” bracket pair to group processes.

```
spawn let x = 1 in
{| let y = x+1 in echo(y) | echo(y+1) | echo(x)|
;;
```

Listing 3.3: A composite process

Here x is bound locally over the composite process by the let statement. “1”, “2” and “3” will be sent to channel “echo” in arbitrary order.
3.2.3 Expressions

Expressions evaluate to a value. They may appear as values (for instance a constant integer), as the value to be sent in a message, or they may be defined as top level constructs. Expressions may send messages on synchronous channels⁷.

- “do” is used to start an expression.

Here an expression sends the value of expression “x” (which evaluates to “1”), on the synchronous channel “print”. The channel “print” returns a value (although in this case it is the empty value — similar to a “null” having the type “void” in Java!).

```plaintext
let x = 1
;;
do print(x)
;;
```

Listing 3.4: An expression

Expressions can also be composited, by the sequence operator “;”. Here a variable is bound locally then the sends to channel “print” are executed in order.

```plaintext
do
  let x = 1 in
  print(x) ; print(x+1)
;;
```

Listing 3.5: A sequential expression

Sequences of expressions can be used inside processes; for example, here a process sends “1” followed by “2” on the synchronous channel “print”, then “3” on the asynchronous channel “echo”.

```plaintext
spawn
  print(1) ; print(2) ; echo(3)
;;
```

Listing 3.6: A sequential process.

3.2.4 Channels

So far the Join-Calculus has not been very dissimilar to any other procedural language (apart from its parallel operator). However, the channel, and the way it is defined, is different and is seen as the main advantage the Join-Calculus has in implementability over other, similar, process Calculus.

Channels are named, and are either synchronous or asynchronous. They are defined by using the “let” keyword. The left-hand-side of the let expression defines the name of the channel, and the right-hand-side defines the process that implements the channel.

Every time a message is received on the channel the implementing process is spawned to handle it. Here is a channel which takes the message received and sends it to the channel “echo” twice.

⁷Synchronous channels return a value (although it may be the empty value). The type system implies that a channel is synchronous if it has a reply statement.
This channel is asynchronous (this is implied by the type system), so it is impossible to know when the message will be processed. Synchronous channels are defined in a similar way, except that they “reply” with a result. Here is a similar synchronous version of listing 3.7:

```
let double_print(x) = print(x); print(x); reply
;;
```

Listing 3.8: An synchronous channel

This channel is synchronous, and returns the empty result. In this case the messages to “print” will have been sent (and the results from “print” returned, since it is a synchronous channel) before the “double_print” channel returns its result.

### 3.2.4.1 Recursion

Channel definitions can be recursive, this allows synchronous channels to be used in a functional style, for example (again from [5]), here is a Fibonacci function:

```
let fib(n) =
  if n <= 1 then {reply 1}
  else {reply fib(n-1) + fib(n-2)}
;;
do print(fib(10))
;;
```

Listing 3.9: A recursive function

### 3.2.4.2 Higher-order functions

Channel names are considered values in the Join Calculus, this allows them to be sent in messages, which can be used to construct higher order functions. A function which applies any other function twice (that is, sends its argument to the given channel, then sends the result to that channel again before returning the result of the second call) can be seen below.

```
let twice(f) =
  let r(x) = reply f(f(x)) in
    reply r
;;
```

Listing 3.10: A higher order function
3.2.5 Join Patterns

Join-Patterns are the main feature of the Join-Calculus that we are interested in; they represent the synchronisation mechanism of the language. The join-pattern is an extension of the channel definition. The designers of the Join-Calculus made a slightly strange decision in their choice of syntax for the join-pattern; it is important to keep in mind that the “|” is parallel composition and not “or”, and that “and” represents choice between join patterns in a definition and not conjunction.

Listing 3.11 shows a data-structure in the Join-Calculus. The join pattern provides a counter. The state of the counter is held in its pending messages, the consumption of the pending message on channel “count” is also a synchronisation mechanism.

Three channels are defined, “count”, “inc” and “get”; the first pattern is “count” and “inc”. If a message is present on both of these channels the process “count(n+1) | reply to inc” is spawned. This process sends the message “n+1” on channel “count”, then replies the empty message to channel “inc”.

\[
\text{let } \text{count}(n) | \text{inc}() = \text{count}(n+1) | \text{reply to inc} \\
\text{and } \text{count}(n) | \text{get}() = \text{count}(n) | \text{reply n to get} \\
\]

\[
\text{spawn } \text{count}(0) \\
\]

Listing 3.11: A counter

If a message is present on both channels “count” and “get”, then the process “count(n) | reply n to get” is spawned. This process re-issues the message “n” on channel “count” and returns the value “n” on channel “get”.

If messages are present on all three channels the runtime chooses one of the patterns to match.

3.2.6 Why the Join-Calculus in Polyphonic C\#

The Join-Calculus was explicitly designed to be implementable, specifically its channels are implementable as functions. One of the designers of the Join Calculus moved to work for Microsoft research, who are interested in finding simpler higher-level alternatives to thread based programming. This lead to the implementation of Join-Calculus like join-patterns in C\# (called Polyphonic C\#) and then C\ω. At the current time neither Polyphonic C\# or C\ω have compilers available for download, but several papers have been published to describe the languages.

3.3 Polyphonic C\#

Polyphonic C\# is a language[1] from Microsoft Research based on the Join-Calculus. It adds Join patterns to the C\# language.

3.3.1 Asynchronous Methods

Polyphonic C\# has methods which do not return any result (procedures) and return immediately. The body of the method then executes at some later time (for example when a thread from a pool becomes available).
3.3.2 Chords

A chord is a set of method signatures with a single body. The same method signature may appear in multiple chords.

1. Each chord may have at most one synchronous method, all the other methods must be asynchronous.

2. When all of the methods in a chord have been called, the body of the chord runs.

3. The chord executes in the thread of the synchronous method, or a new thread if all of the methods are asynchronous.

4. Any value returned from the chord is returned via the synchronous method (if there is one).

5. The synchronous method will block until all of the methods in the chord have been called and the body is run.

3.3.3 Example[1]

```java
public class Buffer {

    public string Get() & public async Put(string s) {
        return s;
    }

    public string Get() & public async Put(int n) {
        return n.ToString();
    }
}
```

Listing 3.12: Chord Buffer

Here, multiple strings and ints can be put into the buffer. Get()s will match one of the chords and retrieve one of the values from a Put(). Excess Get()s will block until a Put() is available. Which Put() matches with which Get() is (in general) non-deterministic.

3.4 Chord Programming Primer

Chords are a synchronisation mechanism. They allow threads to pass messages between each other, and allow a thread to wait for a set of messages. The key features of chord programming are as follows.
3.4.1 Starting a Thread

In a chord language, a thread is started by calling an asynchronous method; here is a class which counts to 20 in a new thread:

```java
public class Counter {
    public async count () {
        for (int i = 1; i <= 20; i++) {
            System.out.println(" "+i);
        }
    }
}
```

Listing 3.13: Async Counter

Threads can also be started by the execution of a fully asynchronous join pattern — in fact, the above is just a special case of this feature. Here is a class which takes two numbers and counts from one to the other in a new thread.

```java
public class Counter {
    public async from (int from) & public async to (int to) {
        int direction = from <= to ? 1 : -1;
        for (int i = from; i != (to+direction); i+=direction) {
            System.out.println(" "+i);
        }
    }
}
```

Listing 3.14: Multiple Async Counter

3.4.2 Locking

A simple binary semaphore is seen below. This illustrates the use of private channels to store the state of the object. If there is a message on the “closed” channel then the semaphore is down, if there is a message on the “open” channel then the semaphore is up.

```java
public class Semaphore {
    public Semaphore () {
        // emits a message on channel open.
        open();
    }

    public void up () & private async closed ()
```


Listing 3.15: Binary Semaphore

3.4.3 Bounded Data Structures

A bounded buffer can be constructed from the previous techniques. This example illustrates the use of | to indicate that a message on either channel can be consumed to trigger the chord. Puts to a full buffer will block, as will gets to an empty one. It uses an array as a cyclic store.

```java
public class Buffer {
    public Buffer(int size) {
        // emits a message on channel empty.
        empty(0, 0, new Object[size]);
    }

    public Object get() &
    private async full(int start, int stop, Object[] messages) |
    private async part(int start, int stop, Object[] messages) {
        // consumes a message on channel get, and one from either full, // or part.

        Object result = messages[start];
        start = (start + 1) % messages.length;
        if (start == stop) {
            empty(start, stop, messages);
        } else {
            part(start, stop, messages);
        }
        return result;
    }

    public void put(Object object) &
    private async empty(int start, int stop, Object[] messages) {
```

22
3.5 Chord Programming Idioms

Chorded programming differs from normal object-orientated programing in several key areas. In particular, the following idioms are central to chorded programming:

3.5.1 List of Idioms

- Mutable state is stored in the pending messages of the object. In object-orientated programing the mutable state is stored in member variables.

- Constructors configure the starting state of an object by sending messages on private async channels. In object-orientated programing the constructors set initial values of member variables.

- Private channels can be used to “switch on” and “switch off” other channels. In Java an explicit wait-notify condition, with message queueing, must be written.

3.5.2 The notion of state

Chord programming differs from normal object-orientated programing in how it handles state in concurrent systems. An illustrative and simple example would be a simple synchronised counter. First, how it could be implemented in Java:

```java
public class Counter {
    private int count = 0;

    public synchronized void inc() {
        count ++;
    }
}
```

Listing 3.16: Bounded Buffer
```java
public synchronized void dec()
{
    count --;
}

public synchronized int getValue()
{
    return count;
}
```

Listing 3.17: OO Counter

And now, an implementation using chords.

```java
public class Counter
{
    Counter()
    {
        value(0);
    }

    public void inc() & private async value(int value)
    {
        value(value+1);
    }

    public void dec() & private async value(int value)
    {
        value(value-1);
    }

    public int getValue() & private async value(int value)
    {
        value(value);
        return value;
    }
}
```

Listing 3.18: Chord Counter

Note that these two counters are not exactly the same, the Object Orientated one has the additional feature that the lock can be obtained by any thread simply by executing

```java
public void extraFeature(Counter counter)
{
    synchronized(counter)
    {
        counter.inc();
        someOtherMethod();
        counter.dec()
    }
}
```

Listing 3.19: OO Counter extra feature
The chord version does not provide this facility, but it could easily be added — much more easily than it can be removed from the Object Orientated version — by simply combining the counter with a semaphore.

This is a general, and important, idiom. Mutable state is stored in the pending messages of the object, not in member variables.

3.5.3 Initial state

An object can behave differently depending on how the constructor issues messages. For example here is a general semaphore with a configurable upper bound.

```java
public class Semaphore
{
    public Semaphore(int upperBound)
    {
        for (int i = 0; i < upperBound; i++)
        {
            open();
        }
    }

    public void up() & private async closed()
    {
        open();
    }

    public void down() & private async open()
    {
        closed();
    }
}
```

Listing 3.20: General Semaphore

In general, constructors configure the starting state of an object by sending messages on private async channels — this is different to object orientated programming, which would use member variables.

3.5.4 Read/Write locks

Here is a read/write lock which uses private channels to make available different parts of the interface, in this case read locks and write locks depending on the current situation.

```java
public class ReadWrite
{
    public ReadWrite()
    {
        available();
    }
}
```

25
```java
public void readLock() & private async available() {
    readable(1);
}

public void readLock() & private async readable(int count) {
    readable(count+1);
}

public void readUnlock() & private async readable(int count) {
    if (count == 1) {
        available();
    } else {
        readable(count-1);
    }
}

public void readUnlock() & private async writePending(int count) {
    if (count == 1) {
        writeReady();
    } else {
        writePending(count-1);
    }
}

public void writeLock() & private async available() {
    unavailable();
}

public void writeLock() & private async readable(int count) {
    writePending(count);
    waitTillReady();
    unavailable();
}

private void waitTillReady() & private async writeReady() {
    return;
}

public void writeUnlock() & private async unavailable() {
    available();
}
```
3.6 Active Objects [17]

The active object pattern is similar to the idea of asynchronous methods. When a client sends a message (a method call) to the active object, the request is put into a queue. The active object then uses its own, dedicated, thread to execute each of the requests in turn.

This has two main advantages; firstly the active methods in the object are automatically synchronised (they all execute in the active object's own thread). Secondly, the active object can make its own decisions about how many threads a particular request needs.

We choose to look at the active object pattern in Polyphonic C♯ and compare it to a standard Java implementation for two reasons; firstly because it is one of the examples shown in [1], and secondly because it lends itself particularly well to a chorded implementation.

3.6.1 Polyphonic C♯ implementation [1]

The Polyphonic C♯ implementation, shown in [1], uses an abstract super-class to describe the pattern:

```java
public abstract class ActiveObject {
    protected bool done;

    abstract protected void ProcessMessage();

    public ActiveObject ()
    {
        done = false;
        mainLoop();
    }

    async mainLoop()
    {
        while (!done)
        {
            ProcessMessage();
        }
    }
}
```

Listing 3.22: Active Object

Then all of the active objects inherit from this super-class, and implement the `ProcessMessage()` method as part of a chord (or chords). The method `mainLoop()` is a single asynchronous method, so runs in its own thread. For example, from the same paper, here is an implementation of ActiveObject;
The main loop in ActiveObject calls ProcessMessage(), which allows one of the chords WireQuote or CloseDown to run each time.

This idea of message passing between active processes is similar to the communication primitives in CSP[14], where a method on an object can be though of as a channel on a process.

### 3.6.2 Java Implementation

Listing 3.24 is an implementation in Java 1.4 (without generics), it does not define futures or any other advanced concepts.
m_controller = new Controller();
}

public void deactivate()
{
    m_controller.stop();
}

private boolean isActive()
{
    return !m_controller.isStopped();
}

protected final void scheduleTask(final Task task)
{
    m_controller.scheduleTask(task);
}

private static class Controller
{
    private TaskQueue m_queue = null;

    public Controller()
    {
    }

    public synchronized void start()
    {
        if (m_queue == null)
        {
            m_queue = new TaskQueue();
        }
    }

    public synchronized void scheduleTask(final Task task)
    {
        start();
        m_queue.addTask(task);
    }

    public synchronized void stop()
    {
        if (m_queue != null)
        {
            m_queue.stop();
            m_queue = null;
        }
    }

    public synchronized boolean isStopped()
    {
        return m_queue == null;
    }
}
/**
 * This is a use once Scheduler. It starts when it is first given a task, but cannot be restarted once stopped.
 * It will complete all pending tasks before stopping.
 */
private static class TaskQueue
{
    private final ArrayList m_list = new ArrayList();
    private Thread m_thread = null;
    private boolean m_stop = false;
    private static final long MAX_WAIT = 10000;

    public synchronized void addTask(final Task task)
    {
        if (m_thread == null)
        {
            m_thread = new Thread(new Runner());
            m_thread.setDaemon(false);
            m_thread.start();
        }
        m_list.add(task);
        notifyAll();
    }

    private class Runner implements Runnable
    {
        public void run()
        {
            while (!shouldStop())
            {
                try
                {
                    nextTask().doTask();
                }
                catch (final InterruptedException e)
                {
                    // check in loop condition
                }
            }
        }
    }

    private synchronized boolean shouldStop()
    {
        return m_stop && m_list.isEmpty();
    }

    public synchronized void stop()
    {
        m_stop = true;
        m_thread.interrupt();
    }
}
public synchronized Task nextTask() throws InterruptedException {
    while (true) {
        if (m_list.isEmpty()) {
            wait(MAX_WAIT);
        } else {
            return (Task) m_list.remove(0);
        }
    }
}

private static interface Task {
    void doTask();
}

Listing 3.24: Implementation of Active Object in Java

3.6.3 Comparison

The Java implementation is much more complex. This is primarily because it has to handle the queueing and synchronisation of task starting and stopping itself. The chorded version has all of this structure implicitly defined by the use of join-patterns. This is why the chorded approach to multi-threaded programming and synchronisation is considered a higher level representation.

The active-object example is an extreme case of the difference in clarity of the two styles of programming; however it is difficult (we have failed so far) to find an example where the clarity of the Object-Oriented implementation is better than the clarity of the chorded representation.

This is mainly due to the encapsulation of the wait-notify conditions; in Java the wait-notify conditions are encoded across at least two methods (a wait in one method and the notify in another). These conditions can involve arbitrary logic, complex condition loops and must be synchronized explicitly. In chorded programming the synchronisation conditions are captured in the pattern of chords composing the object.

Sometimes however, especially in non-blocking data structures, the object-orientated version is of similar clarity but shorter than the chorded representation; an example of this can be seen in the “counter” example, listings 3.17 and 3.18 – although as previously described these counters are not identical.

3.7 Conclusions

We looked at the Join-Calculus, then at Polyphonic C♯ and how it uses ideas from the Join-Calculus. It uses asynchronous methods and Join-patterns to control threads and synchronisa-
tion. We described how to program in chorded languages and what the principal differences from object-orientated programming style are.

The Join-Calculus provides an implementable, high-level approach to concurrent programming. In particular its use of channels defined as processes allows it to be implemented using functions (or methods). Polyphonic C♯ is based on the Join-Calculus, it provides asynchronous methods and Join-Pattern style synchronisation, although no compiler is yet publicly available for it.

The Polyphonic C♯ provides a higher level view of message passing and thread control than the normal Java monitor and wait-notify condition variables. Many common concurrency control structures, like locks and semaphores, can be implemented with chords as can common data-structures like counters, buffers and queues.

Chorded programs can look very different from more typical object-orientated programs; programmers are required to understand a new set of idioms for encoding state in chorded languages.

The Join-Calculus does not map exactly to chorded programming. The Join-Calculus does not have private channels, although it does support lexically scoped names which can be used in a similar way. The Join-Calculus does not expressly support objects, although it does allow channels to be collected together in a class-like manner. The Join-calculus is a programming language and does not directly describe its implementation and semantics.

In order to support the implementation of chorded programming and to allow us to reason about, and analyse, the properties of chorded programs we would like to develop a formal description that reflects more precisely the structure and features of chorded programs.
Chapter 4

Programs, Classes and Objects

Computer Science is no more about computers than astronomy is about telescopes. — E. W. Dijkstra

4.1 Introduction

So far we have loosely described what a chorded program looks like and how a chorded system behaves. We have not looked at how to implement or reason about chorded systems.

In order to reason about the properties of chorded programs a formal description of a chorded program is needed. In this chapter we present such a description. We try to keep the formulation as simple as possible, avoiding ideas about execution and semantics in preference to presenting the program as a set of constraint forming relations between finite sets.

Later on, we hope this will allow us to vary the accuracy of the analysis we preform (by using stronger or weaker definitions of operations on these sets). The well-formedness rules are presented to keep our analysis safe (that is the outcomes will be valid if not accurate).

We will also use this formal system to help show how our implementation of chords in Java works (see chapter 6).

The constraints are developed by construction on a data structure of channels, types (classes) and objects. As a rough-overview the final system looks something like in figure 4.1.

We will begin with programs, then go on to look at types, visibility and finally objects and state.

4.2 Program

A program in our chord language consists of a set of type names $\text{TypeName}_*$, each $t \in \text{TypeName}_*$ is mapped to a $\text{td} \in \text{TypeDefinition}_*$ by the function $\text{class}(t)$ as defined below (see section 4.3).

TypeNames are primitive and atomic, although TypeDefinition is a tuple of relations (as seen in figure 4.1).

1$\text{TypeName}_*$ differs from $\text{Name}_*$. $\text{Name}_*$ is a type, and is potentially infinite. $\text{Name}_*$ is the set of type-names actually appearing in the program — and is strictly finite. Similar for $\text{Body}_*$, and the other $*$ subscripted sets. Later we will use this fact to help show that our analysis terminates.
Figure 4.1: Overview of a single class system
We define $\text{Body}_*$ to be all of the bodies in the program:

$$\text{Body}_* = \{ b \mid \exists td \in \text{TypeDefinition}_*: b \in \text{bodies}(td) \}$$

similarly for channels.

$$\text{Channel}_* = \{ c \mid \exists td \in \text{TypeDefinition}_*: c \in \text{channels}(td) \}$$

We are going to use a running example to illustrate this section. The box-outs (like this one) which follow will contain the values of various of the sets and relations as calculated for this example. The code used for the example will be:

```java
class E {
    public void up() & private async closed() {
        open();
    }

    public void down() & private async open() {
        closed();
    }
}
```

Listing 4.1: The Example

All of the channels and bodies in the program, which contains only the class E.

$$\text{Channel}_* = \{(E, down), (E, up), (E, closed), (E, open)\}$$

$$\text{Body}_* = \{((E, up), (E, closed)), "open()"),
\((E, down), (E, open)), "closed()"), \}
$$

4.3 Class

A class is a set of channels and a set of bodies.

We define a function $\text{class}(t)$ which maps from a type name to a type definition. Type names are somewhat ill-defined, since the precise formulation depends on the underlying representation, however a type name must be mapped to a unique type definition (the function $\text{class}(t)$ is bijective, so invertible). $\text{class}(t)$ returns a well-defined unambiguous description of a class in our chord language.

$$\text{class} : \text{TypeName}_* \to \text{TypeDefinition}_*$$
class\( (t : \text{TypeName}) = \{[b_1, \ldots, b_n : \text{Body}, \{c_1, \ldots, c_m : \text{Channel}]^t}\) \\

The function \(\text{bodies}(td)\) returns the set of bodies\(^2\) belonging to the class \(t\).

\[
\text{bodies} : \text{TypeDefinition} \rightarrow \mathcal{P}\{\text{Body}\} \\
\text{bodies}([b_1, \ldots, b_n : \text{Body}, \{c_1, \ldots, c_m : \text{Channel}]^t) = \{b_1, \ldots, b_n : \text{Body}\}
\]

The function \(\text{channels}(td)\) returns the set of channels that can be sent to on objects of this type.

\[
\text{channels} : \text{TypeDefinition} \rightarrow \mathcal{P}\{\text{Channel}\} \\
\text{channels}([\{b_1, \ldots, b_n : \text{Body}, \{c_1, \ldots, c_m : \text{Channel}]^t) = \{c_1, \ldots, c_m : \text{Channel}\}
\]

The bijection \(\text{type}(t : \text{TypeDefinition}\_\ast)\) is the inverse of the bijection class

\[
\text{type} : \text{TypeDefinition}\_\ast \rightarrow \text{TypeName}\_\ast \\
\text{type}([\{b_1, \ldots, b_n : \text{Body}, \{c_1, \ldots, c_m : \text{Channel}]^t) = t
\]

A formal representation of the class “E”.

\[
\text{class}(E) = \{((E, \text{up}, (E, \text{closed})), \text{“open();”}), \\
((E, \text{down}, (E, \text{open})), \text{“closed();”}), \}, \\
((E, \text{down}, (E, \text{up}, (E, \text{closed}, (E, \text{open}))))^E
\]

### 4.3.1 Ordering on TypeName

Since TypeName is not well defined in our system, there is no well defined ordering. However, one should be provided since it is needed to break ties in later total orders. For Java-like type names a lexicographical ordering of the fully qualified class name should be sufficient. The ordering on TypeName must be a total-order so that its finite subsets (TypeName\_\ast) will be well-ordered. The importance of this can be see in section 4.4.1.

