INTRODUCTION OF HIGH LEVEL CONCURRENCY SEMANTICS IN OBJECT ORIENTED LANGUAGES

By
G Stewart von Itzstein
Bachelor of Computer and Information Science (Honours)

A thesis submitted for the Degree of
Doctor of Philosophy

31st December 2004
Adelaide
South Australia

Reconfigurable Computing Laboratory
http://rcl.unisa.edu.au
School of Computing and Information Science
University of South Australia
http://www.vonitzstein.com/research

University of South Australia

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGEMENTS

There are a number of people and organizations that have contributed to both the research and to the author's capability to complete this thesis.

A number of organizations contributed to my research. Firstly, Sun Microsystems for the initial equipment I used for my development of the compiler prototype. The Ecole Polytechnique Federale De Lausanne (EPFL) that partly funded my visits to Switzerland. The Sir Ross & Sir Keith Smith Trust for additional travel assistance. The School of Computer and Information Science at the University of South Australia for time and material support especially during the final part of my PhD. I would also like to thank the Australian Government and the Australian people for funding me whilst undertaking both my undergraduate and postgraduate education.

To my family, Dallas Hohl and my brother in law Mark Hohl thanks for being there for me. To Dad thanks for all those days I should have been out building fences. I would also like to thank Uncle Mark for being a role model whilst I was studying.

To Ben Avery and Ron Graml thank you for being such great friends. To my friends and colleagues at the university Grant Wigley for our political discussions, Dr Wayne Piekarski for being pushy, Phillipa Osborne for the chats, Justin Taylor for the moral support, Greg Warner, Malcolm Bowes to technical support. Frank Fursenko for those coffees in the bar. Jim Warren for his supervision of my honours thesis. To my old flat mates Wayne Jewell and James Abraham thanks for putting up with me.

I would also like to thank my old boss, Professor Brenton Dansie and my new boss, Professor Andy Koronios. I would also like to thank the general staff within the department without you the school would not function. Thank you for all the assistance with my teaching and research. Jo Zucco and Kirsten Wahlstrom thank you for giving me the time to finish my write up.

I would also like to thank Dr Christoph Zenger and Professor Konstantin Laufer who marked my thesis. Thank you for the extensive feedback that contributed to the quality of the final product. Also to Sue Tyerman, for the last minute editing. I would also like to thank Professor Martin Odersky for the initial direction for this thesis, Dr Matthias Zenger for his significant technological assistance. Finally, the author wishes to especially thank Dr David Kearney for his supervision.

But like all systems, it has a weakness. The system is based on the rules of a building. One system built on another. If one fails, so must the other. (Wachowski 2003)

ABSTRACT

Concurrency expression in most popular programming languages is still quite low level and based on constructs such as semaphores and monitors, that have not changed in twenty years. Libraries that are emerging (such as the upcoming Java concurrency library JSR-166) show that programmers demand that higher-level concurrency semantics be available in mainstream languages. Communicating Sequential Processes (CSP), Calculus of Communicating Systems (CCS) and Pi have higher-level synchronization behaviours defined implicitly through the composition of events at the interfaces of concurrent processes. Join calculus, on the other hand has explicit synchronization based on a localized conjunction of events defined as reduction rules. The Join semantics appear to be more appropriate to mainstream programmers; who want explicit expressions of synchronization that do not breach the object-oriented idea of modularization. Join readily expresses the dynamic creation and destruction of processes and channels which is sympathetic to dynamic languages like Java.

The research described here investigates if the object-oriented programming language Java can be modified so that all expressions of concurrency and synchronization can use higher-level syntax and semantics inspired by the Join calculus. Of particular interest is to determine if a true integration can be made of Join into Java. This work seeks to develop a true language extension not just a class library. This research also investigates the impact of the Join constructs on the programming of well-known concurrency software patterns including the size and complexity of the programs. Finally, the impact on the performance of programs written in the language extension is also studied.