**Axiom 4.1 (TypeName totally ordered)** \(\text{TypeName is a totally ordered, possibly infinite, set.}\)

### 4.4 Channels

A channel is similar to a method in OO programming. In chorded-programming it is one element of a chord. Channels are defined by name, and are implemented by a queue of messages in objects. Channels are defined by their containing type and a unique (within the given type) name;

\[
\text{Channel} = (t : \text{TypeName}, n : \text{Name})
\]

\(^2\) Here we use TypeDefinition, rather than TypeDefinition\_\ast, since there is no bijection requirement on this function. This means that any TypeDefinition \(\notin\) TypeDefinition\_\ast can simply map to the empty set of Body
Name : valid channel names

\[
\text{name}((t: \text{TypeName}, n: \text{Name})) = n \\
\text{type}((t: \text{TypeName}, n: \text{Name})) = t
\]

The name and type of a channel;

\[
\text{name}(\text{E}, \text{down}) = \text{down} \\
\text{type}(\text{E}, \text{down}) = \text{E}
\]

### 4.4.1 Channel ordering

We need to order the channels. It will become clear why when we look at how pattern matching is used in the implementation of chorded-systems.

By definition the name of a channel is unique within its containing type. Since we already have a total order on TypeName (axiom 4.1) we can use it to define a total-order on Channel.

The total order we will use is: first order the Channel by the ordering on type names (as mapped by \(t : \text{Channel} \rightarrow \text{TypeName}\)). Then we can use the name of the channel (as defined by \(\text{name} : \text{Channel} \rightarrow \text{Name}\)) to break ties (which will occur for every channel in a particular type). Any total order over \(\text{Name}\) will do, a simple lexicographic (dictionary) ordering is what we will use.

By defining a total-order over the infinite set Channel we can ensure that Channel\(^*\) (which is a finite set, taking its order from Channel) is well ordered. In turn allowing channels to be indexed within a class (see 4.4.2).

**Lemma 4.2 (Channel totally ordered)** Channel is a totally ordered, possibly infinite, set.

**Lemma 4.3 (Channel\(^*\), well ordered)** Channel\(^*\) is a well ordered, finite, set.

### 4.4.2 Indexing channels and bodies

In order to map channels in class definitions onto queues in objects in an efficient, implementable way, it is necessary to order and index the channels. In C^\# this is done using the order that the chords appear in the source-code of the class, our formal description uses a less arbitrary method based on a well defined ordering of all channels within a program.

We describe channels and bodies within a class with a numerical subscript (index). This subscript refers to the position of the channel within the infinite set Channel (or, more precisely, is based on the ordinal of the elements of some finite subset of Channel).

A complete description of how these indexes are derived can be found in appendix D.
4.5 Bodies

Bodies of code are defined by the pattern that guards the body, and the code which makes up the body. Each body represents a single chord in the program.

\[
body = (p : \text{Pattern}, s : \text{Code}) \\
\text{Pattern} : \mathcal{P}[	ext{Channels}] \\
\]

\[
\text{pattern}((p : \text{Pattern}, s : \text{Code})) = p \\
\text{code}((p : \text{Pattern}, s : \text{Code})) = s
\]

The type of code is not defined, since it is implementation dependent. Later on, however, we define some functions which can be applied to code.

4.6 Inheritance

In our basic formalism we do not deal with inheritance or polymorphism. It is unnecessary since we do not have assignment (or any other notion of variables or store). There is no subsumption. We only have type to support our notion of visibility.

4.7 Operations on Code

We avoid specifying the format and semantics of the code used to write the polyphonic language in our formalism. This way the formalism can be used to analyse any language which provides the necessary properties. These properties are defined by the following functions.

\[
\text{calledChannels} : \text{Code} \rightarrow \text{Channel}
\]

The implementation of this function will depend on the actual language underlying the analysis, but it must follow these rules:

1. Returns at least the maximum number of channels. We will show later that this must be the case to allow our analysis to be safe.

2. Must obey the rules of well-formedness (section 4.9).

\[
\text{calledChannels}("\text{open();}"); = \{1 : (\text{E, open})\}
\]
4.7.1 Counting calls

What does $\text{Channel}^\gamma$ mean? The answer is somewhat complex, so a more full discussion is presented in appendix $E$. It is a notational way of counting the number of times a particular channel can be called. Instead of mapping to some set $\in \mathcal{P}(\text{Channel}_\gamma)$, the calledChannels relation maps to an augmented set. Each member of the set $\text{Channel}^\gamma$ is prefixed by some $n \in \mathbb{N}^\infty$, representing the number of times the channel can be called from the given Code.

The key feature is that $\text{Channel}^\gamma$ is of a fixed cardinality.

We choose this method, of ensuring that the set remains finite by construction, because it was simpler than constructing the set then proving it was finite in all cases (appendix $E$ has more details).

4.8 Channel Protection

Our formalism has two types of channels; public and private. Private channels can only be called from within the object itself. Public channels can be called by at least one other part of the program. In a Java-like language, private channels would be methods marked “private”. Public channels would be methods marked “public”, “protected” and “package”. We don’t deal with inner classes in this formalism.

We need this notion of visibility in order to allow us to reason about chords in the ways described in the introduction and primer.

4.8.1 Visibility

We modify our definition of channel to include the notion of privacy. The new definition is augmented by a visibility modifier of either public or private.

$$\text{Channel} = (t : \text{TypeName}, n : \text{Name}, v : \text{Visibility})$$

$$\text{Visibility} \in \{\text{public, private}\}$$

$$\text{visibility} : \text{Channel} \rightarrow \text{Visibility}$$

$$\text{visibility}(t : \text{TypeName}, n : \text{Name}, v : \text{Visibility}) = v$$

$$\text{privateChannels} : \mathcal{P}(\text{Channel}_\gamma) \rightarrow \mathcal{P}(\text{Channel}_\gamma)$$

$$\text{privateChannels}(\text{chan}) = \{c | c \in \text{chan} \land \text{visibility}(c) = \text{private}\}$$

$$\text{publicChannels} : \mathcal{P}(\text{Channel}_\gamma) \rightarrow \mathcal{P}(\text{Channel}_\gamma)$$

$$\text{publicChannels}(\text{chan}) = \{c | c \in \text{chan} \land \text{visibility}(c) = \text{public}\}$$
Well Formedness

A class is well formed, if it only contains patterns for which it has channels\(^3\), and it only calls channels which are in the program and are visible to it.

### 4.9.1 Patterns

We need to find out which channels are used in patterns in a class.

\[
patterns(td : TypeDefinition) = \{c | b \in bodies(td) \land c \in pattern(b)\}
\]

### 4.9.2 Called Channels

We also need to find out which channels are called.

\[
called(td : TypeDefinition) = \{c | b \in bodies(td) \land s \in code(b) \land n : c \in calledChannels(s)\}
\]

### 4.9.3 Program Formedness Rules

All classes must be well formed. From now on, class means well formed class. The classes are well formed if:

\[
\forall (td \in TypeDefinition) : patterns(td) \subseteq channels(td)
\]

\[
\forall (td \in TypeDefinition) : called(td) \subseteq Channel_{*}
\]

\[
\forall (td \in TypeDefinition) : \forall (c \in called(td)) : \text{private}(c) \implies (\text{type}(td) = \text{type}(c))
\]

Although, in all the examples we look at, our program only consists of one class we do not preclude programs which consist of many-classes. Since our analysis is primarily interested in the private channels of a class, the presence of other classes is immaterial.

### 4.10 Object

An object implements a class, and is a set of pending messages. In Polyphonic C\(^\#\) (and similarly in our Java-based implementation; see subsequent chapters) an object is an augmented C\(^\#\) object, with the addition of pending message queues and the code for synchronisation and dispatch of chords.

\(^3\)A pattern containing a channel not in the class will never get called (this situation is similar to a method in OO programming not having a name!).

40
For channels($\text{class}(t)) = \{c_1, \ldots, c_m : \text{Channel}\}$ an object of type $t$ is:

$$\text{object}_i = [q_1, \ldots, q_m : \text{mQueue}]$$

$m\text{Queue}$ : list of valid messages

$$\text{length} : \text{mQueue} \rightarrow \mathbb{N}^\infty$$

$$\text{length}(q : \text{mQueue}) = |q|$$

$\text{empty} : \text{mQueue} \rightarrow \text{boolean}$

$$\text{empty}(q : \text{mQueue}) = \text{length}(q) == 0$$

$finite : \text{mQueue} \rightarrow \text{boolean}$

$$finite(q : \text{mQueue}) = \text{length}(q) \neq \infty$$

$$\text{class}([q_1, \ldots, q_m : \text{mQueue}]) = \text{class}(t)$$

4.10.1 Mapping channels to queues

The mapping of channels to queues is simply by index equality. We showed, in section 4.4.2, that channels can be assigned a unique (Lemma D.1), contiguous (Lemma D.2) and well defined (Lemma D.3) index. By definition our objects have queues indexed in the same well defined, unique and contiguous manner; so by making sure (again — by definition) that there are the same number of queues as channels the mapping must be one-to-one. The mapping on the indexes is defined by the function;

$$\{(i, j) \mid (i, j) \in \{1, \ldots, m\} \times \{1, \ldots, m\} \land i = j\}$$

4.10.2 mQueue

What is an $m\text{Queue}$? An $m\text{Queue}$ is a structure for containing an ordered set of messages. Queues have two operations,

- $\text{add}(\text{Message}, \text{mQueue})$, which puts the given message at the end of the queue.
- $\text{remove}(\text{Message}, \text{mQueue})$, which sets the message equal to the first element of the queue then removes the first element of the queue. This operation does not work on empty queues.

4.10.3 Notation for queues

Queues are described using a list notation. There are four possible notations for queue;
- Unknown contents: $[\ldots]_m$
- Queue of length $n$: $[n]_m$
- Empty queue: $[]_m$, or $[0]_m$
- Infinite length queue: $[\infty]_m$

Since our analysis does not deal with the execution of chords, we do not need to store the contents of each message (which might be the parameters of the channel); we only need to store the presence or absence (and number) of messages on each channel.

4.10.3.1 Definition of add

$$\text{add}(\text{message}, [\ldots]_m) = [\ldots]_m$$
$$\text{add}(\text{message}, [n]_m) = [n + 1]_m$$
$$\text{add}(\text{message}, []_m) = [1]_m$$
$$\text{add}(\text{message}, [\infty]_m) = [\infty]_m$$

4.10.3.2 Definition of remove

$$\text{remove}(\text{message}, [\ldots]_m) = [\ldots]_m$$
$$\text{remove}(\text{message}, [n]_m) = [n - 1]_m \text{ (where } n > 0)$$
$$\text{remove}(\text{message}, [\infty]_m) = [\infty]_m$$

Note that $\text{remove}(\text{message}, [\ldots]_m)$ is undefined, if this happens in an equation system the analysis is not well formed. And has to stop at that point. However, on inspection of the pattern matching algorithm presented in chapter 6 is is clear that this situation never arises.

4.11 Object State

Object state can change in two well defined ways. Either a message arrives on a channel, or a chord is dispatched. Given two objects $\text{object}_u$ and $\text{object}_v$, $\text{object}_u$ in state $s$ can become $\text{object}_v$ in state $s + 1$ iff $\text{object}_u = \text{object}_v$, or

$$\text{arrive}((m, c_i), \text{object}_u) = \text{object}_v$$

or

$$\text{dispatch}((\text{pattern}, \text{body}), \text{object}_u) = \text{object}_v$$

These are the only ways the state of objects in the system can change.
4.11.1 Message arrival

When a message (message) arrives on channel $c_i$, it is added to the corresponding queue $q_i$.

\[
\text{arrive} : \text{Message} \times \text{Channel} \times \text{Object} \rightarrow \text{Object}
\]

\[
\text{arrive}((\text{message}, c_i), [q_1, \ldots, q_m : \text{mQueue}]') = [q'_1, \ldots, q'_m : \text{mQueue}]'
\]

where

\[
\forall j \in (1, \ldots, m) : (j \neq i \rightarrow q'_j = q_j) \land (j = i \rightarrow q'_j = \text{add}(\text{message}, q_j))
\]

and

\[
\text{type}(c_i) = t
\]

The last assertion prevents a message for one type arriving for an object of another type — potentially on a non-existent channel index. However, by well-formedness (section 4.9) we know that only channels which exist in the program are called; and by our channel to queue mapping (section 4.10.1) we know each channel has a corresponding queue. So the final assertion is redundant; messages can not arrive for non-existent queues even without the type check.

4.11.2 Chord dispatch

When a chord ($\text{pattern, body}$) is dispatched from object $= [q_1, \ldots, q_m : \text{mQueue}]'$ the first message is consumed from each queue corresponding to each channel in the pattern.

\[
\text{dispatch} : \text{Body} \times \text{Object} \rightarrow \text{Object}
\]

\[
\text{dispatch}((\text{pattern}, s), [q_1, \ldots, q_m : \text{mQueue}]') = [q'_1, \ldots, q'_m : \text{mQueue}]'
\]

where

\[
\forall j \in (1, \ldots, m) : (c_j \in \text{pattern} \rightarrow q'_j = \text{remove}(\text{message}, q_j)) \land (c_j \notin \text{pattern} \rightarrow q'_j = q_j)
\]

and

\[
\forall j \in (1, \ldots, m) : c_j \in \text{pattern} \rightarrow \text{type}(c_j) = t
\]

From the dispatch algorithm presented in chapter 6, we can see that a chord will only be dispatched from an object if there is at least one message available in each queue which the chord consumes a message from. This makes the final assertion above redundant. It also means that remove will be well defined — it will never be called on an empty queue.

4.12 System State

The state of the overall system is described by the state of the objects within the system, and the set of messages which are “in transit”. See chapter 9.
4.13 Conclusion

We have shown how an abstract formal representation of a chorded program can be built up. This representation is independent of the underlying language, as long as the language has certain properties. We defined well-formedness of a representation.

We showed how channels can be indexed and mapped to queues, and how these queues operate. Finally we described state change in our chorded system and showed it to be limited to two transformations.

This is the first such formal description of chorded programs (although perhaps the Join-Calculus can also be considered to be formal, it is itself a language).

The formal description does lack some aspects of chorded programs; in particular it does not distinguish between synchronous and asynchronous channels. Perhaps this could be added to the formalism by future work.
Chapter 5

Implementation Overview

The question of whether computers can think is like the question of whether submarines can swim. — E. W. Dijkstra

In chapter 3 we described how to program with chords, and what chorded programs look like. Then, in chapter 4 we formalised the notion of a chorded program. Now we are going to look at the implementation of chorded programming produced for this project. In this chapter we will examine our strategy for implementing a compiler supporting chords, and in several subsequent chapters look at how various features of our formal description are supported in the code output by the compiler.

5.1 Introduction

We developed an implementation of chords for Java as a pre-processor. The chorded-Java is translated to standard Java and then compiled to byte code with a standard Java-Compiler.

This chapter covers the framework developed for performing the compilation (pre-processing) and looks at the choice of tools. We use Antlr as a parser generator and also a tree-walker. We use Velocity to generate the auxiliary code for matching, marshalling and dispatch in chorded programs.

We believe this is the first publicly available implementation of chord support in an object-orientated language.

5.2 Implementation Strategy

Polyphonic $C^\#$ does not currently have a compiler available for download. However, in [1], the authors describe an implementation based on translating Polyphonic $C^\#$ into $C^\#$. This is the simplest way of adding new features to a language, it is similar to wrapping an existing compiler with the new one.

When the idea of looking at an implementation of chords in Java came up, two possible strategies were suggested; the first was to follow the approach described in [1]. The second was to write a compiler which compiled directly to bytecode.
Normally it is considered bad to implement a pre-processor for Java (which is what the first strategy would boil down to). This is because pre-processors can hide the real semantics of the eventual code from the programmer. To some extent this would be the case here too. However, as a first step toward a full chorded Java compiler the first method was chosen.

Implementing a chorded-Java to Java translator should require less effort than implementing a full blown, and extended chorded-Java to bytecode compiler, hopefully resulting in a more correct and complete implementation.

5.3 Objectives

- Create an implementation that can easily be updated to accommodate newer versions of Java as they arrive.
  
  Java 1.5 is just around the corner, and further developments to the Java language are likely.

- Produce Java classes that require no external library support.
  
  Asking developers to ship an extra library, just so they can use chords seems excessive. It will probably simplify usage if the produced Java classes are entirely self contained.

- Support a simple and clean method of changing the implementation of chords.
  
  The algorithms for implementing chord matching and dispatch are still in development. It is likely that better (faster, simpler) ways of implementing them will found. The compiler should be easily updatable to reflect these changes.

5.4 Target Java Version

When this project began, Java 1.5 was not available. At the moment it is still only available in beta. We decided to target the latest stable version of Java available; 1.4.2.

5.5 Translating Java

How should chorded-Java be translated into normal Java? There are four stages to the process.

1. Parse the chorded-Java.

2. Work-out what structures present in the parsed data need to be translated.

3. Calculate the translation.

4. Output the results as Java.
5.6 Parsing the chorded-Java

5.6.1 Selecting a parser-generator

To parse a complex language like Java it is normal to use a parser-generator. The author had some experience with Antlr\cite{ANTLR}, and knew that there was a Java syntax file available for use with it. Antlr comes highly recommended and is native to Java.

One of the most attractive features of Antlr is its support for Tree Parsers. Once the chorded-Java has been parsed and tokenised it is arranged into a tree like structure. We can then use a tree-parser to collect information from the tree and to translate the aspects of the tree that need to be translated.

The tree-traverser is generated from a file similar to the parser generating syntax file, but with tree nodes instead of tokens. Each rule in the tree-traverser can contain arbitrary Java code. The tree can be re-arranged added to and pruned arbitrarily.

This turned out to be a very powerful mechanism for manipulating Java code.

The version of Antlr used was 2.7.2, which was the latest stable version available when the project started. Antlr is in active development, and has been for some time (it started in 1989), since the start of this project, versions 2.7.3 and 2.7.4 have been released. It is likely that Antlr will continue to be maintained for some time in the future.
5.6.2 Modifying the Grammar

Although the Java grammar included with Antlr was quite complete, it was necessary to alter some aspects of it. The main change was to how methods are recognised. In the original grammar the definition of a method was contained in the definition of field. The definition is largely in-line.

Listing 5.1: Original Method Grammar

5.6.2.1 Recognising Chords

A chord is quite similar to a normal method definition, except for the presence of multiple prototypes. At the highest level the structure of a chord is chordPattern chordBody. A chordBody is identical to a normal Java method body. A chordPattern is somewhat different to a normal Java method prototype though.

A chordPattern can be thought of as consisting of one or more prototypes separated by “&”.

public void a(int u, int v) & private async c(int w)
There is a slight complexity, which is that only the synchronous element of a chord can throw an exception. Given that at most one element of a chord can be synchronous; a fuller example of a chord might be:

```java
public void a(int u, int v) & private async c(int w) throws Exception {
    // body
}
```

The modified grammar captures this structure almost exactly. It matches a list of chord elements (elementList), followed by an optional throwsClause then finally a compoundStatement (which is the chord body). Matching this rule creates a tree node of type "METHOD_DEF", with children of type "ELEMENTS" (see section 5.6.2.2) and "THROWS" (un-modified from original grammar).

Note that the modifiers for the first element along with the return type of the first element are matched by this rule, and not by the elementList rule. This is necessary to distinguish this rule from other rules also in field, like innerclass (not shown).

These matched tokens are then passed to the elementList rule, so the proper tree nodes can be created for each chord element.

### 5.6.2.2 Recognising Element Lists

So, an element list is a set of chord elements, separated by “&” — but with the modifiers and return type already consumed from the first member of the set.

`elementList` becomes a fairly simple rule, which distinguishes the first element (with its missing components) from the subsequent elements. The subsequent elements are matched by the rule `chordComponent`.

---

1An exception thrown by an async channel would have no-where to go, the thread calling the async channel will be running in a different part of the program, or could possibly even have terminated.
Matching the elementList rule creates a tree node of type “ELEMENTS”, with a list of nodes of type “CHORD_ELEMENT” as its children.

```
getElementList[AST mods, AST type]
  : (e:element[mods, #type] (chordComponent)*)
  {#elementList = #(#[ELEMENTS,"ELEMENTS"],
      #elementList);}
```

Listing 5.5: Element Lists

Again, we pass the already matched modifiers and return type down to the next rule `element`.

### 5.6.2.3 Recognising Elements

The element rule is fairly straight forward. The most complex part of it is the construction of the “TYPE” node. The designers of the original grammar devised a way of representing types in the generated tree which was not easy to understand from reading the grammar file. Luckily Antlr comes with a visualisation tool which creates a graphical representation of the generated AST. This allowed us to understand how they “TYPE” node was constructed.

The complexity stems from having to represent array types, such as `Object[][]`. This would be represented as the (degenerate) tree “[[],Object]”. Once this was understood, the rest of the rule fell into place.

```
element![AST mods, AST type]
  : id:IDENT LPAREN! decls:parameterDeclarationList RPAREN!
    rt:declaratorBrackets[#type]
    {#element = #(#[CHORD_ELEMENT, "CHORD_ELEMENT"],
      mods, #(#[TYPE,"TYPE"], rt),id,decls);
    }
```

Listing 5.6: Elements

This listing shows where the modifiers and return type from the original match are finally added to the generated tree — they become children of the “CHORD_ELEMENT” node.

### 5.6.2.4 Recognising Chord Components

chordComponents are all of the elements in the elementList except for the first one (which is not preceded by an “&”). This rule took a few attempts to get right, until it was realised that a chord component was basically the element rule, prefixed with an “&”, a list of modifiers and a return type.

```
chordComponent!
  : BAND! m:modifiers t:typeSpec[false] e:element[#m, #t]
   {#chordComponent = #e;}
```

Listing 5.7: Chord Components
We match the “&”, and discard it (which is what “!” means in this context). Then match the modifiers and the return type. We pass the modifiers and return type to the element rule and return the node created by the element rule as the result of chordComponent.

This ability to return arbitrary tree-nodes from each rule is one of the key features of Antlr. It allows the user to separate out similar concepts into different rules, without being left with an excessive number of different types of nodes in the Abstract Syntax Tree (AST).

5.6.3 Objectives met

Since the changes to the Antlr grammar are fairly well restricted to changes to method recognition, it should be straightforward to port these changes to new versions of the standard Java grammar as they become available.

Of the changes planned for Java 1.5, the only one that should affect method recognition is Generics. This will require some minor changes, such as an additional optional node on type (representing the generic element) — these changes should be picked up automatically when the 1.4.2 changes are ported forward to the 1.5 grammar.

5.7 Tree translation with Antlr

Antlr supports tree walking and tree translation directly. It can automatically generate a TreeParser from a definition file, in a similar way to how it generates a parser from a grammar definition.

In our implementation we use two tree walkers. The first one collects information about the chorded-Java program, and translates the tree ready to be merged with the auxiliary code generated by velocity (see section 5.10). The second walker we use converts the resulting translated tree back into Java source code, and is described in section 5.8.

5.7.1 Collecting Information

The first tree parser collects information about the chorded program, then translates the tree to provide hooks into the user-defined logic. These hooks are used by the chord implementation code to call the chord bodies when they are selected for dispatch.

For example, the chorded-Java code (which is part of an implementation of the Dinning Philosophers):

```java
public process SimpleSeating implements Seating
{
    public SimpleSeating(final int seats)
    {
        available(seats, seats);
    }

    public void sitDown()
    & private async available(final int seats, final int empty)
    {
        if (empty > 1)
```
Would have each chord body replaced with a private method. The names of the methods are munged in a well-defined way, and then stored along with the information collected on each of the chords. The parameters of the separate chord elements are also combined at this stage.

Would have each chord body replaced with a private method. The names of the methods are munged in a well-defined way, and then stored along with the information collected on each of the chords. The parameters of the separate chord elements are also combined at this stage.

Listing 5.8: Example code for translation

Listing 5.9: Example code after translation
During the translation, all of the information about which chord body belongs to which chord pattern, and the configuration and type information are stored in a data structure. This structure is depicted in figure 5.2.

Figure 5.2: Overview of the model API

5.7.2 Collecting the information

As the tree walker reaches each node in the AST it collects information about the structure of the program. And sometimes replaces the existing node with a translated one.

In this example, a chord is recognised. A new compoundChord object is generated by calls to the API. Then each element of the chord is parsed by the tree parser (the rule for chord_element(child) follows this example). The resultant structure is then registered with the compoundChord. Finally a new "METHOD_DEF" node is created and substituted into the tree (this is the node containing the munged name and combined parameters).

```
methodDef !
  : #(METHOD_DEF e:elements (t:throwsClause)? (s:slist)?)
```
if (#e.getFirstChild().getNextSibling() != null)
{
    CompoundChord chord = currentGenerator().nextChord();

    if (t != null)
    {
        chord.throwsClause(chord.throwsClause(t), t);
    }

    AST child = e.getFirstChild();
    while (child != null)
    {
        chord.register(chord_element(child));
        child = child.getNextSibling();
    }

    #methodDef = #(METHOD_DEF, chord.getBodyElement(m_astFactory), t, s);
}
else
{
    [...] other stuff.
}

Listing 5.10: Information collection and translation

ANTLR’s ability to return arbitrary objects from each rule is seen again here. Instead of returning a tree node, the “chord_element” rule returns a Java Object called “ChordElement” (which is an implementation of the Element interface in the API, not shown in figure 5.2).

{
    e = new ChordElement(chord_modifiers(#m),
                         chord_type(#t),
                         chord_methodProto(#p));
}

Listing 5.11: Object creation in an ANTLR tree rule

The information collected by this tree parser is type checked (see section 5.9) then finally used by the Velocity powered chord implementation generator (see section 5.10) to produce the code performing the chord tasks.

The translated tree is printed by another tree walker, described in section 5.8, and combined with the generated implementation code.

5.7.3 Creating Nodes

During the translation process, it is necessary to create new nodes (to represent, for example, the munged method names).
Antlr allows arbitrary Java code to be mixed with the Tree-Walking syntax. This allows us to create data structures in Java then create nodes from these structures. For example, here is a method which creates a "CHORD ELEMENT" node:

```java
public AST createElement(final Modifiers modifiers, final Type type, final Prototype prototype, final AST throwsClause) throws SemanticException {
    AST params = createParameters(prototype.parameters(this));

    return #(#[CHORD_ELEMENT,"CHORD_ELEMENT"], modifiers.getNode(m_астFactory), createTypeNode(type), prototype.nameNode(this), params, throwsClause);
}
```

Listing 5.12: Creating a node

Note the use of # to create a new node in the AST, combined into a normal Java method. The tree translation is driven by a combination of Antlr tree walking and node manipulation syntax, and Java code which interfaces to the data structures created during the tree walk.

5.8 Tree printing with Antlr

It should be possible to get Antlr to reverse the parsing process and replace each node in the tree with the syntax tokens representing it. Unfortunately this requires the grammar to be constructed in quite a fiddly and involved manner. The process is somewhat error-prone.

The process is difficult to debug, since, when the printing processes is not working properly it produces invalid output. This output is often all on one line and, since it is invalid, Java code formatters refuse to format it. We found the whole process frustrating, and annoying. It would probably have been better to start on a simpler grammar than the whole of the Java syntax, to learn exactly how the process was supposed to work.

In the end an alternative approach was used. We created a second tree-walker which visited each node in the tree and converted it into its string representation. This process was a little tedious, but in the end produced a more flexible solution. It was easy to tweak the rules in the tree-printer to produce output in a particular style.

The tree printer is fully encapsulated, so could be used separately from the rest of the project code — should the need to print out an Antlr Java AST arise in the future.

It would be an interesting project to turn the tree-printer into a full blown Java pretty printer / source-code formatter. This would probably only be a couple of weeks work and would primarily involve adding a user configuration file to allow the user to choose the style of the output.

There seems to be a lack of native-to-java, Java source code formatters at the moment. Several of the free ones (like Jalopy) have recently stopped being free. And some of the best ones (like
JIndent) were never free in the first place. The author used the code-formatter in the Eclipse IDE (which is a little lacking) and a program called “astyle” which is written in C++ (uggh), whilst writing the code for this project.

5.9 Type checking

Once the information from the chorded-Java program has been collected, it is necessary to check the typing of the chords is valid. Several type rules are checked, most of which are the same (or Java versions of) the C♯ rules, and match the description of chorded programming given in chapter 3:

- Chords can have at most one synchronous element.

- Each occurrence of a chord-element must have exactly the same parameters.
  
  This includes both the name of the parameters and the types. Although it is not strictly required that the names be the same, since each chord-element is reduced to a single method (see section 6.4), and the names of method parameters are visible (in the Java-doc for one) it was thought prudent to enforce this rule.

  The types must be the same, since they determine the type of the message received on the channel represented by the chord-element. And each channel can only receive messages of a single type.

- Each occurrence of a chord-element must have exactly the same return type. Again, since each chord-element maps onto a single method there is no other possibility.

  Since which chord matches each time is non-deterministic, it would make no-sense to have the return type depend on which particular chord matched, and the Java language doesn’t support variable method return types even if we wanted to provide this feature.

- If a chord-element appears in a chord, it cannot also appear by itself as a method.

  This rule tries to stop the programmer creating unreachable code. If a chord-element appears by itself, every message sent to it would be consumed immediately. Chords which required more messages would never be dispatched.

  This is not a rule of chord programming in general, but is an artifact of the unfair nature of the dispatch algorithm. In particular, if this were not the case then normal method calls to any object which had chords would have to be run through the chord dispatch algorithm. The chord dispatch algorithm is of a higher-complexity (and slower) than normal method dispatch.

  If a fair dispatch algorithm were used, this rule would not be required (although some method calls might be slowed down)

  See section 6.5.2, for more on fairness.

If a type error is found, an error messages is output and the compile fails.
5.10 Using Velocity to generate the auxiliary code

One of the objectives of the implementation is to allow the generated classes to run without any external support — no library or auxiliary classes which have to be distributed along with the generated class.

A second objective was to have easy support for changing the way that chords are implemented. So that new algorithms could easily be added / old algorithms easily be modified.

5.10.1 What is Velocity?

Velocity is the Java template engine from the Apache project. It provides a powerful language for creating templated output. It was designed as a replacement for JSP or PHP based web-sites, supporting strong separation of implementation and presentation. JSP and PHP web sites tend to have program elements, like database access, mixed in with presentational elements, like code to position images. Velocity separates these two concerns.

This strong separation of presentation and implementation is an example of the MVC (model-view-controller) design pattern[6]. The usefulness of this software was soon recognised, and a diverse array of secondary applications began to emerge. Automatic generation of SQL, Postscript and XML alongside more traditional report generation. It also emerged that velocity was a viable mechanism for automatically generating source code from higher level representations.


5.10.2 Alternatives

There are several alternative approaches which could have been followed.

5.10.2.1 First Alternative — using Antlr

The most obvious approach would have been to generate all of the auxiliary code by building extra branches in the AST.

This approach would have reduced the number of libraries needed by the compiler, and was initially tried. Several problems soon became apparent though:

- No API. There is no specific API available for creating nodes from Java. The only way to create nodes is to use the constructors for the objects which represent the nodes, then set all of the attributes and children of the nodes “by hand”. This takes a long time and involves a large amount of code.

We did look into the possibility of creating such an API, but due to the complexity of the internals of Antlr, and the fact that the design and construction of such an API has virtually nothing to do with the main features of this project, it seemed inappropriate.
• Long. Because of this lack of API, the code to implement even simple expressions was huge. An expression like

\[
x \cdot \text{setZ}(y \cdot \text{a(new } F(x) \{ \text{int } b() \{ \text{return } y \cdot l(); \} \}))
\]

Listing 5.13: Weird statement

could run to hundreds of lines. Not very practical.

• Maintainability. Due to this lack of abstraction, simple changes to the output code required large changes to many different areas of the generating code. This was directly counter to our objective of supporting new techniques for implementing chords.

• Reusability. Without building the specialist API, the created code was almost totally once only. Having a second implementation (for instance supporting a different algorithm) required writing almost as much code as the first one.

5.10.2.2 Second alternative — inline

The second alternative tried was to generate the code as strings inline in the Java code. For example, building a method header would look like this (except, probably using “java.lang.StringBuffer” for efficiency);

```java
String buildPrototype(String[] modifiers, String returnType, String name, String[] parameterTypes, String[] parameterNames)
{
    String result = join(modifiers, ",");
    result = result + "\n" + returnType + "\n" + name;
    result = result + "(" + join(
        merge(parameterTypes, parameterNames, new Joiner()
        {
            String join(String first, String second)
            {
                return first + "," + second;
            }
        }), ",");

    return result);
}
```

Listing 5.14: Inline method header

Not totally illegible, but again requiring allot of code to support all of the features of Java. The code which generates the actual resultant code was a mishmash of strings, configuration variables and methods calls.

To make this solution preform in a reasonable manner, StringBuffer would have to be used. This creates even harder to read code.
There are very few (in fact, none which are free and currently maintained) ready made solutions for generating Java from Java. I expect most people who need to generate Java code, use this inline approach.

### 5.10.3 How velocity solves this problem

Velocity is a macro-language, it allows inline expansion of arbitrary macros. It provides access to Java objects through a simple API.

An example will illustrate best. Here we look at how building a method prototype would be expressed in Velocity. We use the MVC approach, since this is the primary mode of operation of velocity.

#### 5.10.3.1 Model

We designed the following simple interface to represent an element. The implementation of this interface is provided by the data source (in this case the information calculated from the AST).

```java
public interface Element {
    /**
     * @return A string representation of this type.
     */
    String getType();

    /**
     * The name of this element.
     */
    String getName();

    /**
     * The collection of the Parameters of this element.
     */
    Collection getParameters();

    /**
     * The modifiers for this element.
     */
    String[] getModifiers();

    /**
     * @return True iff this element has more than zero parameters.
     */
    boolean hasParameters();
}
```

Listing 5.15: A simplified method element
### 5.10.3.2 Controller

The controller sets up the data structure for use by velocity. In this example, a collection of elements is added to the velocity context under the variable name “elements”.

```java
context = new VelocityContext();
context.put("elements", m_elements.getElements());
```

Listing 5.16: Setting up the context

### 5.10.3.3 View

The code generation is now totally separated from the data source. The velocity engine takes a template and the context created by the controller and merges the two to obtain the output code.

The template contains a series of macro expansions and inline code which expand to some function of the context. As an example, here are the macros we wrote to create the code for a method header from the element presented above.

```java
#macro (printElementHeader $element)
#printModifiers($element.modifiers) $element.type $element.name (  
  #printParameters($element.parameters))
#end
```

Listing 5.17: Velocity Macros for a method element

The interpretation of the macro is quite literal. It prints the modifiers, followed by the type, followed by the name of the element and finally the parameters surrounded by brackets.

Some supporting macros are also required; one to print the parameters correctly, and the other to print the modifiers. The first one has an interesting feature, ${parameter.print()} calls the Java method “print()” of the parameter. In this way, the expressiveness of velocity can be combined with the power of the Java language.

```java
#macro (printParameters $parameters )
#set ( $tmpSep = "")
#foreach ($parameter in $parameters )
  ${tmpSep}$parameter.print()#set ( $tmpSep = ", " )#end
#end

#macro (printModifiers $modifiers )
#foreach ($modifier in $modifiers ) $modifier #end
#end
```

Listing 5.18: Supporting macros

However, velocity only really comes into its own when a complicated example is looked at. Here is the velocity template that generates a queue class (which are discussed in section 6.3).

```java
#if ($element.hasParameters())
  private class Que$[element.mask]
```

60
{  
    private ArrayList m_list = new ArrayList();
    public Object[] deque()
    {  
        return new Object[]{
            #set($tmpSep="")
            #foreach($parameter in $element.parameters)
                $tmpSep m_list.remove(0)
            #set($tmpSep = ",")
            #end
        };
    }

    public void enqueue(#printParameters($element.parameters true))
    {
        #foreach($parameter in $element.parameters)
            m_list.add($parameter.box("$parameter.name"));
        #end
    }

    public boolean empty()
    {
        return m_list.isEmpty();
    }
}  

#else
    [... some other stuff)
#end

Listing 5.19: Queue Template

The representation of the queue class is quite literal, looking much closer to the final output than either of the two alternatives discussed earlier\(^2\).

### 5.10.4 Velocity Version

The latest stable version of velocity available when the project started was version 1.3.1. Since then a major release 1.4 has occurred. The changes are largely in the introspection support, so have no direct effect on our implementation. However, it is possible that these new features could be used to improve the way velocity is used in our implementation.

Velocity is in active development by a large community of programmers and users, as such I expect it to continue improving and to be supported for some time.

### 5.10.5 Objectives met

Using Velocity allowed us to abstract the generation of the implementation code from the parsing and manipulation of the input code. By transforming the input code in a well defined way, then providing an API describing the results of the transformation to the velocity template, it is possible to restrict the entire description of the chord implementation algorithm to the velocity template.

\(^2\)# is a call to a builtin command or user defined macro, whilst $ dereferences a variable.
Figure 5.3: Overview of implementation plugability
As shown in figure 5.3, this allows the chord algorithm, used to implement chord support, to be plugged into the chorded-Java to Java compiler. Because the plugin is simply a Velocity template it is even possible to select or alter the implementation at run time.

Not only does this offer the user increased flexibility, it also speeded up the development process massively; since adjusting the chord implementation algorithm just required alteration to the velocity template (a text file), no recompilation between changes was required.

5.10.5.1 Potential Problems

It is possible that future chord algorithms will require a different representation of the user part of the logic, or more information from the model API. This could require modification of these components, or in the case of a significant change in the method of implementing chords, an architectural revision. Such a change is not anticipated, but if it occurred might require significant reworking of the system.

5.11 Conclusions

We designed and implemented a framework for a chorded-Java to Java compiler (or pre-processor). We used Antlr to parse and translate the chorded-Java, and again to output the results. Velocity was used to provide a pluggable API for creating chord-implementation code. Two such implementations were created and are described in chapter 6, and subsequent chapters.

The pre-processor works well. It supports all of the basic features of chords, the implementation of which will be discussed in the next few chapters.

The changes to the Antlr grammar are localised so should be easy to port to a Java 1.5 grammar when it becomes available. Both of the technologies used in the pre-processor are in active development and are likely to continue to be supported.

Now that the framework is working it could be used as a reference for implementing a direct to byte-code compiler. This would eliminate the potential problems that pre-processors introduce, particularly the extra level of indirection between the executed byte-code and the program as seen by the developer.

The use of Velocity allows rapid development of implementations of the auxiliary code which implements chords, by removing the change-compile-run cycle. It also means that different implementations of the auxiliary code generator can be plugged into the system and use the hooks into the parsed code provided by the pre-processor.
Chapter 6

Implementation Issues

To invent, you need a good imagination and a pile of junk. — Thomas A. Edison
(1847 - 1931)

6.1 Introduction

We would like to be able to implement chords in a fast, simple and correct manner. In this, and subsequent chapters, we develop two implementations of chords for Java, one (called the simple algorithm) derived from our formalism, the other (called the polyphonic algorithm) based on the algorithm in [1].

In this chapter we take several aspects of our formal description of chorded programs, and develop them into implementable ideas. We look at how the message queues can be implemented, then how channels are implemented, and how messages are put into the queues.

The complexity of the operations on the message queues is also calculated. Three different queues are developed and evaluated, a null-message queue, a non-null-message “queue” and a queue for marshaling synchronous threads.

We then use our formal description to clearly define what constitutes a pattern match, and how pattern matches are chosen. We show how this formal description of pattern matching can be expressed as bit-vectors and give code (similar to that described in [1]) to implement and maintain the state of these bit vectors. The pattern match code is also developed, although the algorithm for selecting a match is presented in a subsequent chapter. Finally we look at the complexity of some parts of the pattern matching algorithms.

The formalism, complexity calculations and other discussion in this section is entirely that of the author’s. The simple implementation of chords is also the author’s, although some elements of it, such as the message queues and the status vector are similar to [1].

6.2 Implementation Overview

Objects in chord languages can be though of as a set of pending messages. In fact, we can have more than one message pending on each channel — so we have a set of lists of pending messages. More formally, an object has
object; = \{q_1, \ldots, q_m : mQueue\}^t
mQueue : list of messages

How can we implement message queues? And how fast is the implementation?
The current state of the object may match one or more chords. If it does match, then one of
those matched chords must be dispatched. How should we identify those chords which can be
dispatched? How quickly can we do this in Java? And how should we choose between multiple
dispatchable chords?

6.3 Implementing message queues in Java

There are essentially two types of message queue in our implementation. The first type of queue
is used for chord elements which do not have any parameters. In our formalism messages on these
channels are represented by the “null” message.
The second type of queue is used when the chord element has parameters. It has to store each
message so that the value of the parameters can be retrieved when the chord is dispatched.

6.3.1 Null message queues

Since the “null” message is always the same, there is no need to store multiple copies of it; instead
it is possible just to count the number:

```java
class NullQue {
    private int m_count = 0;
    public void dequeue ()
    {
        m_count --;
    }
    public void enqueue ()
    {
        m_count ++;
    }
    public boolean empty ()
    {
        return m_count == 0;
    }
}
```

Listing 6.1: Null Message Queue

6.3.2 Null message queue complexity

The complexity of adding and removing messages from a null message queue is low.
• Checking for emptiness is a single operation (comparison of an integer to zero).

• Adding a message to the queue is a single operation (incrementing an integer).

• Removing a message from the queue is a single operation (decrementing an integer).

So the complexity of all operations on this queue is $O(1)$.

### 6.3.3 Non-null message queue

These queues are used to store messages which have parameters.

We need to store each set of parameters (message) that arrives, ideally, so that the message can be recovered in a fairly efficient way. In the spirit of not re-inventing the wheel we will take advantage of an existing structure in the Java standard library; “java.util.ArrayList”. In particular it can be used as a double ended queue.

Normally, in Java, method parameters are passed using a stack. We use this same idea to store the parameters from each message. Remember, we have one queue for each channel in the object.

Take, for example, the following chord;

```java
public void standUp()
& private async available(final int seats, final int empty)
{
    available(seats, empty + 1);
}
```

Listing 6.2: Channel with parameters

Consider the channel “available”. A call to that method would, in a normal OO language, push two integers onto the stack (amongst other things). We mirror this structure by creating a stack of stack-frames to store an image of the relevant portion of stack ready for when we come to execute the method later;

Figure 6.1 shows the saving of parameters when the message arrives. There is no need to copy the receiver since the queue is for a single object only. There is no need to save the space for the local variables, since it will be incorrect anyway — when the method runs it will run on some body defined by the chord, not the channel.

Figure 6.2 shows what happens at chord dispatch. The receiver and space for local variables will be re-created separately. Dispatching a chord will often mean obtaining parameters from several queues (since each channel has a separate queue).

### 6.3.3.1 Boxing and Unboxing

Unfortunately, in Java 1.4.2, “java.util.ArrayList” cannot store primitive types such as “int” directly. To store a primitive type it must first be transformed into its object type.

For example, here is a queue which performs this operation for the channel seen in listing 6.2:
Figure 6.1: Message Arrival

Figure 6.2: Chord Dispatch
private class NonNullQue
{
    private ArrayList m_list = new ArrayList();
    public Object [] deque()
    {
        return new Object[]{
            m_list.remove(0),
            m_list.remove(0);
        }
    }
    public void enqueue(final int seats, final int empty)
    {
        m_list.add(new Integer(seats));
        m_list.add(new Integer(empty));
    }
    public boolean empty()
    {
        return m_list.isEmpty();
    }
}

Listing 6.3: Message Queue with parameters

In Java 1.5 this process will be automated by the compiler.

6.3.4 Non-null message queue complexity

Another reason for the choice of “java.util.ArrayList” as the implementation of the double-ended-
queue is that it has well defined complexity.

• Checking for emptiness runs in constant time.
• Adding a message to the queue depends on the number of elements in the message. The
  complexity is linear in the number of parameters in the message.\footnote{The size, isEmpty, get, set, iterator, and listIterator operations run in constant time. The add operation runs in amortized constant time, that is, adding n elements requires O(n) time. All of the other operations run in linear time (roughly speaking). The constant factor is low compared to that for the LinkedList implementation.\cite{15}}.
• Removing a message from the queue depends on the number of parameters in the message, and runs in linear time.

Creating space on the stack runs in constant time (simply moving the stack pointer).
So the overall complexity of storing a message and recreating the stack is O(n).

6.4 Marshalling message arrival – implementing channels

In our Java implementation of chords, messages arrive by means of a method call. Each channel
that an object can receive messages on is implemented by a method.
As far as marshaling message arrival is concerned, there are two types of channels in our implementation; “async” and synchronous channels. Our formalism does not distinguish the two, it queues and dequeues messages for all channels in the same way. However, we will see that by considering these two types of channel differently we can make significant improvements in the efficiency of our implementation.

6.4.1 Marshalling async channels

Marshalling message on async channels is very straight-forward. It is simply a case of adding the message to the relevant queue. For instance, consider the private async channel;

```java
private async fullUp ( final int seats )
{
    lock();
    que8.enqueue(seats);
    [...] // dispatch code, explained later
    unlock();
}
```

Listing 6.4: Marshalling messages on async channels

The message is queued and the thread that called the method (sent to the channel) is able to carry on executing. We uses an unstructured form of locking to regulated the atomicity of message marshaling, see chapter 7 for more details.

6.4.1.1 Complexity

Marshalling an async channel consists only of putting the message in the queue, which we have already shown has a maximum complexity of $O(n)$.

6.4.2 Marshalling sync channels

Channels which are not async are synchronous. Synchronous channels block the sending thread until the sent message is consumed. This is the first place that the simple and polyphonic algorithms significantly differ.

6.4.2.1 Simple algorithm

In our simple algorithm we queue the threads and the message in separate queues. The threads in the thread queue wait until all of the message for a chord they are involved in are ready. When they are a MethodPointer object is passed, containing all of the messages and a reference to the body to be executed. The threads compete to obtain one of these closures and execute it.
private class ThreadQue8
{
    private ArrayList m_messages = new ArrayList();

    public synchronized _MethodPointer8 que()
    {
        while (m_messages.isEmpty())
        {
            try
            {
                wait();
            }
            catch (InterruptedException e){}
        }
        return (_MethodPointer8) m_messages.remove(0);  
    }

    public synchronized void release(_MethodPointer8 pointer)
    {
        m_messages.add(pointer);
        notifyAll();
    }

    public synchronized boolean empty()
    {
        return m_messages.isEmpty();
    }
}

Listing 6.5: Simple: marshaling messages on sync channels

There is only a single lock involved in this thread queue, but the separate lock for each queue allows us to wake a minimal number of threads when a chord becomes available for dispatch.

6.4.2.2 Polyphonic C♯

In the algorithm presented in [1], we can take advantages of the resources of the blocked thread to store the message parameters without the use of a queue.