The major contribution of the thesis is the design of a new superset of Java called Join Java and the construction and evaluation of the first prototype compiler for the language. The Join Java language can express virtually all published concurrency patterns without explicit recourse to low-level monitor calls. In general, Join Java programs are more concise than their Java equivalents. The overhead introduced in Join Java by the higher-level expressions derived from the Join calculus is manageable. The synchronization expressions associated with monitors (*wait* and *notify*) which are normally located in the body of methods can be replaced by Join Java expressions (the Join methods) which form part of the method signature. This provides future opportunities to enhance further the inheritance possibilities of Join Java possibly minimizing the impact of inheritance anomalies.

DECLARATION

I declare that this thesis does not incorporate without acknowledgment any material previously submitted for a degree or diploma in any university. I also declare that to the best of my knowledge it does not contain any materials previously published unless noted below, or written by another person except where due reference is made in the text

Some of the material in this thesis has been published in the following papers.

A brief introduction of the ideas covered in this thesis appeared in

> *Itzstein, G. S. and Kearney, D (2001). Join Java: An Alternative Concurrency Semantic for Java, University of South Australia Report ACRC-01-001.*

Parts of the concurrency applications chapter appeared in;

> *Itzstein, G. S. and Kearney, D (2002). Applications of Join Java. Proceedings of the Seventh Asia Pacific Computer Systems Architecture Conference ACSAC'2002. Melbourne, Australia, Australian Computer Society: 1-20.*

Some of the implementation chapter appeared in;

> *Itzstein, G., Stewart and Jasiunas, M (2003). On Implementing High Level Concurrency in Java. Advances in Computer Systems Architecture 2003, Aizu Japan, Springer Verlag.*

Parts of the design patterns chapter appeared in;

> *Itzstein, G. and Kearney, D. The Expression of Common Concurrency Patterns in Join Java - 2004 International Conference on Parallel and Distributed Processing Techniques and Applications. Nevada, United States of America.*

G Stewart Itzstein

1

Introduction

Small opportunities are often the beginning of great enterprises.

(Demosthenes).

Table of Contents

1.1 Introduction

This chapter introduces the content of the research reported in this thesis. The chapter is divided into three sections. The first section gives a motivation for the work. An overview of the research is given and indicates possible deficiencies in the expression of concurrency. The second section describes the structure of the thesis. Finally, the last section describes the research contributions of the thesis.

1.2 Motivation

Given the long history of object-oriented and concurrent programming (Dahl and Dijkstra 1972; Hoare 1985; Milner 1989) it might be expected that all the important issues have been thoroughly investigated. In fact, the concurrency implementations in modern mainstream languages are still quite primitive. At the same time, concurrency has become increasingly more critical to modern software and operating systems. This is reflected at all levels of computer systems architecture from hardware multi-threading (Intel 2004) to multi-threaded programming languages where concurrency is more closely integrated into the language syntax. With this increased provision of concurrency, it would be expected that modern languages such as C++ and Java would have incorporated a high-level set of language elements to better express concurrency. However, this is not the case. For example, C++ uses operating system level threads and a semaphore library that is not integrated into the language. In Java even though concurrency appears to be more closely integrated, its expression of synchronization is still quite low-level, being loosely based on monitors (Hoare 1974; Buhr, Fortier et al. 1995). Hansen states that the monitor implementation of Java (its claimed "high-level" expression of concurrency) is not even a true reflection of his original vision of monitors (Hansen 1999). He argued that Java synchronization should be used by default rather than left up to the programmer. Another author, Holub also made the following observation regarding Java.

> *"The Java programming language's threading model is possibly the weakest part of the language. It's entirely inadequate for programs of realistic complexity and isn't in the least bit Object Oriented."(Holub 2000).*

Process calculi on the other hand are designed to model concurrency rather than implement it (Milner 1989). These process calculi have a strong research heritage on the expression and formal meaning of concurrency. The deficiencies identified by Holub and Hansen within mainstream object-oriented languages might be overcome by using ideas from process calculi research.

Concurrency has been implemented in many languages. The first purpose built concurrent programming language was Simula (Dahl and Nygaard 1966). Simula also became the first language that introduced some of the ideas of object-oriented programming. Simula used a crude concurrency mechanism of co-routines that helped simulate concurrency. Later languages such as Ada (Mapping and Team 1994) improved upon this simple mechanism.

Strangely, however, C++ supplied less support for concurrency instead leaving implementations to the outside environment[1].