The idea is to queue the thread, rather than its message. In this way the message (and any parameters) can be held on the stack of the thread (which is where they are when the method is called). This saves us the operation of adding and removing the messages from the queue. It also means we only need one queue (of blocked threads), instead of two queues (one of blocked threads, and one of messages).

The queue used is similar to the null message queue, but instead of counting the number of null messages, it blocks the threads which enqueue themselves, unblocking them when a call to dequeue comes in.

private class ThreadQue
{  
    int m_count;
    boolean m_signalled = false;

    int m_count;
    boolean m_signalled = false;
}
// blocks the calling thread until released
public void que()
{
    m_count++;  // increment the count of threads
    unlock();  // release object lock

    synchronized(this)  // lock queue
    {
        while(!m_signalled)
        {
            try
            {
                // wait until signalled
                wait();
            }
            catch(InterruptedException e)
            {
            }
        }
        // this thread has won the race to be signalled
        m_signalled = false;
    }

    lock();  // obtain the object lock
    m_count--;  // decrement the count of threads
}

// release one of the block threads from this queue
public void release()
{
    synchronized(this)
    {
        // signal a thread to be released and
        // notify all waiting threads.
        m_signalled = true;
        notifyAll();
    }
}

public boolean empty()
{
    return m_count == 0;
}

Listing 6.6: Polyphonic: marshaling messages on sync channels

This code is fairly straight forward, apart from the uses of unstructured locking. There are two
locks involved in synchronising this thread queue, the unstructured locking forms part of our main
dispatch algorithm and is examined in chapter 7. The normal looking synchronisation is the lock
that the threads are blocking on. This use of a separate lock for each thread queue allows us to
wake up the minimal number of threads (only those threads queued on the relevant channel) during
dispatch.
6.4.2.3 Complexity

The complexity of the queueing of synchronous threads is \( O(1) \). Each thread has to perform the same operations regardless of how many threads are already waiting. The thread increments a number, then performs a “wait()” operation. This is a four-way trade off between enqueue complexity, dequeue complexity, memory usage and the accuracy of the queueing. We have opted for decreased memory usage and low enqueue complexity.

The queue as implemented is not FIFO as in the other queues presented in this chapter. The threads are released from the queue in an arbitrary (but fair due to the properties of our scheduler, see appendix C) order. This prevents us from having to maintain a structure to queue the threads in order (which would require an additional lock object for each thread, and another \( O(n) \) queue to hold them in). As will be seen, this does lead to a potential performance problem when threads are released from the queue; all of the waiting threads are awakened and have to contended a single lock to try and be the thread released. This is a potentially slow operation — highly contended locks in Java can preform badly. However, we will see that the polyphonic dispatch algorithm is deliberately biased in favour of dispatching the chords for which the synchronous thread currently has control (as suggested in [1]). So big queues of threads are less likely to build up.

The complexity of the dequeue operation depends on the implementation of locks in the underlying Java system, particularly how efficiently contended locks are implemented. But it is at least \( O(n) \) where \( n \) is the number of queued threads (each thread gets woken up and each, except the one which is released) has to perform a wait operation.

6.5 Pattern Matching — identifying dispatchable chords

The class \( t \) contains a set of patterns corresponding to the channels needed to trigger a particular chord. Each pattern describes the set of channels which must have a message available before the body that pattern is guarding can be dispatched.

We can obtain the set of patterns in an object \([q_1, \ldots, q_m : mQueue]^t\) by using the, previously defined, \( patterns(class(t)) \). That is we can calculate the set of sets of channels which must have a pending message for any chord to be dispatchable.

So now that we have the set of possible matches, and the set of message queues, we need to define what constitutes a match. How can we reduce the set of possible matches to the set of actual matches?

First we have to turn our set of queues into the set of channels with at least one pending message:

\[
\text{channels}(\text{class}(t)) = \{c_1, \ldots, c_m : \text{Channel}\}
\]

\[
\text{queues}(\text{object}_i) = \{q_1, \ldots, q_m\}
\]

\[
\text{pending} : \text{Object} \rightarrow \mathcal{P}\{\text{Channel}, \}
\]

\[
\text{pending}(\text{object}_i) = \{c_i | c_i \in \text{channels}(\text{class}(\text{object}_i)) \land (m_i \in \text{queues}(\text{object}_i)) \implies \neg(\text{empty}(m_i))\}
\]

We know from well-formedness and the definition of object that all \( c_i \) will have a corresponding \( m_i \) and that all patterns will use a subset of the channels in \( \text{channels}(\text{class}(t)) \). So if \( c_i \in \text{channels}(\text{class}(\text{object}_i)) \) then it must be the case that \( m_i \in \text{queues}(\text{object}_i) \).
Now that we have the set of channels with at least one pending message, we can use it to reduce
the set of patterns in the class into the set of dispatchable patterns.

\[
\text{dispatchable} : \text{Object} \rightarrow \mathcal{P}[^\text{Pattern}]
\]

\[
\text{dispatchable(object}_i\text{)} = \{ p | p \in \text{patterns}(\text{type(object}_i\text{)}) \land (p \subseteq \text{pending(object}_i\text{)}) \}\]

With this set of dispatchable patterns we can choose a pattern to dispatch, that is pass the pending
messages into the scope of the code corresponding to the pattern and then execute that code. When
the pattern is dispatched the messages needed are consumed and removed from the queues.

### 6.5.1 Atomicity of dispatch

In order for our pattern matching to be implementable, the match/dispatch step must be atomic.
No additional messages may arrive, and no dispatch may occur whilst matching and dispatching
a chord.

For now we will assume an interleaved model of concurrency which makes these steps atomic.
Later we will look at how to achieve this in a genuinely concurrent (parallel execution) system,
such as Java or C\# (see chapter 7).

### 6.5.2 Fairness

When multiple patterns are available for dispatch we must choose one, since when one is dis-
patched it is possible that the consumed messages will prevent any of the others being dispatched.
There are essentially two choices, to use a fair algorithm or an unfair one. We define fair and
unfair as:

**Fair:** given an infinite number of messages in each queue, if we dispatch an infinite number of
chords each chord will be dispatched an infinite number of times.

Note that this definition assumes that we have an infinite number of messages in each queue. In
practise this is unlikely to happen, but we make the assumption anyway to test our algorithm.
Unfair is the converse of the definition of fair, any correct algorithm for dispatch which is not fair
is unfair. However, unfair could be stated as;

**Unfair:** given an infinite number of messages in each queue, if we dispatch an infinite number of
chords at least one chord will be dispatched less than an infinite number of times.

When we come to do reason about chorded programs we will have to specify the properties of
our dispatch algorithm. It is important that the algorithm implemented matches the algorithm
analysed. Programs which behave correctly under a fair dispatcher may not be correct under an
unfair dispatcher.
6.5.2.1 Fairness in Java and Polyphonic C♯

Neither Polyphonic C♯ nor our implementation of chords for Java uses a fair matching algorithm\[^1\]. Both always use the first match they find. This is the fastest match choosing algorithm\[^2\], although our match finding algorithm might not be the fastest possible.

6.5.3 Determinism

Since the order of scheduling in a concurrent system is non-deterministic it makes little difference whether our dispatch algorithm is deterministic or not (the message order will already have made the observed dispatch order non-deterministic). Additionally, the execution of a chord body can happen any (finite) time after the dispatch (according to the scheduling algorithm). Again, the determinism or otherwise of the dispatch choice will not effect this.

6.5.3.1 Determinism in Java and Polyphonic C♯

Both Polyphonic C♯ and our implementation of chords for Java use a Deterministic dispatch algorithm. For any given set of queues the first match will always be chosen and dispatched. However, both Java and Polyphonic C♯ have essentially non-deterministic schedulers. So this determinism does not show (other than how it manifests in the unfairness mentioned above).

6.6 Encoding “Or” using “And”

Patterns, as we have defined them so far, only deal with a conjunction of channels (a match where all of the channels must have messages). In our general model of chords we also have disjunction (where a match may draw a message from either of two channels). We show here that our reduced matching algorithm (with only conjunction) can encode the full matching algorithm (conjunction and disjunction).

Given three channels \((c_1,c_2,c_3)\) and the pattern \(c_1&(c_2|c_3)\), the possible matches for this pattern are \((c_1,c_2)\) or \((c_1,c_3)\). If we think of these two matches as patterns themselves we see that they are only matched by \((c_1,c_2)\) and \((c_1,c_3)\) respectively.

We can see that \(|\) distributes over \&, that is \(c_1&(c_2|c_3)\) is equivalent to \((c_1&c_2)|(c_1&c_3)\). Our matching algorithm will match the disjunction of the patterns in the class. So we can match \((c_1&c_2)|(c_1&c_3)\) simply by adding both \((c_1&c_2)\) and \((c_1&c_3)\) as patterns in the class, both pointing at the same body.

6.6.1 Typing and naming in disjunction patterns

Whilst implementing chords for Java the disjunction pattern can be supported in a nice way, although it does have one subtlety. When a Java method body executes, the number and type of its parameters are fixed. Since our chord bodies are implemented as Java methods, and use standard

\[^1\]By definition. If a faster algorithm were invented, which could find two (or more) matches and pick fairly between them, then it could also be used to implement our unfair match algorithm — by arbitrarily choosing one of them as the first match.
Java syntax, the same restriction must apply to the compound method headers which represent chords.

As an example, the following chord would not be desirable in our language:

```java
public string get() &
(public async puts(string s) | public async put(int t))
{
    [...]
}
```

Listing 6.7: Bad Chord

The primary reason for disallowing this is that there is no intuitive way of distinguishing the presence or absence of a variable in Java. Using “null” is not appropriate, since that is a legitimate value for a variable.

Instead we introduce the restriction that the parameter types and names must be exactly the same. In this manner we ensure that each variable which could be referenced in the body of the chord will always be well defined when the chord is dispatched.

```java
public void put(Object object) &
    private async empty(int start, int stop, Object [] messages)
| private async part(int start, int stop, Object [] messages)
{
    [...]
}
```

Listing 6.8: Good Chord

Disjunction is not currently supported.

### 6.6.1.1 Possible improved syntax for disjunction

There is a certain amount of redundancy in the syntax described above. An improved syntax might take advantage of the observation that the disjunction really only applies to the name of the channel and not to anything else. It might be possible to reflect this, for instance:

```java
public void put(Object object) &
    private async empty | part (int start, int stop, Object [] messages)
{
    [...]
}
```

Listing 6.9: Better Chord?

Which removes the redundant information, and more accurately reflects the conceptual meaning of `|`.

### 6.7 Efficient encoding of subset operations

As shown in [1], we can easily encode the pattern matching as simple bitwise operations on bit-vectors.
6.7.1 Why bit-vectors

Bit-vectors are used because finding a match between two bit-vectors can be reduced to some very fast, fixed time, bitwise operations. Since the complexity of chord-dispatch, and hence, the complexity of algorithms implemented in our language is largely dependent on the efficiency of our matching algorithm, this is important.

In Java, method calls are very fast (once the initial loading and linking is out of the way). When implementing an algorithm in Java it isn’t normally necessary to think about the complexity of method calls. The objective is to have a similar situation for chords in our implementation for Java.

Chord dispatch will probably always be more complex than method dispatch, firstly because there is inherently more information to process during a chord dispatch than a method call, and secondly because chord dispatch involves complex synchronisation conditions which method calls don’t.

For these reasons, combined with the potential high number of chord dispatches a program will undertake, it is important to make the pattern matching as efficient as possible.

6.7.2 Mapping sets to bit-vectors

We map the set of channels in a class onto a bitvector. Given:

\[ \text{channels}(\text{class}(t)) = \{c_1, \ldots, c_m : \text{Channel}\} \]

we define a bitvector \((b_1, \ldots, b_m : \text{Bit})\) for each class such that \(b_i\) corresponds to \(c_i\).

Each object \(i : \text{Object}\) where \(\text{class}(/i) = t\) has a state bitvector

\[ \text{statusvector} : \text{Object} \rightarrow \text{Bitvector} \]

\[ \text{statusvector}(i) = (s_{b1}, \ldots, s_{bm} : \text{Bit}) \]

the value of the \(i\)th bit is defined as;

\[ s_{bi} = 1 \iff c_i \in \text{dispatchable}(i) \]

Each pattern can also be encoded as a bitvector. For each \(p \in \text{patterns}(\text{class}(i))\) there is a bitvector \((p_{b1}, \ldots, p_{bm} : \text{Bit})\). The value of the \(i\)th bit is defined as;

\[ \text{patternvector} : \text{Pattern} \rightarrow \text{Bitvector} \]

\[ \text{patternvector}(p) = (p_{b1}, \ldots, p_{bm} : \text{Bit}) \]

\[ p_{bi} = 1 \iff c_i \in p \]

A match between a pattern vector and the status vector is defined by a simple boolean expression. For object \(i\), and some \(p \in \text{patterns}(\text{class}(i))\) the bitvector \(pb = \text{patternvector}(p)\) matches the status bitvector \(sb = \text{statusvector}(i)\) if and only if

\[ \text{matches} : (\text{Bitvector}, \text{Bitvector}) \rightarrow \text{boolean} \]
matches(pb, sb) \iff \left( \forall p_{bi} \in pb : (p_{bi} = 1 \iff \exists s_{bj} \in sb : j = i \land s_{bj} = 1) \right)

Or in a more recognisable Java like language

\(((\sim (sb)) \& pb) == 0\)

Where \(\sim\) is bitwise inverse and \& is bitwise and. As an example take the status vector 0, 0, 1, 1, 1
and the pattern 0, 0, 1, 1, 0; the inverse of the status vector is 1, 1, 0, 0, 0. If we interleave the vectors
to get pairs (0, 1), (0, 1), (1, 0), (1, 0), (0, 0), preforming the bitwise-and we get (0, 0, 0, 0, 0) which
tells us that it is a match.

### 6.7.3 Object Pattern Vectors

The pattern-vectors of an object are the bit-vectors representing each pattern of channels in the
class of the object. These bit-vectors are fixed, and are used in the pattern matching algorithm
described in section 6.5.

\[
\text{patternvectors} : \text{Object} \rightarrow \mathcal{P}(\text{Bitvectors})
\]

\[
\text{patternvectors}(\text{object}_i) = \{ pv | p \in \text{patterns}(\text{class}(\text{object}_i)) \land pv = \text{patternvector}(p) \}
\]

### 6.7.4 Implementing Bitvectors in Java

There are several ways to implement bit-vectors in Java. The simplest, most efficient way, is just to
use a number, such as an integer. The problem with using an integer is that it has a limited (fixed)
size. The compiler was implemented to use an integer, so has a built-in check to stop code with
too many channels being compiled. This was felt a good compromise since creating classes with
large numbers of methods is generally considered bad practise[9] and should be avoided anyway.
Still, having such arbitrary limits in a language should be avoided where possible.

The author has since learnt of the existence of “java.util.BitSet”, which automatically adjusts its
implementation depending on the required size of the bit-vector. It might be possible to use this
class instead of the integer currently in-use and then the restriction could be removed. This is an
area for further work.

The following code is currently used to implement the statusvector of objects (again it is similar
to the code in [1], but bit-vectors are a basic data structure, in the same way that a linked-list or
two’s compliment number are).

```java
int m_mask = 0;

public void set(int mask) {
    m_mask |= mask;
}

public void clear(int mask) {
    m_mask &= ~mask;
}
```
6.8 Implementing the pattern matching algorithm

Here we give pseudo-code for a simple pattern matching algorithm using the ideas above. This algorithm will find the set of dispatchable patterns for object $object_i$.

```java
foundMatches = ∅
foreach $pv \in patternvectors(object_i)$
{
    if (matches($pv$, statusvector($object_i$)))
        then $pv \in foundMatches$
}
```

6.8.1 Selecting a match

For now we will simply choose the first $pv \in patternvectors(object_i)$ which matches in the above loop. This is the fastest algorithm (since it can terminate as soon as the first match is found). This algorithm is not fair, since it is possible for an earlier pattern to always match before a later one (the later one never getting dispatched).

This starvation problem could be detected by checking that no later $pv \in patternvectors(object_i)$ is a subset of an earlier $pv \in patternvectors(object_i)$. Future work on the compiler could implement this.

6.8.2 Maintaining the Vector

There is an efficiency problem in the way that the status vector is defined. In our model the vector is calculated from the length of the message queues. However, since our state changes are atomic it is possible to maintain the vector without scanning the queues each time.

When a message is received on channel $c_i$, for $sv = statusvector(object_i)$ we can simply set the bit corresponding to the channel. ($b_i \in sv \rightarrow (b_i = 1)$).

We also have to clear the vector when the queues become empty. Queues only decrease in length when a message is dispatched. So we can apply the following action at that point.
When a pattern \( \{c_1, \ldots, c_n\} \) is dispatched from \( [q_1, \ldots, q_m]^t = \text{object}_i \);
\[ sv = \text{statusvector}(\text{object}_i) \]

foreach \( c_i \in \{c_1, \ldots, c_n\} \)

\[
\text{if} \ \text{empty}(q_i) \\
(b_i \in sv) \rightarrow (b_i = 0)
\]

### 6.8.2.1 Mapping Java methods to bits

During the collection of information undertaken by our pre-processor, the chords appearing in a class are numbered. This number is used in the generation of the munged names that are provided as hooks to the chord-implementation auxiliary code. These numbers are unique per-channel (within the object), well-defined and sequential which are the properties required by any channel indexing function, see section 4.4.2 and appendix D. In addition, this numbering is defined to start from zero.

This index is used to provide the position in the bit vector to which the channel corresponds. For example, channel 0 uses bit zero. Converting this into an integer (which is our chosen representation of bit-vectors) gives us \( 2^0 = 1 \).

The following code is used to implement an asynchronous channel which is numbered 3 in its class. Its bit-vector representation is calculated as \( 2^3 = 8 \);

```java
private void fullUp(final int seats) {
    lock();
    que8.enque(seats);

    if (!matches(8))
        [ 
            set(8); // state has changed, so run the match algorithm
            // more on this later.
            scan();
        ]
    unlock();
}
```

Listing 6.11: Asynchronous Channel

Here a message arrives. The lock is obtained to ensure mutual exclusion (see chapter 7), then the message is put into the pending queue. The queue must now be not-empty (since we have the lock, and we have just put something into it). If the status vector does not match, then the queue must previously have been empty, and we need to update the status vector to represent the new state of the object; this might also mean that a chord which didn’t match before now matches, so we have to run the matching algorithm.

Finally the lock is released.
6.8.2.2 Complexity

The complexity of maintaining the state vector is $O(1)$ for message arrival, a single bit is checked and then possibly set. For chord dispatch, the size of the queue for each of the channels in the chord must be checked and a bit unset if appropriate. Since the “isEmpty()” operation on queue runs in constant time, this gives $O(n)$ operations where $n$ is the number of channels involved in the chord being dispatched.

6.9 Conclusions

We have used our formal description to derive data structures and operations similar to those in [1]. By using this formal approach we hope to convince the reader that the implementation of chords is rigorous, and that the reasoning process applied to the formalism in later chapters can also be applied to the implementation.

We derive the complexity of the operations to do with message arrival and chord dispatch, and show that they are all either $O(n)$ or faster. We also discuss the trade offs between thread enqueueing complexity and thread dequeueing complexity. We have not yet defined an algorithm for implementing the atomicity of dispatch, or shown how that can be implemented in code.

We will describe two such algorithms in chapter 7.

We define and discussed fairness and determinism, deciding that both our algorithms are both unfair and deterministic, although the determinism was masked by the non-determinism of the underlying scheduler.
Chapter 7

Dispatch and synchronisation

Things should be made as simple as possible, but not any simpler. — Albert Einstein

So, now that we have developed a system for maintaining object-state how can it be translated into an algorithm implementable in a standard multi-threaded concurrency system?

7.1 Introduction

We describe an algorithm based on our formal description of chords. This algorithm is called the simple algorithm, and we prove that it is correct in only scanning for matches once each time a message arrives. We describe the implementation of the algorithm in a multi-threaded environment, and calculate the complexity of the operations involved. We discuss how some of the optimisations applied to the algorithm in [1] can be applied to our algorithm.

Then we look at the algorithm in [1] (which we call the polyphonic algorithm), how it relates to our algorithm and what benefits and drawbacks it has. We look at an intuition as to its correctness by comparing it with our algorithm and, by looking at the descriptions in [1], describe why it has to scan for matches twice in some situations.

Finally we see if the complexity of this algorithm is better or worse than our algorithm.

We provided both algorithms in our chorded-Java implementation. The user can select either with a command-line switch to the compiler.

7.2 Simple Algorithm

The most obvious, from our formalism, implementation of a chord matching algorithm is just to check each possible match every time a message arrives. This algorithm combines several of the stages discussed in chapter 6, and can be seen in box 7.1.
For an incoming message $m_j$ on channel $c_k$ for $object_i$;

```
lock(statusvector(object_i))

sv = statusvector(object_i)
(b_k ∈ sv) → (b_k = 1)
[q_1, ..., q_m]^t = object_i
add(m_j, q_k)

foundMatches = ∅
foreach pv ∈ patternvectors(object_i) {
    if (matches(pv, statusvector(object_i)))
        then pv ∈ foundMatches
}
if(foundMatches == ∅)
    unlock(statusvector(object_i))
end

match = selectMatch(foundMatches)
∃b ∈ bodies(class(object_i)) : match = pattern(b)
dispatch(pattern(b))
unlock(statusvector(object_i))
run(b)
```

Algorithm 7.1: Full pattern matching algorithm
7.2.1 Auxiliary Functions

The function \texttt{run(b)} takes the messages consumed in the dispatch step and puts them into the context of the body \texttt{b}. Then makes the code ready to be scheduled.

The function \texttt{selectMatch(foundMatches)} picks one match from the set of matches. In our implementation this will be the first match (which means that the scanning algorithm can safely stop after it finds the first element of \texttt{foundMatches}), see chapter 6.

7.2.2 One run is sufficient

An interesting property can be proved, that running the scan once per message arrival is enough to dispatch all possible matches. The property is expressed as two lemmas, Lemma 7.1 and the supporting Lemma 7.2. The proof of these two lemmas shows that a message arrival can cause at most one match, which our algorithm will dispatch.

7.2.3 No matches can only be followed by single dispatch

Lemma 7.1 (No matches can only be followed by single dispatch) \textit{If no chords matched before a message arrived, then message arrival allows at most one dispatch.}

Which could also be written; there is never a case where two of the \texttt{foundMatches} could be dispatched, without at least one more message arriving.

The proof of this lemma works by contradiction. Assume that a messages arrives on channel \texttt{c_x}, and subsequently more than one chord can be dispatched. More formally, assume, for contradiction, the condition

- The message in the queues of the object before the arrival of the message were not sufficient to dispatch any chords. This is from the pre-condition of the lemma.
- When the new message \texttt{m_y} arrives on channel \texttt{c_x}, more than one chord can be dispatched (without loss of generality, call two of these chords \texttt{b_1}, and \texttt{b_2}).

The proof goes as follows:

- As shown in section 6.5, the set of dispatchable chords depends on the the set of non-empty message queues.
- No chord was dispatchable before the message \texttt{m_y} arrived, but more than one chord was dispatchable afterwards. \texttt{m_y} must have caused a change in the state of the set of pending queues. It can only have done that by changing a previously empty queue into a non-empty queue. This will be \texttt{q_x} (since the message arrived on channel \texttt{c_x}).
- The length of queue \texttt{q_x} after the message arrival will be exactly one. This must be the case, since it was empty and now a single message (\texttt{m_y}) has been added to it.
• If chord \( b_1 \) is dispatched before \( b_2 \) it will consume the single message in the queue \( q_x \). However \( b_2 \) must still be dispatchable. This means it must be dispatchable without the message in \( q_x \). Since this message queue is the only change in state caused by the arrival of the message \( m_x \), the chord \( b_2 \) must have also matched before the arrival of the message. Contradiction\(^1\).

This is intuitive, the chords which start to match on the arrival of a message must have the channel corresponding to the message in their pattern. This means they must all consume a message on that channel when they are dispatched, and since there is only one message on that channel, only one of them can be dispatched.

There is another consequence of this lemma; assuming the precondition of the lemma holds, no chord ever matches at the end of the matching algorithm (at the point the lock is released, after the matching chord has been dispatched). This is just another way of writing the lemma — if a chord did match at the end of the matching algorithm then it would be dispatchable, and we have shown that is never the case.

### 7.2.4 There is never a match before a message arrival

We need to show that the precondition of the previous lemma always holds, in order to prove that our single scan dispatch is correct.

This is an inductive proof, with induction hypothesis:

**Lemma 7.2 (No pending matches between dispatch and arrival)** When a message arrives at an object, before it is added to the queues, no chord matches.

This can be stated as, the queues do not match any chords in the time between message arrivals — they can only match in the time between the start of the dispatch algorithm, where the message is added to the queue and the end of the dispatch algorithm, where the matching chord (if any) is dispatched.

#### 7.2.4.1 Base Case

All objects start with all of their queues empty. Since each chord includes at least one channel, then no chord matches.

Even the presence of constructors will not changes this, since the messages they send will also be subject to the dispatch algorithm.

#### 7.2.4.2 Induction Step

Assuming the lemma is true for state \( n \) (before the arrival of the message), we show that it is true for state \( n+1 \) (after the message has arrived and the dispatch algorithm has been run).

\(^1\text{Since } b_1 \text{ and } b_2 \text{ were arbitrarily and generally picked the case where } b_2, \text{ or any other match, is dispatched first is symmetric with this one.}\)
The precondition for lemma 7.1, holds in state n — no chords match (by induction hypothesis). Lemma 7.1 states that if no chords match before the arrival of a message, then none will match afterwards (this is a paraphrase of the lemma, justified in section 7.2.3). So the lemma 7.2 holds in state n+1.

Hence the hypothesis holds.

7.2.5 Complexity

The worst case complexity is where a message arrives and a match is found. For a study of the complexity of the queue operations see chapter 6. Adding the message to the queue is $O(x)$, where $x$ is the size of the message. Each match check requires one operation (the bitwise comparison), but each pattern must be checked — giving us $m$ operations, where $m$ is the number of chords in the object.

The select matches function is constant time, since we just take the first element of the set. Removing the messages from the queue requires a dequeue operation for each channel that belongs to the pattern that has matched. Each dequeue operation is $O(n)$, so the dispatch operation is $O(n^2)$ in the worst case. However, the operation is limited to the size of the largest chord and the largest number of parameters in a message in any element. And is fixed for any given object.

7.2.6 Implementing the algorithm using threads

We have not yet discussed the implementation of the algorithm using threads. There are two aspects involved in the implementation, firstly that the locking shown guarantees atomicity. The locking is very simple indeed; the lock is obtained before any state of the object is changed by adding a message to the queue, and released after, either no match has been found, or a match has been found and the messages have been consumed from the queue.

The lock is released before the body is run. This prevents the algorithm from adding any deadlocks to the system. No other lock is ever obtained whilst the statusvector lock is being held, so no cycle can occur. If the lock was not released before the body was run then the body may try to obtain another lock, and a cycle could occur.

The second aspect of implementing the algorithm with threads has to do with distinguishing asynchronous and synchronous channels. In particular, a fully-asynchronous chord must be run in a new thread — whereas a chord with a synchronous element must be run in the thread of the caller of the synchronous channel.

Our algorithm can be used as it stands. The auxiliary function run(b) just has to make sure that the body executes in the correct context. Either by passing the messages to the waiting synchronous thread or by creating a new thread and running the body in that.

Our chorded Java compiler implements the auxiliary function run(b), in exactly this way, although it inlines it into the matching code. An example of the match scan for the Simple Algorithm can be see below;

```java
public void scan ()
{
    if (matches(3))
    {
```
que1.deque();
if (que1.empty())
{
clear(1);
}

final Object[] message2 = que2.dequeue();
if (que2.empty())
{
clear(2);
}

threadQuel.release(new MethodPointer1()
{
    public String call()
    {
        return _1(((String) message2[0]));
    }
});

Listing 7.1: Simple Scan

The code dequeues messages from each queue in turn and combines them all into one closure with a reference to the chord body (in this case called "_1"). This chord is then sent to the queue of threads which are pending on the synchronous element of this chord, where it is executed.

A fully asynchronous chord is treated a little differently. Here a new thread is created and started to run the chord body.

ever if (matches(9))
{
    que1.dequeue();
    if (que1.empty())
    {
        clear(1);
    }

    que8.dequeue();
    if (que8.empty())
    {
        clear(8);
    }

    final Thread thread = new Thread(new Runnable()
    {
        public void run()
        {
            _2();
        }
    });
    thread.start();

    88
7.2.7 Memory Barriers

An overview of the synchronisation involved in the simple algorithm can be seen in figure 7.1. Of particular interest is whether or not this algorithm is correct in the Java 1.4.2 Memory Model[11][25] since this is where it is expected to run.

Data changes are made in the critical, central, section of the algorithm. To enter this section a thread must obtain the semaphore protecting it; this ensures structural exclusion. The synchronisation on the lock, which forms the semaphore, will flush memory changes from other threads which have previously modified the critical data.

Figure 7.1: Flow-chart overview of simple algorithm

Once the match has been selected, the message data is dispatched from the queues. The thread releases the semaphore and again synchronises, which makes the changes it has made visible to the other threads. It then performs the necessary operations to run the selected chord.
7.2.8 Optimising the Algorithm

This simple algorithm, as presented so far does present the opportunity for a couple of minor efficiency gains. As discussed in [1], we can take advantage of the limited number of patterns that the receiving channel is involved in.

As we have shown earlier in this chapter, the matches caused by the arrival of a message on a channel will all involve that channel. We can use this fact to avoid scanning chords which can never be effected by the arrival of a message on a particular channel.

By calculating in advance the chords which can be effected by a particular channel, and then only scanning those chords for matches, several iterations of the pattern matching loop can be saved. Our chorded-Java implementation of the simple algorithm does not include this optimisation, although our implementation of the polyphonic algorithm does (since this algorithm already requires us to split up the match finding pass).

In addition, if the message arrival is to a queue which is not already empty, then no additional patterns will match. Since we have already established that the number of matches prior to the arrival of the message is zero, then there is no need to scan for matches under these circumstances. Our implementation of the simple algorithm, and our implementation of the polyphonic algorithm both include this optimisation.

7.3 Polyphonic Re-scanning Algorithm

In [1], the authors propose a two stage algorithm for chord dispatch. This algorithm is somewhat more complex than the algorithm presented in the first half of this chapter. In particular, when a match containing a synchronous element is found the chord is not dispatched straight away — instead the synchronous thread is notified of the possibility of a match. The synchronous thread then wakes up and rescans for the match.

The algorithm trades off simplicity and a single-pass scan, for a multi-pass scan and a more complex implementation; in-order to try and reduce the number of context switches required to dispatch a chord. It is also able to reduce the need to queue and then immediately dequeue messages to a synchronous channel.

7.3.1 Asynchronous Channels

Message which arrive on asynchronous channels are the simplest to describe. The body of any chord which matches as the result of a message on an asynchronous channel will definitely not-execute in the thread that the message arrives on. Either the chord that matches will be fully asynchronous, and the matched chord will run in a new thread, or the matched chord will have a synchronous element — and the matched chord will run in the thread that called the synchronous element.

```java
public void put(final String string) {
    lock();
    que2.enqueue(string);
}
```
if (!matches(2))
{
    set(2);
    scan();
}
unlock();
}  

Listing 7.3: Asynchronous Channel

Here, the message is put into the queue corresponding to the chord. Then, if the chord does not match (that is, before the arrival of the message, the queue was empty), the status vector is updated and a scan for matches initiated. Otherwise the lock is released and the calling thread can continue execution.

7.3.2 Synchronous Channels

The synchronous channel is why this polyphonic algorithm was designed the way it is. In [1], the observation is made that in some situations synchronous channels can select and dispatch a match with no context switch at all. The implementation is optimised for this situation.

When a message arrives on the channel a check is performed to see if the queue already has pending messages; if it does then the thread immediately queues itself. If not, and everytime a waiting thread is woken up, it scans through each of the chords it is involved with. If any of them match, then it dequeues the rest of the pending messages, performs another scan then unlocks the lock and runs the relevant body.

```java
public String get()
{
    lock();

    while (true)
    {
        if (matches(1))
        {
            que1.que();
            if (que1.empty())
            {
                clear(1);
            }
        }

        if (matches(2))
        {
            final Object[] message2 = que2.deque();
            if (que2.empty())
            {
                .clear(2);
            }

            scan();
            unlock();
        }
    }
```
Listing 7.4: Synchronous Channel

return _l(((String) message2[0]));
}
else
{
    set(1);
}
}

Why does it preform a second, full, scan? We already showed that our simple algorithm does not need to. The reason lies in the implementation of ThreadQueue (see listing 6.6). After the thread is released from the queue, but before it regains the status-vector lock, there is a window for messages to arrive. This breaks our invariant.

An example of how this situation can lead to deadlock can be seen in [1].

7.3.3 More Complicated

The polyphonic algorithm is depicted in figure 7.2. The additional complexity of the algorithm contrasts well in comparison to our simple algorithm in figure 7.1.

The double scanning is seen in the complex path on the left hand side of the diagram, as well as the different handling of synchronous and asynchronous calls. The creation of new threads is not handled in an equivalent of the run(b) but is preformed in place in the scan.

7.3.4 Correctness

The correctness of the algorithm is covered in [1], but an intuition can be gained by comparing with our simpler algorithm. The chief difference is that in some cases, when a match is found there is a context switch to wake up the thread that must run the body.

This context switch releases the lock; which allows additional messages to arrive. So when the awoken thread eventually gets to run, it has to scan for matches again. It is also no-longer guaranteed that only one chord will be dispatchable, so a re-scan must be done after the dispatch (which in turn may trigger further re-scans).

This can be though of as running our algorithm in a “do, while matches found” loop.

7.3.4.1 Complexity

The complexity of message arrival and chord dispatch is the same as for our simple algorithm, except that for chords with a synchronous component one less message queue is used for the polyphonic algorithm (because the synchronous message is kept on the stack).

The complexity of finding a match is worse though; in the worst case, two complete scans of for each message arrival are required. Even so, the complexity is still linear.

92
Figure 7.2: Flow-chart overview of polyphonic algorithm
7.4 Comparison

Each synchronous message that arrives at our simple algorithm causes all of the other threads waiting on that channel to be woken up. This is not the case for the polyphonic algorithm from [1]. In the best case it can dispatch a chord for which the synchronous message arrives without waking up any other threads. Although in the normal case it is the same as the simple algorithm (with the addition of the extra scan).

Our simple algorithm is much easier to understand than the polyphonic algorithm, and has simpler properties (especially atomicity properties).

7.5 Conclusion

We developed and implemented two algorithms for chorded Java. The simple algorithm has provable properties and follows easily from our formalism, but the polyphonic algorithm from [1] is probably faster in some situations — it involves less context switches when a synchronous message arrives and causes a chord to become ready for dispatch.

Our simple algorithm only has to scan for matches once on each message arrival, but the polyphonic algorithm sometimes has to scan twice.

There is scope for further optimisation of both implementations by in-lining the scanning code.

Both implementations are available for the chorded Java compiler and can be chosen between from a command-line-switch. Perhaps future work could involve benchmarking the two algorithms in different situations.
Chapter 8

Distribution and chords

8.1 Introduction

We discuss how our chorded Java can be used in a distributed system. We use RMI to send the messages between remote locations, and try and decide under what circumstances the synchronisation is still correct.

To do this we compare the notion of value in the Join-Calculus with the notion of value in chorded-Java, and look at which RMI operations preserve the integrity of which types of values.

Finally we develop an extension to chorded-Java which supports RMI more transparently and show an implementation of the Dining-Philosophers problem which works in a distributed system; then compare this implementation to a thread based one.

8.2 Using RMI to distribute chorded Java

RMI provides a mechanism for distributing Java programs across a network, it supports the use of serialisation, and reference-holding proxy objects, as values in its messages.

When object references are passed using a proxy, only one version of the original object ever exists. Messages to it from the remote location are automatically marshaled for transmission across the network, figure 8.1 shows this.

When object references are passed using serialisation, multiple versions of the original object are created. These copies are not kept in sync with each other, once they have been copied they can be used and modified totally independently; figure 8.2 shows this.
8.3 The notion of value

In the Join-Calculus messages consist of values, these values are primitive and immutable. In Java, messages can consist only of primitives and references pointing to objects. In a shared memory system, the references can safely be considered values for the purposes of passing to channels in chorded objects, since even if the object being referred to does change, every holder of the reference will see the change (provided it is properly synchronised).

8.3.1 Channels as values

Java does not have a direct mechanism for passing channels names as values, but it is able to pass collections of instances of channels as values. Object-references are essentially these collections of channels — each object reference points to the set of channels defined by its type.

<table>
<thead>
<tr>
<th>Language</th>
<th>Primitive values</th>
<th>Immutable Objects</th>
<th>Channel Names</th>
</tr>
</thead>
<tbody>
<tr>
<td>Join-Calculus</td>
<td>Primitives</td>
<td>Not supported</td>
<td>Pass as name</td>
</tr>
<tr>
<td>Chorded-Java</td>
<td>Primitives</td>
<td>Object-Reference</td>
<td>Pass as object reference</td>
</tr>
</tbody>
</table>

8.3.2 RMI values

As discussed in section 8.2, RMI has two ways of passing objects; it can serialise them and pass a copy of the object to the remote process, or it can pass a proxy containing a reference to the object.

In chorded-Java it is important that only immutable values are passed by copying — if mutable objects were passed by copying they could become out of sync between the two locations.

<table>
<thead>
<tr>
<th>System</th>
<th>Primitive values</th>
<th>Immutable Objects</th>
<th>Mutable Objects</th>
</tr>
</thead>
<tbody>
<tr>
<td>Shared Memory</td>
<td>Pass by value</td>
<td>Pass by reference</td>
<td>Pass by reference</td>
</tr>
<tr>
<td>RMI</td>
<td>Pass by value</td>
<td>Pass by value</td>
<td>Pass by reference</td>
</tr>
</tbody>
</table>

8.4 Supporting RMI

For a class to be usable over RMI it must either support the serialisable interface (which in chorded-Java it should only do if it is immutable), or extend “java.rmi.server.UnicastRemoteObject”.

96
If the class extends “java.rmi.server.UnicastRemoteObject” then it should also implement at least one “java.rmi.Remote” interface.

8.4.1 The process keyword

We decided to make extending “java.rmi.server.UnicastRemoteObject” more transparent in chorded Java. We introduced a new keyword “process”, which can be used in place of the keyword “class”. When it is, the chorded-Java is automatically translated to have the correct super-class.

The reason for introducing this keyword is to support future work on finding mechanisms for distributing chorded programs which are more appropriate than RMI, and to possibly support multiple different translations of the process keyword so the user can choose between the available mechanisms.

Because the constructors of remote objects throw “java.rmi.RemoteException”, the translator must also add this exception to the constructors of the “process” where it is missing, or add a default constructor if the process has no constructor at all. For example;

```java
public process SimplePhilosopher implements Philosopher
{
    public SimplePhilosopher(final Position position,
                              final PhilosopherObserver observer)
    {
        [...] \ stuff
    }

    [...] \ other stuff
}
```

Listing 8.1: Example of process

Is translated to;

```java
public class SimplePhilosopher extends java.rmi.server.UnicastRemoteObject
    implements Philosopher
{
    public SimplePhilosopher(final Position position,
                              final PhilosopherObserver observer)
        throws java.rmi.RemoteException
    {
        [...] \ translated stuff
    }

    [...] \ translated other stuff
}
```

Listing 8.2: Translation of process
8.5 Distributed Dinning Philosophers

We decided to create an example to illustrate the translated chorded-Java code running distributed over a network; holding with tradition we implemented a solution to the Dinning Philosophers problem.

The solution can be seen running in figure 8.3.

![Figure 8.3: Distributed dining philosophers](image)

The solution uses the not-enough-chairs principal, where the $n$ philosophers are only allowed to sit at the table $n - 1$ at a time. The architecture of the solution can be seen in figure 8.4.

The viewer application is separated from the philosophers by an asynchronous event notification system; this is needed to stop the philosophers being artificially serialised by the GUI thread.

8.5.1 Comparison to non-chorded version

In order to see if the chorded representation of the RMI application is any clearer, we will compare a few components of it with non-chorded versions of those components.

It is possible that an object-orientated version of the solution would use an entirely different architecture; equally however, chorded-programming is still very much in its infancy and it is entirely possible (and if chorded programming becomes popular, probable) that better idioms and patterns for using chords will emerge.

8.5.1.1 The seating

The seating keeps track of how many seat there are and how many are available, it presents two public channels, “sitDown” and “standUp”. The “sitDown” channel blocks when the seating is full.
Figure 8.4: Philosophic UML
The chorded object has two states, either it is “fullUp”, or it has seats “available”. It is easy for a developer to see that when it is “fullUp” only the “standUp” channel will match, since that is the only channel involved in a chord with “fullUp”.

The state transitions are explicitly encoded, for example when “sitDown” is called it matches the “available” message; then if there is more than one available seat the seating remains “available” else it becomes “fullUp”.

For comparison, a typical Java implementation of the same class can be seen below;

```
public class SimpleSeating extends UnicastRemoteObject implements Seating {
    private final int m_seats;
    private int m_empty;

    public ThreadedSimpleSeating(final int seats) throws RemoteException {
        m_seats = seats;
    }
```
The first observation is that the class explicitly extends “UnicastRemoteObject” and throws “RemoteException”, this is a minor readability problem. The state is held in member variables, so the developer has to explicitly synchronize the methods that manipulate it. The developer is also required to explicitly code the “wait” and “notify” conditions, and deal with the potential interrupted exception.

However, because this class is so small, and the wait-notify condition so simple, a Java developer would probably have no trouble reading it.

### 8.5.2 The philosopher

A much more interesting class, from a distribution point of view, is the philosopher (which has been simplified for clarity, the event reporting code and logic for getting and replacing the sticks has been removed — the actual class is about three times this size).

The code which is left implements the main loop of the philosophers behaviour, and encodes the ability to stop the philosopher and have her resume from the same state — without loss of synchronisation across the system.

```java
public process SimplePhilosopher implements Philosopher
{
    private interface Task
    {
        Task preformTask();
    }

    [...] // implementations of Task.
```

Listing 8.4: Threaded Seating
private Task m_thinking = new ThinkingTask();
private Task m_hungry = new HungryTask();
private Task m_eating = new EatingTask();

private final Position m_position;

class SimplePhilosopher(final Position position,
               final PhilosopherObserver observer)
{
    m_position = position;
    m_observer = observer;
    ready(m_thinking);
}

public async start() & private async ready(final Task task)
{
    Task currentTask = task;
    running();

    while(isRunning())
    {
        currentTask = currentTask.preformTask();
    }
    ready(currentTask);
}

public void stop() & private async running()
{
    stopping();
}

private boolean isRunning() & private async running()
{
    running();
    return true;
}

private boolean isRunning() & private async stopping()
{
    return false;
}

Listing 8.5: Chorded Philosopher

The use of a fully asynchronous chord to start, and restart the thread which runs the philosopher’s main loop simplifies the design considerably. It also more accurately represents the developer’s intention for the asynchronous nature of the client-sever relationship. The client tells the philosopher to start; it is not interested in when, after this time, that the philosopher actually starts (so long as it does start). The use of async makes this explicit.
There is also the possibility of optimising the RMI protocol for use with async messages (since the client knows that it does not need a reply). Future work in this area may prove interesting.

For comparison, here is a similarly simplified threaded version of the system:

```java
public class ThreadedPhilosopher {
    private interface Task {
        Task preformTask();
    }
    private Task m_thinking = new ThinkingTask();
    private Task m_hungry = new HungryTask();
    private Task m_eating = new EatingTask();
    private Task m_currentTask = m_thinking;
    private boolean m_running = true;

    public synchronized void stop() {
        m_running = false;
    }

    public synchronized void start() {
        if (!m_running) {
            Thread thread = new Thread(new Runnable() {
                public void run() {
                    boolean stop = false;
                    while (!stop) {
                        synchronized (ThreadedPhilosopher.this) {
                            if (m_running) {
                                m_currentTask = m_currentTask.preformTask();
                            } else {
                                stop = true;
                            }
                        }
                    }
                }
            });
            m_running = true;
            thread.start();
        }
    }
}
```
This implementation is not the same as listing 8.5; when “start()” returns the caller knows that the thread has definitely been started. This has an important consequence;

- The caller of start must wait while the thread and runnable are constructed and started; however, in comparison to the network delay, this probably makes little difference.
  
  A larger example may have to implement additional code to compensate for this potential performance advantage of chorded code.

Because of the non-determinism in the scheduling of threads however, the caller does not know if the philosopher’s thread has actually started running (been scheduled) in either case.

### 8.6 Conclusions

We discussed the difference between values in shared-memory systems and distributed systems, how RMI copes with this and what impact it has on chorded programs. We decide that it is safe to copy immutable objects in chorded systems, but mutable objects must use the stub-skel remote object features of RMI.

We developed support for RMI in our chorded compiler, adding a convenience keyword “process” — both to simplify the syntax, but also to allow future versions of chorded-Java to support different implementations of distribution using the same keyword.

We implemented a dining-philosophers-problem solution using our RMI support and saw that it was a convenient and clear way of implementing the asynchronous nature of some RMI messages.

There are several possible optimisations that could be made to the RMI protocol in the light of chorded programming. It is possible that async could be supported on the client side, perhaps allowing the client to carry on at an early point in the network communication.

There are also possible optimisations on the server-side. When an RMI message arrives a new thread is started to handle it, perhaps this thread could be reused to run the asynchronous chord instead of starting a second one. Currently the first the RMI handling thread just goes back and returns void to the client before terminating.

Future work could be conducted in both these areas.
Chapter 9

Reasoning about chords

All great men are gifted with intuition. They know without reasoning or analysis, what they need to know. — Alexis Carrel (1857 - 1929)

9.1 Introduction

Now that we have defined what chords are, how chords can be implemented and how programmers program using chords, we can look at how chords can be reasoned about. We will work through a series of examples, in the hope that the results can be used to inform the construction of a framework for automatic analysis.

The three examples in this chapter were picked to illustrate three features of the analysis of chorded programs. The first shows a method for reasoning about properties and invariants of a chorded program, the second shows a method for finding un-reachable code in a chorded program and the third shows that the transition graph of a chorded program can become infinite and describes a method for fixing a finite bound on the graph.

These examples are illustrative rather than exhaustive.

9.2 Conjectures

We conjecture that it is possible to design an algorithm to decide the following things about chorded programs. We hope the examples in this section will go someway to convincing the reader as to why we believe this to be the case, and to perhaps provide a starting point to developing a compiler which can verify these properties automatically.