The authors of Simula incorporated a primitive expression of concurrency via co-routines because they recognized that the world that was to be simulated was itself concurrent. Since then there has been continuing interest in supporting parallelism in a number of different programming paradigms including that of object-oriented. Yet "modern languages" like Java support concurrency and synchronization using technology that is nearly 30 years old. These low-level primitives such as semaphores (Dijkstra 1968) and monitors (Hoare 1974) even though very flexible, are not easy to scale up to large applications nor are they sympathetic to the object-oriented paradigm. This has been recognized in the research community with a number of higher level concurrency implementations being proposed such as JCSP (Welch 1999), JavaTrevini (Colby, Jagadeesan et al. 1998) and JSR-166 (Lea 2002). Yet libraries do not entirely solve the problem as they don't closely integrate with the existing language semantics encouraging poor programming practice that the semantics was supposed to prevent.

Whilst imperative non object-oriented languages concurrency requirements could be satisfied using the low-level concurrency structures, object-oriented languages extensively use the structure of the program to represent information about the modelled domain. Programmers that use object-oriented languages make use of objects within their programs to model the problem they are solving. Consequently, if the language does not support concurrency within the object-oriented framework it will present a serious impediment to the design of quality programs. This is more critical now as object-oriented programming have become one of the most popular programming paradigms for system development (Andrews 1998). Languages such as C++ (Stroustrup 1983) and later Java (Gosling and McGilton 1996) have the attention of a large proportion of mainstream programmers at the current time (JobNet 2004). For instance in 1998 just two years after Java was first proposed the language was being used in 40% of IT companies (Andrews 1998). Java has also been used as a teaching language in 44% of universities (Hardgrave and Douglas 1998). Java has found wide scale acceptance in a number of real world application domains. For example Java's ability to act as an embedded language for small devices has been leveraged by Nokia for an API for mobile phones (Nokia 2003).

[1] For example PThreads (IEEE 1992) would be regarded the most popular thread package for thread programming in C and C++.

To see how easy it is to get into trouble using the low-level concurrency constructs in an object-oriented language like Java, a simple example is now presented relating to the use of a synchronized block. For example, consider the code in Figure 1 below.

The monitor lock reference in Java can be changed half way through a synchronized section of code. This problem is fundamentally caused by the fact that all non-base types in Java are represented by references rather than the object identity itself. Usage of the lock is via a variable reference that can be arbitrarily changed. This means that the user may change what the reference type is pointing to and thus replace the monitor at will. Code in Java can potentially look safe however, when executing is completely unprotected from multiple threads accessing it. Any successful language extension should solve this problem by hiding the lock from the user. In this way, the user cannot accidentally replace the lock at runtime. This problem is only one of the more direct examples. In other situations, problems can be extremely hard to locate and cause infrequently problems at runtime.

Object-oriented languages like Java (described in more detail from page 22) contain methods that change or report the state of an object or class of objects. These methods act as the message passing mechanisms between the objects in the program space. This design lends itself to encapsulation of data and restricts data access between objects. According to the object-oriented paradigm no data should be shared between objects other than those provided by the message passing methods. In Java's concurrency capabilities however, there exists no explicit message passing mechanism for thread communications. Threads communicate via shared

```
public class Breaksync {
   Object lockingObject = new Object();

   public void broken() {
      synchronized(lockingObject) {

            //various code

            lockingObject = new Object();
            //from this point code is no longer protected
            //code here that modifies supposed
            //protected data could corrupt the data structure
      }
   }
}
```

Figure 1. Erroneous Use of the Synchronized Keyword

variables with the synchronized keywords providing protection of the data.

In this thesis, message passing in Java between threads via a novel method of compound method signatures is introduced. That is when all methods from a compound method signature are called the associated body of the compound method is started and the parameters can be passed between fragments. The mechanism is introduced at the language level rather than a library. Intra-process communication is achieved via multiple method signatures sharing a method body. The actual communication is achieved using parameters and return types. If multiple compound methods share fragments, dynamic message passing channels are created. By implementing the synchronization in this way a thread message passing mechanism with very low cognitive overhead to the programmer is achieved in almost exactly the same way as message passing between objects. This innovation also demonstrates how a formal method can co-exist with the object-oriented paradigm.