1. That it is possible to decided that messages to a particular channel in a chord are never consumed.

2. That is is possible to verify at compile time some invariants about the possible states of the pending message queues of an object. And in particular that it is possible to determine that certain channels will never have pending messages at the same time.
3. That it is possible to detect some cases where a particular chord can never be dispatched, and therefore the code in its body is unreachable.

4. That it is possible to decide a maximum bound on the number of messages in some channels, or at least to say whether the queue will stay of finite length.

9.3 Transition Graphs

To construct our analysis we are going to use directed graphs to visualise, and analyse, the state transitions that chorded programs undergo.

9.3.1 Constructing transition graphs

Transition graphs for chorded programs, or at least for classes in chorded programs can be constructed because of the well defined way in which state can change in a chorded class (see section 4.11).

The states in the transition graph of class\( (t) \) will be a subset of the \( \mathcal{P}(\text{channels}(\text{class}(t))) \). That is, in general, an object may have messages pending on any subset of its channels at any particular time.

Having said that, because of the nature of private channels, for any given object, some of the possible transitions cannot be preformed.

9.3.1.1 Private channels

A private channel can only be sent to by code in the class containing the private channel. Since sends can only occur in code which is in chord bodies we can construct the set of possible transitions that an object can undergo.

1. Public channels can always be sent to at any time.

2. Private channels can only be sent to when a pattern is matched and a chord is dispatched.

   When the body of the chord executes it will send to 0 or more channels. The dispatch of chord \( \text{pattern}(\text{body}) \), will consume messages from the channels in pattern and send message to the channels described by \( \text{calledChannels} \)(body).

   In particular, for a chord in type \( t \), we are interested in the set of channels,

   \[ \{ c | (c \in \text{calledChannels}) \land c \in \text{privateChannels}(\text{channels}(\text{class}(t))) \} \]

   We call this mapping of chord to called channels a conversion and use the symbol \( \mapsto \) to distinguish them.

   We disregard the public channels in calledChannels, since we have established already that they can always be sent to at any time.
9.3.1.2 The problem of non-determinism

At first glance it may appear that the dispatch of a chord consumes a set of messages from the queues and then adds a set of messages to the queues. However, this would ignore the non-determinism of scheduling in a concurrent system.

When a chord is dispatched, its body is scheduled for execution. However in a concurrent system there may be some delay before the execution begins.

**Axiom 9.1 (Non-Deterministic execution delay)** After a chord is dispatched there is a non-deterministic delay before it sends any messages.

During the execution of the chord body, the scheduler may pause the execution and switch to a different thread of control at any time.

**Axiom 9.2 (Non-Deterministic inter-message time)** There is a non-deterministic delay between each messages that an executing chord body sends.

And lastly, because we defined calledChannels(body), to give us the set of possible sent messages, and not the order they are sent in we have to consider that;

**Axiom 9.3 (Non-Deterministic message order)** The messages sent by an executing chord body may arrive in any order.

9.3.1.3 Initial state

Objects have a set of initial states that they can be in. These initial states correspond to the constructors of a class. Each constructor will set the object up with a set of 0 or more messages on 0 or more private channels. We have not so far addressed the issue of constructors.

\[
\text{initialState} : \text{TypeDefinition} \rightarrow \mathcal{P}\{\text{Channel,}\}
\]

Should be defined for the language to be analysed. It returns the set of initial states that the object can be in.

9.4 A simple lock

We will consider a simple lock.

```java
class E {
    public E() {
        unlocked();
    }
}
```
```java
public void lock() & private async unlocked()
{
    locked();
}

public void unlock() & private async locked()
{
    unlocked();
}
```

Listing 9.1: Example Lock

What properties would a programmer like to show about this lock?

<table>
<thead>
<tr>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Lock Liveness</strong></td>
<td>If an infinite number of lock and unlock actions are performed, the lock will transition from the locked() to the unlocked() state an infinite number of times.</td>
</tr>
<tr>
<td><strong>Lock Safety</strong></td>
<td>The lock will never be both locked() and unlocked() at the same time.</td>
</tr>
</tbody>
</table>

### 9.4.1 Formal Description of E

E can be described by the class:

```
class(E) =
[((E, lock, public), (E, unlocked, private)), “locked();”],
((E, unlock, public), (E, locked, private), “unlocked();”])
```

And by the object:

```
object_E = [q_1, q_2, q_3, q_4]^E
```

or equivalently by the syntax:

```
object_E = [[...], [[...], [[[...], [[[...]]]]]]^E
```

### 9.4.1.1 Initial States

What are the initial states that the lock can be in? We know that when an object is created, exactly one of its constructors is called and returns. E has only one constructor which sends exactly one message on the private channel unlocked().

The initial state of the object is therefore:

```
object_{E-initial} = [0_1, [0_2, [0_3, [1_4]]]^E
```

108
9.4.1.2 Public channels

Two of the channels are public, so they always have an infinite number of messages available.

\[ \text{object}_{E0} = [[\infty], [0], [\infty], [1]]^E \]

9.4.1.3 Possible conversions

There are two chords in the class E. So there are two possible message conversions,

\[ \text{Con}_1 = \{(E, \text{lock, public}), (E, \text{unlocked, private})\} \leadsto \{1 : (E, \text{locked, private})\} \]

\[ \text{Con}_2 = \{(E, \text{unlock, public}), (E, \text{locked, private})\} \leadsto \{1 : (E, \text{unlocked, private})\} \]

9.4.2 Working out the transition graph of E

Now the transition graph of the object can be worked out. We start with the initial state of the object,

\[ \text{state}_0 = (\text{object}_{E0}, \text{transit}_0) \]

\[ \text{object}_{E0} = [[\infty], [0], [\infty], [1]]^E \]

\[ \text{transit}_0 = {} \]

Applying the pattern-matching algorithm shows a single match with Con\(_1\). Changing the state of the object to:

\[ \text{state}_1 = (\text{object}_{E1}, \text{transit}_1) \]

\[ \text{object}_{E1} = [[\infty], [0], [\infty], [0]]^E \]

\[ \text{transit}_1 = \{1 : (E, \text{locked, private})\} \]

There is only one message in transit, and no matches, so we have no choice, the next transition is the arrival of the message.

\[ \text{state}_2 = (\text{object}_{E2}, \text{transit}_2) \]

\[ \text{object}_{E2} = [[\infty], [1], [\infty], [0]]^E \]

\[ \text{transit}_2 = {} \]

Now there are no messages in transit, but there is a single match, with Con\(_2\); producing the state:

\[ \text{state}_3 = (\text{object}_{E3}, \text{transit}_3) \]

\[ \text{object}_{E3} = [[\infty], [0], [\infty], [0]]^E \]

\[ \text{transit}_3 = \{1 : (E, \text{unlocked, private})\} \]

Again, there are no matches and only one message in transit, giving our final transition to:

\[ \text{state}_0 = (\text{object}_{E0}, \text{transit}_0) \]
The graph of this system can be seen in figure 9.1.

Figure 9.1: Transition diagram of a lock

9.4.3 Showing the properties

So, now that we have found all of the states of the system, and the valid transitions between them, can we show that the properties (safety and liveness) are satisfied?

9.4.3.1 Lock Liveness

The graph (figure 9.1) is strongly connected (see appendix B). From our definition of liveness for the chord system (Axiom C.2), we know that the lock will execute an infinite number of steps. There is no choice in the transitions that the lock can take, so every state will be visited an infinite number of times. Since in one of the states (state0) there is a message on the unlocked() channel and none on the locked() channel, and in another (state2) there is no message on the unlocked() channel but there is a message on the locked() channel then our Lock Liveness condition is satisfied,

9.4.3.2 Lock Safety

The graph (figure 9.1) contains all of the possible states of objects of type E. None of those states has messages on both locked() and unlocked(), so the situation can never occur.

The lock is safe.

9.4.4 Accuracy

With what accuracy have we determined the safety and liveness conditions? Are the answers above definite answers or can the properties still be violated? The accuracy of the answers for each of these properties is very different.
9.4.4.1 Safety

The safety answer is completely accurate. There is no way any transition can occur that is not in our graph, and so no state can be reached where the property is violated.

9.4.5 Liveness

The liveness answer is inaccurate. Our analysis finds the maximum number of transitions (and hence states) that can occur. If, for instance, one of the chords was changed to

```java
public void lock() & private async unlocked()
{
    if(random.nextInt(100) < 75)
    {
        locked();
    }
}
```

Listing 9.2: Broken Lock

then after some time the lock may stop changing state. Our analysis cannot detected this situation. In order to detect this situation, calledChannels would have to return the set of sets of possible called channels. And each set evaluated for each state. This extension and its implications could be looked at in future work.

9.5 Detecting unreachable code

We will consider our simple lock with an innocent looking addition.

```java
class F
{
    public F()
    {
        unlocked();
    }

    public void lock() & private async unlocked()
    {
        locked();
    }

    public void unlock() & private async locked()
    {
        unlocked();
    }

    public void block() & private async locked()
    & private async unlocked()
    {
        unlocked();
    }

    ... (remaining code)
}
```

111
9.5.1 Formal Description of $F$

$F$ can be described by the class:

\[
\text{class}(F) = 
\left\{
\begin{array}{ll}
((F, \text{lock, public}), (F, \text{unlocked, private})), & \text{“locked();”}, \\
((F, \text{unlock, public}), (F, \text{locked, private})), & \text{“unlocked();”}, \\
((F, \text{unlock, private}), (F, \text{locked, private})), & \text{“unlocked()”}, \\
((F, \text{block, public})_1, (F, \text{lock, public})_2, (F, \text{locked, private})_3, \\
(F, \text{unlock, public})_4(F, \text{unlocked, private})_5) \\
\end{array}\right\}^F
\]

And by the object

\[
\text{object}_F = [q_1, q_2, q_3, q_4, q_5]^F
\]

or equivalently by the syntax,

\[
\text{object}_F = [[...], [...], [...], [...], [...]]^F
\]

9.5.2 Initial States

The initial states of the object are defined by the constructor. The initial states of this object are similar to object $E$.

\[
\text{object}_{F\text{-initial}} = [[0], [0], [0], [0], [1]]^F
\]

9.5.3 Public Channels

There are 3 public channels in this class, again we set all of the channels to $\infty$.

\[
\text{object}_{F0} = [[\infty], [\infty], [0], [0], [0]]^F
\]

9.5.3.1 Possible conversions

There are three chords in the class $F$. So there are three possible message conversions,

\[
\begin{align*}
\text{Con}_1 &= \{(F, \text{lock, public})_1, (F, \text{unlocked, private})_4\} \rightarrow \{1 : (F, \text{locked, private})_2\} \\
\text{Con}_2 &= \{(F, \text{unlock, public})_3, (F, \text{locked, private})_2\} \rightarrow \{1 : (F, \text{unlocked, private})_4\} \\
\text{Con}_3 &= \{(F, \text{block, public})_1, (F, \text{locked, private})_3, (F, \text{unlocked, private})_5\} \\
& \rightarrow \{1 : (F, \text{unlocked, private})_5\}
\end{align*}
\]
9.5.4 Working out the transition graph of F

Now the transition graph of the object can be worked out. We start with the initial state of the object,

\[ \text{state}_0 = (\text{object}_{F0}, \text{transit}_0) \]
\[ \text{object}_{F0} = (\{\{\infty\}_1, \{\infty\}_2, \{0\}_3, \{\infty\}_4[1]_5\})^{F} \]
\[ \text{transit}_0 = {} \]

Applying the pattern-matching algorithm shows a single match with Con$_1$. Changing the state of the object to:

\[ \text{state}_1 = (\text{object}_{F1}, \text{transit}_1) \]
\[ \text{object}_{F1} = (\{\infty\}_1, \{\infty\}_2, \{0\}_3, \{\infty\}_4[0]_5)_F \]
\[ \text{transit}_1 = \{1 : (F, \text{locked}, \text{private})_3\} \]

There is only one message in transit, and no matches, so we have no choice, the next transition is the arrival of the message.

\[ \text{state}_2 = (\text{object}_{F2}, \text{transit}_2) \]
\[ \text{object}_{F2} = (\{\infty\}_1, \{\infty\}_2, \{0\}_3, \{\infty\}_4[0]_5)_F \]
\[ \text{transit}_2 = {} \]

Now there are no messages in transit, but there is a single match, with Con$_2$; producing the state:

\[ \text{state}_3 = (\text{object}_{F3}, \text{transit}_3) \]
\[ \text{object}_{F3} = (\{\infty\}_1, \{\infty\}_2, \{0\}_3, \{\infty\}_4[0]_5)_F \]
\[ \text{transit}_3 = \{1 : (F, \text{unlocked}, \text{private})_5\} \]

Again, there are no matches and only one message in transit, giving our final transition to;

\[ \text{state}_0 = (\text{object}_{F0}, \text{transit}_0) \]
\[ \text{object}_{F0} = (\{\infty\}_1, \{0\}_2, \{\infty\}_3, \{1\}_4)_F \]
\[ \text{transit}_0 = {} \]

The graph of this system can be seen in figure 9.2, which is identical to the original lock system.

9.5.5 Finding the problem

The conversions used in building the transition graph are the obvious place to start. We can see that only 2 of them are used. Since conversions represent chords this observation corresponds directly to only 2 of the chords matching at any point.

The conversion which doesn’t match is Con$_3$, the conversion representing the chord

\[ (\{(F, \text{unlock, private}), (F, \text{locked, private}), (F, \text{block, public})\}, "\text{unlocked}()") \]
This chord needs to consume a message on private channels (F, unlock, private) and (F, locked, private). But this situation never occurs in the system. The chord is unreachable, it can never run.

### 9.5.6 Accuracy and safety

The analysis is safe. It will not mistakenly identify reachable code as unreachable. The transition graph generated is maximal, all possible states are found, so if a chord doesn’t match any of those states it will never match.

The analysis in not accurate (complete). It may miss some occurrences of unreachable code. This inaccuracy largely stems from the inaccuracy of the

\[
calledChannels : \text{Code} \rightarrow \mathcal{P}\{\text{Channel}\}
\]

function. It is possible to construct a method which looks under analysis like it will call a channel but in practise never does. This is language dependent, but in Java could be caused by an object (A) calling a method in another object (B) that always returns false. However, because Java supports separate compilation this fact cannot be used for optimisations in object A just incase a latter version of object B starts returning true.

It may also be impractical to fully analyse the code in an object, it is often a trade off between compilation complexity (and hence time) and the accuracy of the analysis[13].

### 9.6 Properties of infinite graphs

Some graphs are infinitely large, is it still possible to reason about them?

```java
class G {
    public G() {
        open();
    }

    public void down() & private async open()
}
```
Listing 9.4: Infinite Graph

9.6.1 Formal Description of G

G can be described by the class;

\[
\text{class}(G) = \left\{ \left[ \left( G, \text{down}, \text{public} \right), \left( G, \text{up}, \text{public} \right), \text{""} \right], \left( G, \text{open}, \text{private} \right), \left( G, \text{down}, \text{public} \right), \left( G, \text{open}, \text{private} \right), \left( G, \text{up}, \text{public} \right) \right\}^G
\]

And by the object

\[
\text{object}_G = [q_1, q_2, q_3]^G
\]

or equivalently by the syntax,

\[
\text{object}_G = [[\ldots]_1, [\ldots]_2, [\ldots]_3]^G
\]

9.6.2 Initial States

The initial states of the object are defined by the constructor. The initial states of this object are similar to object E.

\[
\text{object}_{G,\text{initial}} = [[0]_1, [1]_2, [0]_3]^G
\]

9.6.3 Public Channels

There are 2 public channels in this class, again we set all of the channels to $\infty$.

\[
\text{object}_{G0} = [[\infty]_1, [1]_2, [\infty]_3]^G
\]

9.6.3.1 Possible conversions

There are two chords in the class G. So there are two possible message conversions,

\[
\text{Con}_1 = \{(G, \text{down}, \text{public})_1, (G, \text{open}, \text{private})_2 \} \rightarrow \{\}
\]
Con₂ = {(G, up, public)₁} ↝ {1 : (G, open, private)₂}

The problem is now starting to become obvious. Intuitively Con₂ transforms a single (G, up, public)₁ to a single (G, open, private)₂. However we have already established that there are an infinite number of messages on each public channel. Let’s see how this progresses.

9.6.4 Working out the transition graph of G

Starting with the initial state and no messages in transit;

\[
\text{state}_0 = (\{ \text{object}_G \}, \text{transit}_0)
\]

\[
\text{object}_G = \left[ [\infty]_1, [1]_2, [\infty]_3 \right]^G
\]

\[
\text{transit}_0 = \{
\}
\]

Either Con₁ or Con₂ match. Giving us two possible states; first Con₁:

\[
\text{state}_1 = (\{ \text{object}_G \}, \text{transit}_1)
\]

\[
\text{object}_G = \left[ [\infty]_1, [0]_2, [\infty]_3 \right]^G
\]

\[
\text{transit}_1 = \{
\}
\]

And now Con₂:

\[
\text{state}_2 = (\{ \text{object}_G \}, \text{transit}_2)
\]

\[
\text{object}_G = \left[ [\infty]_1, [1]_2, [\infty]_3 \right]^G
\]

\[
\text{transit}_2 = \{ 1 : (G, open, private) \}
\]

Taking each of these states in turn, state₁ has only one possible transition — Con₂ matches, giving:

\[
\text{state}_3 = (\{ \text{object}_G \}, \text{transit}_3)
\]

\[
\text{object}_G = \left[ [\infty]_1, [0]_2, [\infty]_3 \right]^G
\]

\[
\text{transit}_3 = \{ 1 : (G, open, private) \}
\]

State 2 has several possible matches, Con₁ and Con₂, and the pending message could arrive. Giving us;

\[
\text{state}_3
\]

\[
\text{state}_4 = (\{ \text{object}_G \}, \text{transit}_4)
\]

\[
\text{object}_G = \left[ [\infty]_1, [1]_2, [\infty]_3 \right]^G
\]

\[
\text{transit}_4 = \{ 2 : (G, open, private) \}
\]

\[
\text{state}_5 = (\{ \text{object}_G \}, \text{transit}_5)
\]

\[
\text{object}_G = \left[ [\infty]_1, [2]_2, [\infty]_3 \right]^G
\]
transit_5 = {}

State 3 matches Con_2, and also the arrival of the message, giving transitions to

\[
\text{state}_6 = (\{\text{object}_{G6}\}, \text{transit}_6)
\]

\[
\text{object}_{G6} = [[[\infty]]_1, [[0]]_2, [[\infty]]_3]^G
\]

\[
\text{transit}_6 = \{2 : (G, \text{open, private})_2\}
\]

and

\[
\text{state}_0
\]

State 4 matches Con_1, Con_2 and can have a message arrive;

\[
\text{state}_6
\]

\[
\text{state}_7 = (\{\text{object}_{G7}\}, \text{transit}_7)
\]

\[
\text{object}_{G7} = [[[\infty]]_1, [[1]]_2, [[\infty]]_3]^G
\]

\[
\text{transit}_7 = \{3 : (G, \text{open, private})_2\}
\]

\[
\text{state}_2
\]

State 5 matches Con_1 and Con_2

\[
\text{state}_2
\]

\[
\text{state}_8 = (\{\text{object}_{G8}\}, \text{transit}_8)
\]

\[
\text{object}_{G8} = [[[\infty]]_1, [[2]]_2, [[\infty]]_3]^G
\]

\[
\text{transit}_8 = \{1 : (G, \text{open, private})_2\}
\]

The diagram of the states so far can be seen in figure 9.3. It should be clear by now that the number of states in this system is very large (infinite in fact), asking a computer to analyse them all would be a fruitless task. However, some of the states are very similar (state_0 and state_5 both match only Con_1 and Con_2).

### 9.7 Reducing the transition graph of G

This observation that transition graphs can become infinite seems to belie our assertion that we can design an analysis that will definitely terminate. How can we find the maximal transition graph of an object when the transition graph can be infinitely large? The solution to this dilemma stems from two observations;

1. There are a finite number of channels in an object (Lemma 4.3).
2. The set of pending messages is a finite size (see section 4.7.1).

9.7.1 Macro State

Since the space of states of objects in our system is potentially infinite, we need to describe a finite notion of state which still allows us to reason correctly about the behaviour of objects, but with a terminating algorithm in finite resources.

To do this we define the notion of the macro state of an object. The macro-state of an object is defined by the transitions that it can undergo;

**Axiom 9.4 (Macro State)** *Two states are in the same macro state, if they match the same possible transitions: they match the same message conversions, and can receive the same pending messages.*

Now that we have this definition of Macro-States we can show that the number of Macro-States in any given system is strictly finite (infact, it turns out we can calculate the exact upper bound).

9.7.1.1 Queue states

Which message conversions match in any given state depends entirely on the state of the queues in the object. More specifically it depends on whether any given queue is empty or not (see section 6.5).

The object’s queue state can be reduced to the object’s queue-macro-state by the following function. Given an object

\[ object = [q_1, \ldots, q_m : mQueue] \]
we can find the macro-state-vector

\[ \text{macroState}(\text{object}) = \{ s_0, \ldots, s_m : \text{boolean} \} \]

We create an indexed set (see 4.10.1) of booleans

\[ \text{macroState}(\text{object}) = \{ s_i | q_i \in \text{object} \land (s_i = \neg \text{empty}(q_i)) \} \]

That is, we describe the state by the channels that have pending messages, which in turn determines which chords can match in that state.

Since the number of queues in an object is finite (it is exactly \( m \)), we can determine the maximum possible number of macro-states the object can be in. It is the same as the maximum possible number of states that the queues can be in when the state of a queue is either “empty” or “not empty”.

The number of possible configurations of \( m \) booleans is; \( 2^m \). That is, if an object has \( m \) channels, it can be in \( 2^m \) macro-states based only on channels. However, we have already shown that the public channels always have the value \( \infty \); they do not need to be considered in the number of possible states. So for an object \( \text{object} = [q_1, \ldots, q_m : \text{mQueue}] \), the number of possible macro states are

\[ |\text{queueMacroStates}(\text{object})| = 2^{\text{public(channels(type(\text{object}))}}} \]

9.7.1.2 Overall macro-states

There are also the pending messages to consider; again only messages pending on private channels are important. There are exactly the same number of possible states of the pending messages as there are of the message queues (since messages can only be pending on an queue which exists!). This gives the final upper bound on the number of macro-states as

\[ |\text{macroStates}(\text{object})| = 2^{2^{\text{public(channels(type(\text{object}))}}}} \]

Of course, not every macro-state will be reachable in every system; so the total reachable macro-states may be considerably less than this figure.

9.7.2 Recalculating the transition graph

We have only one private channel in the transition graph of \( G \), the channel “open”, so we have only \( 2^{2 \times 1} = 4 \) possibly macrostates. The possible macrostates are;

\[
\begin{align*}
\text{mState} &= (\text{queues}, \text{pending}) \\
\text{mState}_0 &= (\{\}, \{\}) \\
\text{mState}_1 &= (\{\}, \{(G, \text{open, private})_2\}) \\
\text{mState}_2 &= ((G, \text{open, private})_2, \{\})
\end{align*}
\]
\[ mState_3 = ((G, \text{open}, \text{private}_2), (G, \text{open}, \text{private}_2)) \]

\( mState_0 \) represents the starting configuration of our object; a similar process to that used for state transitions can be applied to our construct our macro-state transition graph.