1.3 Structure of Thesis

The rest of the thesis is organized as follows. In Chapter 2, the literature is reviewed. In Chapter 3, existing semantics of Java are reviewed and then the syntax and semantics of Join Java is introduced. Problems in the current semantics of Java are also covered. In Chapter 4, the implementation of the syntax and semantics of Join Java is described. The focus of this chapter is the technical challenges that have been overcome in the implementation. In particular, Chapter 4 describes the pattern matcher that is at the heart of the runtime component of Join Java. Chapter 5 examines how design patterns in concurrency and synchronization can be expressed in Join Java. Chapter 6 applies Join Java to classical concurrency applications. It is shown how Join Java can express most of these problems in a simple compact form. Chapter 7 benchmarks the performance of Join Java and shows that further optimizations are possible to overcome the remaining bottlenecks in the performance. The impact of boxing and unboxing in Java is examined for its effect on the performance of the language extension. Chapter 8 provides conclusions and suggests further research.

1.4 Research Contribution

The Join Java language introduces several features that do not have clear analogues in the original Join calculus formulation and other Join calculus-based systems. Imperative object-oriented versions of the Join calculus such as Join Java have much in common with the "joint action" approach taken in Disco (Kurki-Suonio and Jarvinen 1989; Jarvinen and Kurki-Suonio 1991). This language is fully message-based while most other implementations tend to be predominantly state-based. This thesis makes three main contributions to the state of the art in concurrent object-oriented languages. These contributions are;

1. Provision of high-level synchronization constructs into a mainstream language. These high-level constructs allow programmers to better model parallel behaviour in their programs without low-level synchronization mechanisms.

2. Provision of a safer intra-thread message passing mechanism via a guard-style method signature implementation. This high-level mechanism allows the dynamic creation of channels at runtime.

3. Improvement of thread integration in Java through a new return type that effectively adds asynchronous methods to Java. A side benefit of this is to allow parameters to be passed to threads at the time of their creation.

This dissertation explores how an expression of concurrency borrowed from process algebra can be incorporated into a mainstream object-oriented programming language. The extension has the aim of increasing the level of abstraction in which concurrent programs can be written.

2

Literature Review

Men who love wisdom should acquaint themselves with a great many particulars.

(Heraclitus)

Table of Contents

2.1 Introduction

In this chapter, the literature relevant to the expression of concurrency is reviewed. It is intended to mention all the key researchers, define the key concepts, and identify open questions in the literature that the work of this thesis aims to fill.

The survey is organized into five sections. In section one, concurrency as a general concept is covered. How the expression of concurrency has progressively become more abstract is reviewed. Some of the more popular abstractions associated with concurrency are presented. Process calculi are examined in section two for inspiration in deriving newer higher-level concurrency constructs for mainstream programming languages. This section also gives a detailed introduction to the Join calculus. In section three, an overview of the object-oriented paradigm with its motivations and history is presented. In section four, the options for implementing concurrency in object-oriented languages are examined. A number of categorizing attributes of concurrent object-oriented languages are also identified. Finally, a number of examples of real object-oriented languages that claim to support concurrency are then categorized using the identified attributes.

2.2 Abstractions of Concurrency

In this section, the fundamentals of the representation of the concepts of concurrency are examined. In the first part of this section, abstraction is defined. A definition from (Kafura 1998) is adopted and applied to the domain of concurrency. In the second part, the basic principles that form the foundation for any concurrent system are examined. The third part builds on these foundations by looking at higher-level concurrency abstractions such as monitors, semaphores and shared variables. It presents a common hierarchy that has emerged from the literature that illustrates the levels of abstraction. It also notes how results from design patterns literature present an even higher level of concurrency abstraction.

2.2.1 Abstraction

This thesis has adopted Kafura's definition of abstraction

> *Abstraction is a design technique that focuses on the essential aspects of an entity and ignores or conceals less important or non-essential aspects. (Kafura 1998)*

The key concept in abstraction is information hiding. Information hiding implies a layer at which some specific information is not visible and a layer below which that information is visible. This implies a strong relationship between abstraction and layering.

Kafura also sets out four general properties that make a good abstraction. Kafura firstly says that abstraction is concerned with attributes and behaviour. The attributes are the characteristics of the abstracted entity. Behaviours are the operations that the abstracted entity normally performs. The four properties that operate on the abstraction are;

1. The first property of good abstraction is that the abstraction is *well named*. That is if the abstraction has a good name the users of the abstraction will have an implicit understanding of it. An example of this is a variable, where the name, for example *counter,* implies the function as an abstraction.

2. The second property is that an abstraction be *coherent*. An abstraction is coherent when the attributes and behaviours that comprise the abstraction are related and expected given the domain in which they operate. For example, the attributes of a counter are to store an integer and the behaviours of a counter are to increment and decrement the integer.

3. The third property of an abstraction is that it be *minimal*. The abstraction should provide the least behaviours and attributes needed. For example, the behaviour increment by two is not minimal for a counter. A counter which has a colour attribute is an example of an abstraction that has more attributes than it needs.

4. The fourth property is that it is *complete*. This means that the abstraction should have enough behaviours and attributes to model the entity that is being abstracted. The counter behaviours presented previously are not complete because you need to be able to reinitialize to some reference value typically zero.

Kafura's general properties could also be used to evaluate the various abstractions of concurrency. To supply a good abstraction of concurrency there is a need that the concurrency mechanism to be *well named*. For example, mutex could be considered a poorly named abstraction although the full expansion *mutual exclusion* would be more appropriate and considered a better abstraction by the first property. Co-routines are well named as the name implies a similarity to subroutines but parallel in nature. *Coherence* implies that the attributes and behaviours of say, semaphores (Dijkstra 1968) which have two behaviours **up** and **down** are related and expected based upon the name. The third property that it is *minimal* can be seen in monitors where the abstraction only describes an entity that has behaviours **wait** and **notify**. The final property *completeness* implies that the monitors are complete enough to represent synchronization.

2.2.2 The Hierarchy of Abstractions of Concurrency

It has been noted that concurrency relies on three primitive abstractions (Schneider and Andrews 1986) *atomicity*, *basic communication*, and *synchronization*. These primitive abstractions are the lowest level abstraction of concurrency. For a concurrent system to be usable, it must implement these primitive abstractions in some form. Consequently, these abstractions are considered to be the foundation abstractions for concurrency.

All concurrent systems are composed of multiple actions that are active at the same time communicating and synchronizing with each other. The most fundamental abstraction of concurrency that is of interest to computer scientists is the *atomic action* (Lomet 1977). Every concurrent system needs some form of *atomicity*. This is the point in which an action is indivisible. Historically semaphores (Dijkstra 1968) were proposed before atomic actions; however it is well established that semaphores are built on the idea of atomic actions. These

abstractions of concurrency fit neatly into the Kafura properties presented earlier. That is they are *well named coherent*, *minimal* and *complete*. The abstraction of *atomicity* is *well named* as it implies indivisible action. The abstraction is *coherent* as its behaviour is what would be expected. The abstraction is certainly *minimal* and it is *complete*.

For concurrency to be useful there needs to be some form of *communication* between various parallel actions in the system. If communications were not supported in a concurrent system the only result from concurrent processes would be side effects such as printing information to the screen. The most basic form of *communications* is a variable in a shared address space (Schneider and Andrews 1983). Communication using such a shared variable can only occur correctly if the read and write operations on the variable are shown to be atomic. The abstraction is *well named*. It shows *coherence* with its behaviours being related to reading and writing from a shared variable. The abstraction is *minimal* in that it only is concerned with reading and writing to and from a shared variable and again is *minimal* as the behaviours and attributes could not be reduced any further. The abstraction is *complete* as all the behaviours necessary for the abstraction are present.

Synchronization is about the ordering of atomic events. *Synchronization* can thus also be defined as a mechanism for controlling access to a shared resource (Bloom 1979). For *basic communication* to be reliable, the abstract concept of *atomicity* is also necessary. *Synchronization* is how mutual exclusion of access is guaranteed to a shared variable in the previous example. Mutual exclusion means that only one parallel process can access a shared variable at one time, thus eliminating the problem of processes reading data structures that are currently being altered. *Synchronization* requires *atomicity* to work and assists *communications* to work reliably. Again, this abstraction obeys Kafura's basic properties of being *well named*, and *coherent* in that the behaviours of synchronization say that given a specific order of calls the result will always be correct. It is also *minimal* as the abstraction restricts itself to representing ordering of operations (the behaviour). The abstraction is *complete* as it deals completely with ordering of atomic actions.