\( mState_0 \) matches conversion \( \text{Con}_2 \), to give us \( mState_1 \).

\( mState_1 \) matches both conversion \( \text{Con}_2 \), to transition to itself, and can also have the pending message arrive to transition either to \( mState_2 \) or \( mState_3 \).

\( mState_2 \) matches conversion \( \text{Con}_2 \) to transition to \( mState_3 \), and \( \text{Con}_1 \) to transition to \( mState_0 \).

\( mState_3 \) matches conversion \( \text{Con}_2 \), to transition to itself, it also matches \( \text{Con}_1 \) to transition to itself or to \( mState_1 \) and the pending message can arrive to transition to itself or \( mState_2 \).

The graphical representation of this system can be seen in figure 9.4

![Figure 9.4: Macro States of infinite graphs](image)

By using our algorithm for detecting strongly connected components, see appendix B, on the graph we find one strongly connected component. This tells us that there is no deadlock it this system (within the provisos on accuracy discussed earlier in the chapter). We can also see that every possible conversion is used within this system, so there are no un-reachable chords\(^1\).

### 9.7.3 Macro-State Analysis Accuracy

Unfortunately, although we are now sure the analysis will complete, we have lost almost all of the information about the state of the queues (other than empty or not empty).

### 9.8 Conclusion

We show three examples of using our formalism to reason about chords. In the first example we are able to reason about the safety and liveness of a lock. In the second example we detected that some code in a particular class is un-reachable, and in the third example we show that a transition graph of infinite size can be reduced to a finite sized graph. We show there is no deadlock in the reduced system.

\(^1\)Since every possible transition within a strongly connected component is reachable from every state within that component.
This finite graph is bounded in size $O(2^n)$ where $n$ is the number of private channels in an object. In practise the number of reachable states in a graph may be much smaller than this. Because the graph is of finite size it should be possible for a compiler to automatically construct and analysis the macro-state graph.

Unfortunately we lose a large amount of information in the conversion from an infinite graph to a finite graph. In particular we lose all of the information on the number of messages in a queue (other than empty or not empty). Future work may be able to rectify this by using a combination of the state and macro-state transition graphs.
Chapter 10

Conclusions

If you cannot — in the long run — tell everyone what you have been doing, your doing has been worthless. — Erwin Schrödinger; Science and humanism

10.1 Evaluation of Objectives

We take each objective in turn and evaluate the completeness and effectiveness of our solution to it.

10.1.1 Description of chords

We hope to describe exactly what chorded-programming is and formally describe the properties of chorded programming.

We described chorded-programming and how it differs from Object-Orientated programing. In particular we made observations about idioms for encoding state in chorded-programs.

Chord programming is a new area, so there is very little literature on how to program in chords, and what good and bad chord programing style is. The author of this project has only been programming with chords for the past 6 months; so these observations are based on a position of relative inexperience.

It is likely that, if chord programing becomes popular, a more comprehensive list of idioms and patterns will become available. However, because the observations on chorded-programming style made in this report are such fundamental ones it is likely that they will still be valid for some time.

The formal description of chorded programs is quite accurate. It reflects well the structure and states of chorded-classes. It does have some missing features however:

- Asynchronous and synchronous channels are not distinguished between.

This reduces the accuracy of any analysis. For instance, if a chord called inside a body is synchronous, then we know that subsequent chords in the body cannot be called until it has matched.
• The types and possible-value-ranges of the parameters in the messages are totally disregarded.

It is possible that this information could be used to further reduce the size of the transition graphs.

It is also necessary to accurately reflect method over-loading in chorded-Java; at the moment this is not supported by the formalism (which requires names to be unique).

The formal description is independent of the underlying language, so could be applied to Polyphonic C♯ as well as chorded-Java; and possibly to future chorded-languages as well. We believe this is the first formal description developed specifically to describe chorded languages.

10.1.2 Compiler

We will then use this description, and the published literature, to develop a compiler for a chorded variation of Java.

The compiler works well; it is reasonably robust and fully functional. It supports the full Java syntax. We believe this is the first time a chorded-compiler has been made available publicly.

Some features of the compiler need more testing, for instance novel nesting of chorded inner-classes may throw up bugs (see appendix A).

The use of the MVC (model-view-controller) architecture separates the concerns of the compiler well. The model is fairly generic, presenting the information collected from the chorded-Java in an implementation independent way.

The view, which generates the code which implements the message marshaling, chord matching and chord dispatch, is pluggable. This allows the compiler to easily support multiple implementations of chords. It also allows the user of the compiler to provide her own implementation, with no recompilation, if required.

The use of Velocity removes the compile from the edit-compile-run cycle when developing new implementations of chord support. This reduces development time considerably.

The compiler is a pre-processor which translates chorded-Java to Java. A compiler which goes straight to bytecode would have several advantages:

• Less possibility for differences between the apparent semantics of the chorded-Java and the actual running code.

• More scope for optimisation of the resultant bytecode.

Less semantic information is lost before the bytecode is generated; a message queue would still be represented as a message queue, and not as an ArrayList as is currently the case.

Our compiler does not deal well with inheritance. A discussion of inheritance in chorded-languages can be found in appendix A.
10.1.3 Alternate Algorithm

We hope to provide alternative algorithms for implementing chords, and use our formal description to show that our implementation of chords is correct.

We provide two algorithms for implementing chords. The polyphonic algorithm is the algorithm from Polyphonic $C^\sharp$ and is implemented in a similar way. The simple algorithm is developed from our formal description and is significantly different from the Polyphonic $C^\sharp$ algorithm.

We are able to prove that our simple algorithm works correctly with only a single pass for chord matches. We prove this without needing to consider the details of the algorithm for running the selected match. We show that the algorithm is largely $O(n)$, although there is a bounded $O(n^2)$ operation involved.

It turns out that the Polyphonic $C^\sharp$ algorithm is of very similar complexity, however it has to scan for matches twice under some circumstances\[1\].

Our simple algorithm has to wake up threads (if any are queued) each time a synchronous message arrives — this is not the case for the Polyphonic $C^\sharp$ algorithm; sometimes it can dispatch a chord with no context switch.

We don’t know which algorithm is faster, since there is considerable scope for optimisation of both algorithms. In particular it is possible to pre-calculate the possible effects of some message arrivals, and also to inline some operations.

It might be possible or reduce the number of context switches in the simple algorithm, without compromising the simplicity or the single-scan property. Further work is required.

Testing and measurement in a large number of circumstances, such as load, hardware, Java Run-time and contention would be needed.

10.1.4 Possibility of Analysis

We will try to use our formal description framework to show how to reason about some properties of simple chord-programs, and to show that we can keep an analysis of chorded programs finite. We hope that this work could lead to the creation of a compiler which can analyse and optimise chorded-programs.

We are able to use our formalism to reason about some useful properties of chorded programs. We show examples of reasoning about safety and liveness properties, as well as the detection of unreachable code.

We show that it is possible for the transition graphs created in to become infinite in size.

However, by applying a transformation it is possible to keep the transition graphs within a finite upper bound, which we calculate as $2^{2n}$ in the number of private channels of an object. This means that it might be possible to implement automatic analysis of chorded programs in a compiler; possibly with static invariant checking.

Unfortunately, in the transformation from an infinite graph to a finite graph we lose a large quantity of information. Future work might be able to rectify this problem to some extent.
10.1.5 Distribution

We will also look briefly at how distribution effects our implementation and framework; including how to use chords with RMI, how the compiler can support the programmer in using this combination, and what effect chorded programming may have on RMI.

We described how channel names and message values could be handled using RMI. Mutable objects are collections of instances of names and are passed by reference. Immutable objects and primitives can be passed by value.

Java does not currently offer a way of marking an object as immutable, so it is up to the programmer to make sure that only immutable objects are serialised over RMI. This is not ideal, but difficult to fix without significant changes to the Java language.

We added a keyword to support distribution of chorded programs. In the current version this keyword causes the compiler to put in the correct superclass and exceptions to allow the class to create remote objects. This could be extended to support alternate distribution frameworks.

We created a demonstration program which solves the dining philosophers problem in a distributed setting. Some parts of the code, particularly calls to non-blocking (asynchronous) methods on remote objects were much easier to express in chorded-Java.

It is possible that RMI could be optimised to support asynchronous methods. Future work would is required

10.2 Contributions

The key contributions made by this project are set out in the evaluation above, and in bullet point form in the introduction (chapter 1).

10.3 Future Work

- Build up a fuller survey of chorded-programming style, idioms and patterns. The advantages (if any) of chorded programming over Object-Orientated programming should begin to emerge as more examples and applications are created.

- Add the missing features to formalism. Particularly support for distinguishing synchronous channels from asynchronous channels.

- Turn source code printer into a configurable separate source code formatter library.

- Add support for Java 1.5 to the compiler.

- Produce a chorded-Java to bytecode compiler. This should allow additional opportunity for optimisation, and reduce the possibility of unexpected implementation semantics.

- Determine if an efficient implementation of a fair dispatch algorithm is possible. Fair dispatch provides more intuitive properties to the chord programmer.
• Work on reducing the number of context switches in the simple algorithm. It is possible that if the number of context switches were reduced that the simple algorithm could perform much better than the polyphonic algorithm — however it might not be possible to find any reduction.

• Optimise and benchmark both algorithms in a variety of circumstances to see which one provides better performance. Both implementations are currently largely un-optimised so benchmarking results would be misleading without first optimising both algorithms.

• Support additional distribution frameworks besides RMI.

• Look at how distribution frameworks can be optimised for chorded programs, and particularly at the effects that the presence of “async” methods has on message passing in RMI.

• See if it is possible to preserve more information when converting from infinite state transition graphs to finite state transition graphs.

• See if analysis, optimisation and invariant checking can be added to the compiler using the principals set out in chapter 9.

• Look at inheritance in more detail. Some specific suggestions for the direction of such a study are given in appendix A.
Appendix A

Innerclasses and Inheritance

Implementers aren’t considered bozos anymore. John Sculley (1939 - )

There are a few features of our implementation that we have not covered in the main body of our report.

A.1 Inner-Classes and Privacy

Unfortunately Java does not provide an access modifier that can prevent inner-classes from accessing members of their enclosing class. This has implications for the safety of our translation from chorded-Java to Java; in particular a programmer could corrupt the state of a chorded object by using inner-classes to access private members of the generated class.

This problem cannot easily be fixed by making the compiler check the chorded-Java program for any such violations, due to the presence in Java of the reflection framework. Runtime checking would probably be needed. A future direct-to-bytecode compiler may be able to prevent this problem.

A.2 Chorded Inner-Classes

Officially, due to lack of time for testing, our chorded-compiler does not support inner-classes. However, some basic tests have shown that simple static chorded inner-classes work as expected.

The compiler uses a stack based, recursive, approach to handle innerclasses, starting a new compile pass for each one (except of course, not re-parsing the code).

A.3 Inheritance

Our compiler is very ambiguous about inheritance; it currently imposes no restrictions on the developer. The author spent some time thinking about inheritance in chorded languages without reaching a firm conclusion; some notes on the issues involved are presented here.
A.3.1 What is Inheritance for?

Inheritance in Java has two effects, it allows refinement of the source code and refinement of type.

A.3.1.1 Refinement of Source Code

Refinement of source code is used to reduce code replication. Instead of repeating the same code several times in classes that only vary the behaviour slightly, subclasses describe the changes and additional behaviour that distinguishes them from their superclass.

In Java inheritance can be used to implement abstract (undefined) behaviour, override virtual methods and add additional functionality.

A.3.1.2 Refinement of Type

Inheritance is also one of the mechanisms that Java uses to describe subsumption type relationships. A subclass can be used in place of any of its superclasses. Java also has a second mechanism; interface implementation. Implementing an interface allows the implementing class to be used anywhere the interface is used. Interfaces can be arranged into inheritance trees (interfaces can extend interfaces), including trees with multiple inheritance.

A.3.2 Problems with inheritance

A.3.2.1 Fragile Base Class

Inheritance is highly sensitive to changes in the base class. An example of this comes when a method is added to a super class.

```java
class A {
    public int a(int x) {
        return x*x;
    }
}
class B extends A {
    public int b(int x, int y) {
        return x+y;
    }
    public String c(int z) {
        return "The square of \( z \) is \( a(z) \);"
    }
}
```

1Virtual in this case being visible non-final methods.
Listing A.1: Inheritance Problem

In listing A.1 are two classes, “A” and “B”. Imagine “A” is in a library owned by me, and “B” is in an application written by you which uses my library. Because my library is a general one I have no idea that you have used my class “A” in this way.

For some reason I decided to change the implementation of class “A”.

```java
class A {
    public int a(int x) {
        return b(x, x);
    }

    public int b(int x, int y) {
        return x*y;
    }
}
```

Listing A.2: Inheritance Problem Two

When I ship this new version of my library your class “B” will break, since it has unintentionally overridden $b(\text{int} \ x, \ \text{int} \ y)$. Some languages get around this problem by requiring methods which are overriding to be explicitly labelled as such.

In general it is more reliable to wrap a class you wish to refine the behaviour of. In this way any changes in the behaviour of the wrapped class can be controlled and mediated by the wrapping class. An exception to this is when a class is explicitly designed for inheritance; these classes generally mark most of their methods as final and are abstract. Several design patterns use this meta-pattern, Template Method\[10\] is the clearest example of this.

A.3.3 Construction

Objects also suffer from inheritance problems when being constructed. If the constructor of a class (A) calls a virtual method which has been overridden (in B), then the writer of the sub-class B must be aware of it. If not the subclass writer may try to access members of the class B, which will not have been initialised by the time the constructor of A runs.

A.3.4 Inheritance in Polyphonic C\#\[1\]

Polyphonic C\# allows inheritance in a very restrictive way[1]. It is possible to override chord elements in a subclass from a super class, but if an element is overridden all of the elements it appears in a chord with must also be overridden. This gives a transitive closure effect, where all of the elements in the same system of chords must be overridden. You cannot, for instance, use the code in listing A.3.

---

\[2\]This is one example of the OO-design analogue of the fragile base class language implementation problem
class C
{
    virtual void f() & virtual async g()
    {
        // body1
    }

    virtual void g() & virtual async h()
    {
        // body2
    }
}

class D : C
{
    override async g()
    {
        // body3
    }
}

Listing A.3: Illegal Inheritance[1]

Here, class “D” must also override both f() and h(), so the whole contents of class “C” in fact. This is because if only g() is overridden then any calls to f() will block forever and calls to h() will be disregarded totally\(^3\).

A.3.4.1 Expressiveness

These restrictions limit the expressiveness of inheritance in polyphonic C\(^4\). It is difficult to modify the behaviour of a polyphonic object without having to override all of the elements (except in objects with multiple separate synchronisation systems\(^4\)).

To illustrate this problem of expressiveness here is an example of a bounded buffer[8] written in a Java like polyphonic language;

public class FIFO
{
    public FIFO(int size)
    {
        empty(new Object[size], 0, 0);
    }

    public void put(Object o)
    & private async empty(Object[] buffer, int i, int n)
    {
        put(o, buffer, i, n);
    }
}

\(^3\)ignoring memory concerns
\(^4\)and these type of objects may well be doing to much, perhaps they should be split into multiple smaller objects

132
public void put(Object o)  
   & private async some(Object[] buffer, int i, int n)
   {  
      put(o, buffer, i, n);
   }

private void put(Object o, Object[] buffer, int i, int n)
{  
    buffer[(i+n) % buffer.length] = o;
    check(buffer, i, n+1);
}

public Object get()  
   & private async full(Object[] buffer, int i, int n)
   {  
      return get(buffer, i, n);
   }

public Object get()  
   & private async some(Object[] buffer, int i, int n)
   {  
      return get(buffer, i, n);
   }

private Object get(Object[] buffer, int i, int n)
{  
    Object o = buffer[i];
    check(buffer, (i+1) % buffer.length, n-1);
    return o;
}

private void check(Object[] buffer, int i, int n)
{  
   if (n == buffer.length())
   {  
      full(buffer, i, buffer.length);
   }
   else if (n == 0)
   {  
      empty(buffer, n, i);
   }
   else
   {  
      some(buffer, n, i);
   }
}

Listing A.4: FIFO Inheritance Example

Of the two public methods, `get()` and `put()` either will cause the other to be overridden. This is the entire functionality to the class.

133
A.4 Inheritance in the Join Calculus[8]

The join calculus defines several inheritance operators, which we will look at in turn to see if they are applicable to chorded Java.

A.4.1 Disjunction

The Join-Calculus allows a subclass to disjoin an inherited class with additional reaction rules (or chords). Disjunction is purely a way of adding functionality, the new rules compete for messages with the old ones.

A.4.1.1 Applicability

This is a simple rule, similar to adding additional methods to a subclass. In a deterministic implementation chords must be disjoint from chords in the superclass\(^5\).

A.4.1.2 Example of disjunction

```java
class A {
    public A() {
        empty();
    }

    public async put(Object o) & protected async empty() {
        contains(o);
    }

    public Object get() & protected async contains(Object o) {
        empty();
        return o;
    }
}

class B or A {
    public boolean isEmpty() & protected async empty() {
        empty();
        return true;
    }

    public boolean isEmpty() & protected async contains(Object o) {
        contains(o);
    }
}
```

\(^5\)Otherwise the duplicate chords would never be matched.
Listing A.5: Disjunction Inheritance Example

Class “B” adds an extra method, isEmpty(), to the single place buffer represented by class “A”. The class “B” could equivalently be written as:

class B {
    public B () {
        empty () ;
    }

    public async put ( Object o ) & protected async empty () {
        contains ( o ) ;
    }

    public Object get () & protected async contains ( Object o ) {
        empty () ;
        return o ;
    }

    public boolean isEmpty () & protected async empty () {
        empty () ;
        return true ;
    }

    public boolean isEmpty () & protected async contains ( Object o ) {
        contains ( o ) ;
        return false ;
    }
}

Listing A.6: Equivalent Representation of B

A.4.2 Other Join inheritance operators

There are two other operations in the Join Calculus related to inheritance, although in the Join Calculus they are implemented by the same operator we will treat them separately. This leads us to more longhand way of writing out programs, but the separate operators are simpler to understand.

A.4.2.1 Rename

This operator renames an element in the superclass. This is equivalent to renaming a channel, and can be used to wrap the superclass methods. The changes late bind, so any calls of the element
in the superclass will still try to bind to the old name. If an alternate element is not provided (via
disjunction or another rename) then a compile failure will occur.

In chorded Java a renaming operation could also change the visibility of the element. Again this
decision is designed to support a wrapping style of inheritance rather than a replacement style. A
(non-chorded) example illustrates how this is similar to method over-riding.

class C
{
    public void printName(String name)
    {
        System.out.println(name);
    }
}
class D or C
{
    rename [public void printName(String)]
        to [private void printSimpleName(String)];

    public void printName(String name)
    {
        printSimpleName("Name: " + name);
    }
}

Listing A.7: Rename Operator

Which in standard Java could be written as;

class C
{
    public void printName(String name)
    {
        System.out.println(name);
    }
}
class D extends C
{
    public void printName(String name)
    {
        super.printName("Name: " + name);
    }
}

Listing A.8: Rename is Overriding++

A.4.2.2 Match

This operator adds additional code to an existing chord. A second, async, body is added. This
example shows the use of match to add logging code to a superclass;
class E {
    public void printName(String name) {
        System.out.println(name);
    }
}

class F or E {
    match [public void printName(String name)] {
        log("Printing " + name);
    }
}

Listing A.9: Match Operator

Which could be written in a Java-like pseudo code as

class E {
    public void printName(String name) {
        System.out.println(name);
    }
}

class F extends E {
    public void printName(String name) {
        run in a new thread: log("Printing " + name);
        super.printName(name);
    }
}

Listing A.10: Match Operator is Overriding

A.4.3 Subsumption

The suggested notion of inheritance in chorded-Java only allows refinement of source code, it does not automatically create a subtype/supertype relationship. To allow subsumption the programmer must explicitly use interfaces. To continue the buffer example from above;

class A implements 11 {
    [...]
}

class B implements 12 {
}
interface I1
{
    async put(Object o);
    Object get();
}

interface I2 extends I1
{
    boolean isEmpty();
}

Listing A.11: Subsumption and Disjunction Inheritance

A.4.4 Conclusion

More sophisticated inheritance support, than that provided in Polyphonic C♯ is possible in chorded programming; future work in this area might prove interesting.

In Java inheritance for type-subsumption is confused with inheritance for code reuse.
Appendix B

Finding strongly connected components

It is possible to identify the strongly connected components of a graph using a well-known $O(n)$ algorithm[29]. We describe the algorithm here for the convenience of readers.

B.1 What is a strongly connected component

A graph is strongly connected[22] if for all nodes there is a path to every other node. Or for a graph $G = (V, E)$, $\forall v_i, v_j \in V$ there is a path from $v_i$ to $v_j$ and a path from $v_i$ to $v_j$.

The strongly connected components of a graph are its maximal strongly connected subgraphs[13].

The strongly connected components partition the graph (no vertex is in two strongly connected components).

B.2 Algorithm for discovering strongly connected components

B.2.1 The Depth-First-Search algorithm

$G$ is a graph containing nodes $V[G]$, and $\text{Adj}[u]$ for some $u \in V[G]$ is the set of vertexes adjacent to $u$.

We attached data to each vertex as we go through the graph. $\text{state}[u] \in \{\text{visited}, \text{unvisited}\}$, all vertexes are initially unvisited. We also store the discovery time $\text{discover}[u] \in \mathbb{N}$ and the finish time $\text{finish}[u] \in \mathbb{N}$. Both times are initially zero.

Finally, for each node we store its parent in $\text{parent}[u] \in V[G]$. Initially the parent is nil.

We can optimise the algorithm B.1 by using a list of nodes in place of $V[G]$ and removing them when they are marked visited (this saves unnecessary work in the outer loop).

B.2.2 Using DFS to find the strongly connected components of a graph

The algorithm B.2 can be used to find the strongly connected components of a graph $G[29]$.
DFS(G:Graph)
  time = 0
  foreach u ∈ V[G] do
    if state[u] = unvisited then DFS-VISIT(U)

DFS-VISIT(U:Vertex)
  state[u] = visited
  discover[u] = time
  time+ = 1
  foreach v ∈ Adj[u] do
    if state[v] = unvisited then
      parent[v] = u
      DFS-VISIT(v)
  finish[u] = time
  time+ = 1

Algorithm B.1: DFS algorithm

1. Run DFS(G)
2. Compute the transpose of G (G^T)
3. Run DFS(G^T), but order V[G^T] according to decreasing finish times as calculated in (1).
4. The vertexes of each tree in the forest of trees found by DFS(G^T) is a strongly connected component.

Algorithm B.2: Strongly connected components algorithm
B.3 Reducing graphs to DAGs using strongly connected components

Once we have found the strongly connected components of a Graph (G) we can use the analysis to create a reduced graph. The graph where each node is one of the strongly connected components.

Lemma B.1 (Acyclic Components) If two vertexes are in the same strong component then no path between them ever leaves the component[29].

So if we draw a graph, E, of the strongly connected components, such that for any two strongly connected components \( u, v \) there is an edge from \( u \) to \( v \) in \( E \) if and only if there is an edge from one of the nodes in the strong component \( u \) to one of the nodes in the strong component \( v \) in the original graph \( G \).

Figure B.1 illustrates. By Lemma B.1 we know that this reduced graph will be a DAG (Directed Acyclic Graph)[13].