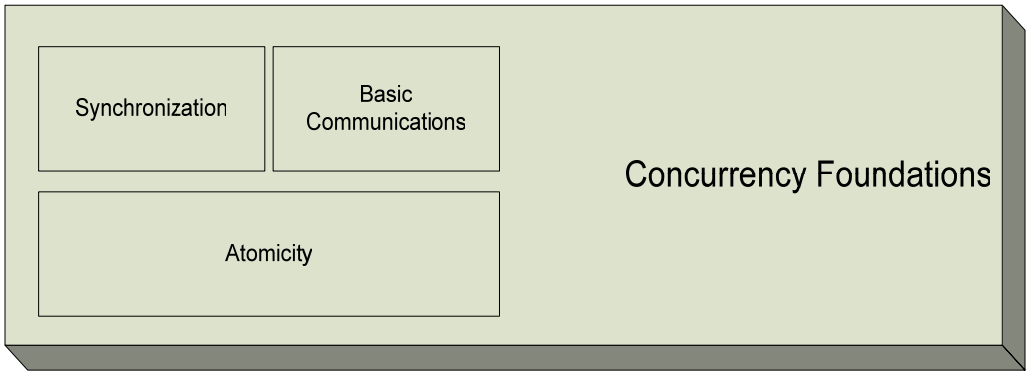

Figure 2. Foundation of Concurrency Abstraction

This layer of abstraction is illustrated in Figure 2 where it is represented as the foundation for a hierarchy of abstraction. This hierarchy will be built on; showing how each level of abstraction requires the lower levels of abstraction to function. Each higher-level abstraction gives the users easier to understand and utilize mechanisms for achieving concurrency and synchronization whilst hiding the lower-level details.

A number of concurrency abstractions have become popular in programming languages. These abstractions are somewhat higher in abstraction than that of *atomicity*, *synchronization*, and *basic communications*. However, if examined closely it could be seen that they make use of the lower-level abstractions to function. For example, *semaphores* make use of *atomic calls* to undertake check and set operations. Without *atomicity* of calls, *semaphores* would not work correctly. Similarly, *monitors* express a higher-level abstraction of concurrency. *Monitors* make use of *synchronization atomicity* and *communications* to function. Consequently, monitors and semaphores are considered a higher level of abstraction than that of *atomicity*, *synchronization*, and *basic communications*. One could look at *monitors* as an abstraction that hides the locks and queues associated with entry and exit to protected sections of code. The locks act as synchronization mechanisms and queues control the ordering of threads accessing protected segments of code.

It is clear that semaphores and monitors can be expressed in terms of each other's implementations; consequently, they are expressed side-by-side in the hierarchy. These mechanisms can be considered primitive abstractions in the abstraction hierarchy. Figure 3 shows yet another layer being added called intermediate-level abstractions. This level adds yet more abstractions such as buffered channels. These layers in turn reflect this increase in expressability via hiding of lower-level abstractions.

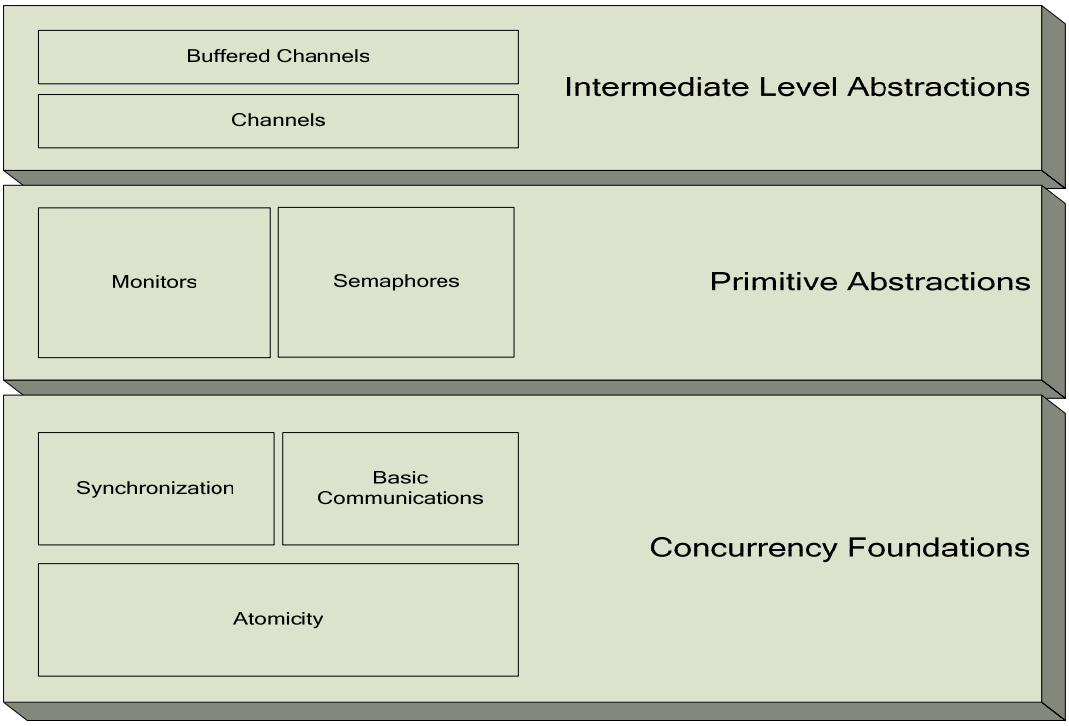

Figure 3. Extended Concurrency Abstraction Hierarchy

From the applications development point of view concurrency problems can also be partitioned into design patterns. That is most concurrency problems can be implemented using one or more generic design patterns proposed by a number of authors (Lea 1998; Schmidt 2000). Programmers make use of these "design patterns" to solve their specific application requirement. However, these patterns tend to be expressed in a high-level way that leads to difficulties if the language they are using only provides low-level abstractions. This is the case with most mainstream concurrent languages. Consequently, one of the most difficult aspects of programming using patterns is the translation from the general implementation supplied by the pattern designer to the specifics of the language and the problem. This difficulty could be reduced if there was a higher-level abstraction that better matched the generalized way design patterns are expressed. This missing layer between intermediate-level abstractions and design patterns can be seen in Figure 4. In this thesis, it is suggested that an extension to the Java language based on the Join calculus is one possible higher-level layer that more closely matches a number of design patterns.