Figure B.1: Reducing a graph using strong components
Appendix C

Notes on Scheduling

We need to describe some assumptions about the scheduler running our chord system.

**Axiom C.1 (Safe Scheduler)** The scheduler will always be able to pick at least one task to schedule — even if it is an empty idle process.

**Axiom C.2 (Live Scheduler)** If the scheduler runs for an infinite number of execution steps, every schedulable (ready) process will be scheduled for an infinite number of execution steps (unless the process requires less than an infinite number of execution steps to complete - in which case it will be scheduled for that many execution steps)

**Axiom C.3 (Fair Scheduler)** If a scheduling choice is executed an infinite number of times, then each possible choice will be chosen an infinite number of times

These axioms ensure that the scheduler will not add any deadlocks to the system.
Appendix D

Indexing channels

This chapter is a brief justification of, and explanation of the mathematics behind, our use of index subscripts to identify channels, bodies and queues.

We describe channels and bodies within a class with a numerical subscript (index). This subscript refers to the position of the channel within the infinite set Channel (or, more precisely, is based on the ordinal of the elements of some finite subset of Channel).

**Lemma D.1 (Unique Subscripts)** The subscripts are unique; no two channels have the same subscript and are in the same class.

**Lemma D.2 (Contiguous Subscripts)** The subscripts are contiguous and range from 1,...,m where $m = |\text{channels}(t)|$.

**Lemma D.3 (Well Defined Subscripts)** The value of the subscript is described by the function $\text{index} : \text{Channel} \rightarrow \mathbb{N}$

In particular, since the set Channel, is well ordered (see 4.4.1) all subsets of Channel, have a first element (which we will call $c_0$). We can then define the index in the set $\text{channels}(t) = \{c_1, \ldots, c_m : \text{Channel}\}$, which by the definition of Channel, is a subset of Channel,

\[
\text{index}(c) = 1 \iff c = c_0 \\
\text{index}(c) = i \iff c \neq c_0 \land \exists c' \in \text{channels}(t) : (\neg \exists c'' \in \text{channels}(t) : (c' < c) \land (c'' < c) \land (c' < c'')) \land i = (\text{index}(c') + 1)
\]

Proof Sketch:

1. If there is only one item the index equation is trivially correct (that item must be the least item so is $c_0$).

2. For any $c$ other than $c_0$ we know that there is some $c'$ also in the set that proceeds it (since Channel, is well-ordered, there is a linear order on the elements in any subset of Channel,). And $c_0$ is before every other element (by definition).
3. For any $c$ other than $c_0$ we know that there is exactly one element $c'$ which is immediately before $c$. That is, there is no other element $c''$ which is larger than $c'$ but smaller than $c$. This is also a property of linear order. Because we are dealing with a subset of the linearly ordered set, there may be elements in the order but not present in the subset which lie between $c'$ and $c$. However, by (2) there must be an element which is less than $c$ and is present (because $c$ is not $c_0$).

4. Each element (except the first) is given the index of the previous element incremented by 1. Each element except for the last has exactly one element as its immediate follower (again by the properties of linear order) so this index is unique. The index is also contiguous, since every element (except $c_0$ and the last element) has exactly one immediate predecessor and one immediate follower. It follows that the maximum index is the size of the set (each additional element added increases the highest index by 1).

A similar argument applies to bodies. We use this subscript in our definition of queues and in our optimised pattern matching algorithm.
Appendix E

γ augmentation

γ augmentation is a notational way of making sure the sets returned by calledChannels remain finite. It was invented by the author specifically for this purpose.

It is used for counting the number of times a particular channel can be called. Instead of mapping to some set \( \mathcal{P}(\text{Channel}^\gamma) \), the calledChannels relation maps to an augmented set. Each member of the set \( \text{Channel}^\gamma \) is prefixed by some \( n \in \mathbb{N}^\infty \), representing the number of times the channel can be called from the given Code.

E.1 Syntactic Sugar

If we were using \( \mathcal{P}(\text{Channel}_0) \), then the element not being present at all would be equivalent to augmenting the element with 0 in the augmented set. For this reason, we can optionally write the set \( \text{Channel}^\gamma \) down without any elements that are augmented with zero (they are there really, we just don’t write them to save space).

For example, the following Java-like code

```java
class A {
    public void a() & private async b() {
        // method 1
        c();
    }

    public void a() & private async c() {
        // method 2
        while (someCondition) {
            [...] // adjust someCondition
            b();
        }
        c();
    }
}
```
would produce a set augmented as follows;

calledChannels(method 1) = \{0 : (A, a), 0 : (A, b), 1 : (A, c)\}
calledChannels(method 1) = \{1 : (A, c)\}
calledChannels(method 2) = \{\infty : (A, b), 1 : (A, c)\}
calledChannels(method 2) = \{0 : (A, a), \infty : (A, b), 1 : (A, c)\}

E.2 Justification of $\gamma$ augmentation

$\gamma$ augmentation is designed to make sure that our set of called channels remains finite, even under the circumstances where a channel might be called an infinite number of times. As we will see later, we need our augmented set to remain finite (or at least any instances of our augmented set that we can use), so that we can be sure our analysis will terminate.

Instead of using $\gamma$ augmentation, there are two obvious ways we could have counted the number of possible calls to a channel.

1. Use the set $\mathbb{N}^\infty \times \text{Channel}$. However, that set of tuples is infinite (since $\mathbb{N}^\infty$ is infinite). We show in the following subsections that, in fact, our augmented set can be described by construction on finite subsets of $\mathbb{N}^\infty \times \text{Channel}$. And our augmentation notation is a convenient way of making sure that we don’t perform non-cardinality preserving operations on the subsets of $\mathbb{N}^\infty \times \text{Channel}$.

2. Use the set of sets $\{x\}$, $\{x, \{x\}\}$, $\{x, \{x\}, \{x\}\}$
   . . . for each channel. This is equivalent, since $\mathbb{N}^\infty$ can be constructed using this technique.

We would like that wherever $X$ is a finite set (for instance where $X = \text{Channel}$), all the instances of $\gamma$ augmented $X$ are also finite.

We can decide if any given $\gamma$ augmented set is finite by seeing if it obeys the following rule;

For some given $x : X\gamma$

$$\forall n : u, m : v \in x : u = v \rightarrow n = m$$

In English; $x : X\gamma$ has at most one occurrence\(^1\) of each element of the set $X$. If there is an occurrence of an element from the set $X$, that element will be augmented by exactly one number.

A more precise definition using the set $\mathbb{N}^\infty \times X$. For some $x \subseteq \mathbb{N}^\infty \times X$, $x$ is $\gamma$-finite\(^2\) if,

$$\forall (n, u) \in x : \neg \exists (m, v) \in x : v = u \land n \neq m$$

We know this set is finite, since it only contains at most one element for each element of $X$, and $X$ is finite (by definition).

---

\(^1\)In fact, this is more to do with the notion of what a set is than anything else

\(^2\)There are other finite subsets of $\mathbb{N}^\infty \times X$, but we are not interested in them
We can map each $\gamma$ augmented set $(x_1 : X^\gamma)$ to a set $(x_2 \subseteq \mathbb{N}^\infty \times X)$ meeting the condition above using

$$x_2 = \{(n, u) | u \in x_1\}$$

we donate this transformation $\gamma^{-1}$ and vice-versa

$$x_1 = \{n : u | (n, u) \in x_2\}$$

we donate $\gamma$, where $\gamma(x_2) = x_1$.

Thus the maximum number of elements present in any $\gamma$ augmented set, is the same as the maximum number of elements in the set being augmented (in our example $|X|$). In fact, this is also the minimum number, since the absence of an element from $X^\gamma$ is equivalent to it being there but augmented with 0.

Now that we know how to tell if a subset of $\mathbb{N}^\infty \times X$ meets our condition and can be mapped to a finite $X^\gamma$. We need to show that the calledChannels function can be implemented. And that we can use the result correctly (that is in a finite way) in our analysis later.

We define the following operations over $\gamma$ augmented sets, which preserve the cardinality (and hence the finiteness) of the operands in the result.

We will first define the operators, then describe how they can be used to implement calledChannels and also give a brief discussion on how they can more generally be used to manipulate sets of the type Channel$^\gamma$. 

E.2.1 ⊔

⊔ is analogous to the set union operation.

$$x_1 : X^\gamma \sqcup x_2 : X^\gamma =$$

$$\{n : u | (n : u \in x_1 \land m : u \notin x_2) \lor (n : u \in x_2 \land m : u \notin x_1) \lor (m : u \in x_1 \land m' : u \in x_2 \land n = \min(m + m', \infty)\}$$

Since we have already established (or more to the point, decided) that each of $x_1, x_2$ have at most one occurrence of each $x \in X$, we have three cases. The element occurs in either one, both or neither of the sets being ⊔.

- Both. The first and second disjunctions are false. But the third holds. We end up with a single occurrence of the element in the resultant set — augmented by exactly the maximum of the augmentation in the two contributing sets ($x_1, x_2$).

Interestingly, this is the also the case where it is possible that commutativity of the ⊔ operator could be broken. It isn’t (since the order of the arguments to + make no difference to the result). However, this isn’t particularly important since we never use commutativity in our analysis.

- One. Either the first or second disjunction is true. The remaining two disjunctions are false. So exactly one element is present in the resultant set.

Strictly speaking however, this situation can’t happen. Since we have already stated that the absence of an element from $X^\gamma$ is equivalent to it being there but augmented with 0. Intuitively the behaviour is like the presented comprehension.

149
• Neither. The set comprehension trivially does not include this element (since none of the disjunctions can be true).

This situation never occurs either, for the same reason as above.

Because the last two situations cannot occur, the cardinality of the result will be the same as the cardinality of both \( x_1 \) and \( x_2 \). The following definition is equivalent:

\[
x_1 : X^γ \sqcup x_2 : X^γ = \{ n : u \mid (m : u \in x_1 \land m' : u \in x_2 \land n = \min(m + m', \infty)) \}
\]

**E.2.2**

\(-\)

\(-\) is analogous to set difference. Normal set difference is defined as \( A - B = \{ x \mid x \in A \land x \notin B \} \). We need to take into account augmentation whilst preserving cardinality.

\[
x_1 : X^γ - x_2 : X^γ = \{ n : u \mid m : u \in x_1 \land m' : u \in x_2 \land n = \max(m - m', 0) \}
\]

The result of the comprehension is every element from \( x_1 \) with its original augmentation, except where the element is in \( x_2 \) augmented by a number other than 0, in which case we get the element from \( x_1 \) augmented by a function of its augmentation in \( x_2 \) and \( x_1 \). So the maximum size of the result is \( |x_1| \).

There are two interesting properties of this operator.

1. It is impossible to reduce an element augmented with \( \infty \) in the first operand to any other augmentation in the result. This is as it should be given the intuition of infinity.

2. An element in the second operand augmented with \( \infty \) will reduce the augmentation of the corresponding element in the first operand to zero in all cases where property (1) does not apply.

**E.3 Usage of \( \gamma \) augmentation**

First we must show that calledChannels can be implemented using only the operators we have defined.

**E.3.1 Implementing calledChannels**

Here is a simple algorithm for implementing calledChannels in our Java-like language. For a given body of code, the following can be applied:

```
calledMethods = \( \gamma(\{0\} \times \text{Channel}) \)
foreach channel called in the code
  if (inloop(channel))
    calledMethods = calledMethods \cup \{(\infty, \text{channel})\}
  else calledMethods = calledMethods \cup \{(1, \text{channel})\}
```
Since both \([0]\) and Channel, are finite, \(\gamma([0] \times \text{Channel},)\) is finite. It is also \(\gamma\)-finite, since \([0]\) has only one element, the cross-product draws only one pair for each element in Channel,. Thus the initial value of calledMethods is \(\gamma\)-finite. The only operation applied to calledMethods is \(\cup\), which is cardinality preserving. It is only ever applied with calledMethods as one operand and a one element set (actually, the other elements of \(\gamma([0] \times \text{Channel},)\) are there, we just don’t write them), so the result of the algorithm must be finite.

Due to time constraints, we don’t look at this algorithm in detail, but it gives an intuition as to how the cardinality preserving operators are used.

E.3.2 The analysis

We will see in our analysis that this cardinality preservation of \(\cup\) and \(\cap\) on finite \(\gamma\) augmented sets helps to show that our algorithm terminates.

By only ever constructing finite \(\gamma\) augmented sets, and then using the cardinality preserving operations on them we can be sure to preserve finiteness.
Appendix F

Notes on Notation

There is nothing that can be said by mathematical symbols and relations which cannot also be said by words. The converse, however, is false. Much that can be and is said by words cannot successfully be put into equations, because it is nonsense. — C. Truesdell (1919-2000)

F.1 Types

F.1.1 boolean

\[ boolean \in \{ \text{true, false} \} \]

F.1.2 Infinite numbers

We use \( \mathbb{N}^\infty \) to illustrate values which we expect to be natural numbers or infinity. We explicitly lift the set \( \mathbb{N} \) to include \( \infty \) when \( \infty \) is a valid value.
Appendix G

Scenebeans

Scenebeans is a Java framework for building and controlling animated graphics[24]. During the course of implementing the chorded-Java compiler, it was suggested by Professor Magee that the author look at scenebeans as a way to animate some examples we were working on at the time.

Although scenebeans is a powerful framework for producing customisable animations in Java, we discovered that scenebeans did not have several features we felt were required. In particular;

- Scenebean macro-expansion was not powerful enough. The number of objects in a Scenebean animation was difficult to customise, particularly because scenebeans did not support variables or indirection. This lack of indirection meant that scenebeans macros could not be data structures but only flat values.

- Scenebeans did not support Java-Swing, it used a heavy-weight AWT (abstract windowing toolkit) component for its output, making it unsuitable for using with modern Swing applications.

- The external synchronisation required to use scenebeans animations was clumsy and error prone (although possibly quite fast).

Although we did not end up using Scenebeans, at the time we were evaluating it using it seemed likely. So we decided to rectify these missing features.

G.1 The pre-processor

Scenebeans uses an xml format to describe the animations it draws. When the xml file is read in by the Scenebeans tool it runs it through a pre-processor. That pre-processor does several things;

- Expands loops. Scenebeans allows a long section of repetitious xml to be replaced by a loop which generates it. For example;

```xml
<forall var="N" values="0,1,2,3,4,5,6,7" sep="",
> <behaviour algorithm="move" id="move-1-\$[N]">
  <param name="from" value="0"/>
</behaviour>
</forall>
```
Listing G.1: A loop in xml

Creates 8 copies of the behaviour, each one with the “${N}” replaced by the number from the “values” attribute. The “values” attribute is split according to the character in “sep”.

- Replaces macros. Scenebeans provides a Java API which allows the values of macros to be defined in the Java code which calls the Scenebeans system. This allows the user of the animation to define certain features (perhaps the strings to label parts of the animation with).

G.2 The macro-expander

We decided to add several new features to the macro expander. These features hopefully add significantly to the power and expressiveness of the xml file format used by scenebeans.

G.2.1 Variables

Scenebeans had no-way of storing a calculation for later use. That meant that if a calculated value was needed more than once it had to be recalculated each time. This causes two problems:

1. The calculation has to be performed multiple times.
2. The calculation has to be written out several times in the xml file. This causes maintainability problems, since if the calculation needs to be updated it has to be changed consistently in several places.

To fix this a new xml data type was added;

```xml
<where var="something" value="some${thing}"/>
</where>
```

Listing G.2: Macro variable

Which can be used anywhere a forall can, and works like an eval() or similar — it puts the expanded result of “some${thing}” into the macro defined in var (called “something” in this case).

This instruction create a new macro or assigns over an existing one. A variation on the instruction can be used to create a scoped macro, which will only be valid between the start and end tags, and will not overwrite an existing macro;

```xml
<where var="something" value="some${thing}" newScope="true"/>
</where>
```

Listing G.3: Macro variable in new scope
G.2.2 Multiple dereferences

To support data-structures in macros, some form of indirection or pointer support was needed. We decided to allow macros to contain macros. The inner-most macro is expanded first, \"${someMacro}\" and \"${{hello\{world\}}}\" and \"${{someMacro}}\" are examples.

For example, if the macro \"someMacro\" contained the value \"anotherMacro\", and the macro \"anotherMacro\" contained the value \"purpleElephant\". Then the expression \"${someMacro}\" would expand to \"purpleElephant\".

This allows the creation of data structures, for example here is some code containing two associative arrays (or tables). The expression \"{linkFrom,${link}}\" looks up the element which \"${link}\" evaluates to in the table \"linkFrom\". Similarly \"{message,${message}}\", looks up the element which \"${message}\" expands to in the table called \"message\". These tables can be setup by the user before the xml is parsed, giving an ability to make powerful customisations to an animation.

Much more complex data-structures are possible by using levels of indirection.

\[
\begin{align*}
&<\text{forall } \text{var} = \text{"link"} \text{ values = \"${\text{links}}\"}> \\
&<\text{forall } \text{var} = \text{"message"} \text{ values = \"${\text{messages}}\"}> \\
&<\text{command}> \\
&\text{name} = \"send,${message},${linkFrom},${link},${linkTo},${link}\" > \\
&<\text{set } \text{object} = \text{"messageText} - \text{${link}}\" > \\
&\text{param} = \text{"text"} \\
&\text{value} = \"{message,${message}}\" > \\
&<\text{reset } \text{behaviour} = \text{"doMessage} - \text{${link}}\" > \\
&<\text{start } \text{behaviour} = \text{"doMessage} - \text{${link}}\" > \\
&<\text{command}> \\
&</\text{forall1}> \\
&</\text{forall}> \\
\end{align*}
\]

Listing G.4: Data structure

G.2.3 List size operation

In the original scenebeans, although it was possible to define a loop, there was no way of obtaining the number of iterations that loop would make. This meant that it was impossible, say, to write a loop which would lay a series of elements out evenly in a circle (since the gap between each element is defined by the total number of elements).

The list size operation looks like, \#1[abcde], or \#1[abcdef]. And will evaluate as the number of elements in the second \{\} when split with the regular expression in the first \{\}. So in this example 6.

When combined with the multiple macro expansion this can be used as \#1[\{someList\}].

G.2.4 The re-write

Making these changes meant adding support for the new xml structures to the xml parser. The xml parser was easy to understand as written, so this didn’t pose too much of a challenge.
Adding the additional features to the macro parser was a little more complex, and involved largely re-writing the macro-expander. The macro-expander was well encapsulated in a single class, so it was easy to change the implementation without having a ripple through effect on the rest of the code.

To maintain this encapsulation we used private static inner classes to split the functionality out into a strategy pattern[9], as shown in figure G.1. This allowed a recursive implementation of the macro parser, with the strategy changed at each recursion level (depending on exactly what was being matched). This recursion mechanism was needed to cope with the potentially arbitrarily deep nesting on macros in the multiple dereference support.

This makes the expansion slower, and we haven’t gone out of our way to make it fast. However, the macro-expansion is not the determining factor in loading an animation, so the performance change was not noticeable by a user.

![Figure G.1: The Macro Expander](image)

The new macro-expander behaviour is mostly backward compatible with the old one. However, if programmers have used macros which look like multiple expansion (which is unlikely but possible) in old applications these will break.

### G.3 Updating from AWT to Swing

Thanks to the use of delegation in the original scene beans it was relatively straight-forward to refactor the GUI code to support both Swing and AWT.

Figure G.2 shows a UML depiction of the original architecture of the GUI component (Animation-Canvas) provided by the original scenebeans utility. The Animation is the interface to the beans
representing the animation. The \textit{WindowTransform} provides the code to transform the animation to fit the window.

The \textit{AnimationCanvas} is responsible for the animation of the animation, it controls the thread and the pause/resume behaviour.

\begin{figure}[h]
\centering
\includegraphics[width=0.8\textwidth]{fig.png}
\caption{The Original AWT Canvas}
\end{figure}

The objective was to add support for Swing, but without breaking existing applications. To reduce code replication and redundancy we decided to refactor the existing \textit{AnimationCanvas} into a delegate class. It quickly became clear during this process that the \textit{AnimationCanvas} was preforming several diverse tasks. We decided to split these tasks out into several different classes (see figure G.3).

- Drawing. The drawing and image management code was moved into \textit{AnimationDelegate}.
- Animation. The animation of the animation, for instance the frame rate management, was moved into \textit{AnimatorDelegate}.
- Thread handling. The thread management was moved into a state pattern held in a collaboration of inner classes inside \textit{AnimatorDelegate}.

This allowed a clear view of the synchronisation of the animation calculation. Three problems became apparent.

- Thread being started in a constructor. This will break the semantic guarantees provided by the final keyword in the Java 1.5 memory model\cite{20}.
- The animation requires several methods to be externally-synchronized. These methods must always be synchronized externally. And could cause serious bugs if a developer using the library forgets or becomes confused. This is error prone.

Although this external synchronisation might provide some speed increases, it was felt that these speed gains were overridden by the extra programming difficulty.
• Synchronized `paint()` method. Since some applications cause the `paint()` method of a component to be called very often indeed (which is why it care must be taken in a swing component not to recalculate the image of components which haven’t changed). Synchronized methods can be much slower than non-synchronized methods\[11\], so it is not a good idea to have a synchronized paint method. It is a better strategy to indirect updates to the data structure through the Swing event thread (which is how the swing MVC framework is designed).

It became clear that without more significant changes to the structure of the animation GUI components, it would be difficult to moved to a true MVC (or even the swing style Model and combined View/Controller architecture). Some of the calculations are preformed in what should be the view.

The first two problems were easy to fix. We now require the animation to be explicitly started after the GUI component is constructed. This has the nice side-effect of not causing the GUI to be accidentally realised before construction has finished\[1\].

The external synchronisation was easily fixed, by synchronising all accesses to mutable data internally.

To unsynchronize the `paint()` method, we decided to indirect some of the calculations into the swing event thread using “SwingUtilities.invokeAndWait(Runnable)”. This reduced the overall amount of synchronisation needed (significantly, in the case of `paint()`). However, it is a trade off; applications that repaint often will be faster, but applications that do very big calculations in the animation might get laggy. I expect the first class of applications to be larger. Although only time and testing could answer that.

This is not the best solution. A refactoring to a true modified MVC structure would be better. However that would require considerably more time, and we understand that an entirely new version of scenebeans is in development.

G.4 Using Scenebeans to visualise networks

To show the power of these changes to scenebeans, we decided to create a network visualiser capable of being configured with an arbitrary topography, colours, message types and node status messages.

We created a Scenebeans animation configurable with these features by using the extensions to the macro-processor; they proved invaluable, it would have been very difficult (and probably impossible) to create such a configurable animation without the extensions.

G.4.1 The API

We then designed an API to wrap the animation with a friendly representation of the network.

The network is presented as a collection of nodes and links; these (along with status, messages and colours) are passed into the context of the scenebeans animation using the extended macro-processor. The network can easily be configured to have different connectivity and nodes; uni-directional and bi-directional links are supported (figure G.5).
Figure G.3: The updated AWT Canvas, and Swing Panel

Figure G.4: Network Visualisation API
Figure G.5: Nodes and links can easily be configured

Figure G.6: A message being sent

Figure G.7: Connections can change colour
Message sending shown in figure G.6. The operations on NetworkAnimation can then be used to trigger events, like colour change to indicate link-status (figure G.7). Nodes can also change colour, and status labels can be added (figure G.8). All of the animation events are asynchronous, allowing them to animate a simulation without serialising it; notification of completion of events is also supported.

Figure G.8: Nodes can change colour and carry a status label
Bibliography