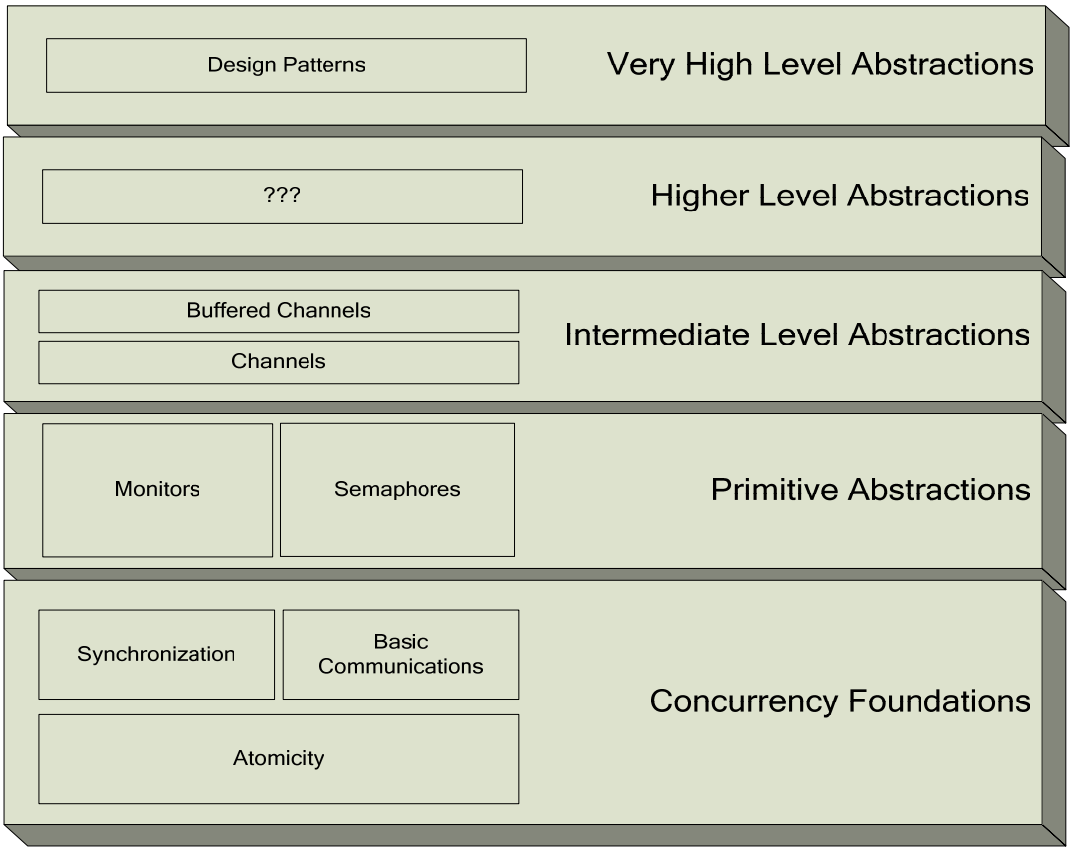

Figure 4. Complete Concurrency Abstraction Hierarchy

2.3 Process Calculi and the Join Calculus

In this section, we firstly introduce some popular process algebras giving some background on their design motivations. The Join Calculus (Fournet, Gonthier et al. 1996) is introduced in more detail and has its major terminology introduced. Finally, Join is contrasted with popular process calculi showing that it has features that are more suitable for implementation into the object-oriented paradigm.

2.3.1 Introduction

The term "process calculi" defines a group of formal description languages that are designed to describe concurrent communicating systems (Fidge 1994). These description languages (also known as process algebra's) have been defined as

> *An algebraic approach to the study of concurrent processes. Its tools are algebraically languages for the specification of processes and the formulation of statements about them, together with calculi for the verification of these statements(Glabbeek 1986).*

In this thesis three process calculi are mentioned; They are calculus of communicating systems CCS (Milner 1980), communicating sequential processes CSP (Brookes, Hoare et al. 1984; Hoare 1985) and finally the algebra of communicating processes (Bergstra and Klop. 1985) ACP. The CSP process algebra was initially inspired by Dijkstra's guarded command language (Dijkstra and Scholten 1990). CCS is designed in a similar handshake communication style to that of CSP. Whilst CSP is directed more at a theoretical basis and tends to be a more specification based language, CCS is more operationally oriented (Baeten and Verhoef 1995). ACP, a more recent work, differs from CSP and CCS in that it is more algebraic in its approach to modelling concurrency. These process algebras' embody the intermediate to higher-level abstractions based on our definition from the previous section. They are frameworks for modelling the behaviour of interacting processes rather than programming them. For a full comparative introduction of the differences between CSP and CSS consult Fidge (1994). Virtually all process algebras have the aim of modelling interaction of concurrent systems with syntax and semantic architectures varying between them. This research is more interested in the models of concurrency that are below the level of the process calculi. That is the syntax and semantics of concurrency and synchronization that are being implemented rather than modelling of behaviour. Consequently, an in-depth discussion of process algebras is beyond the scope of this work. However, the point that should be made is that CCS, CSP and ACP have

implied synchronization rather than explicit synchronization. This contrasts with the Join calculus presented in the next section.

2.3.2 Join Calculus

In this section, a detailed outline of the Join calculus is given. This section starts by giving a brief overview of the calculus and some of the terms that will be used for the remainder of the discussion.

The Join calculus can be thought of as both a name passing polyadic calculus[2] and a core language for concurrent and distributed programming (Maranget, Fessant et al. 1998). The calculi's operational semantics can be specified as a reflexive chemical abstract machine (CHAM) (Berry and Boudol 1992) (Maranget, Fessant et al. 1998). Using this semantic the state of the system is represented as a "chemical soup" that consists of active definitions and running processes (Maranget, Fessant et al. 1998). Potential reactions are defined by a set of reduction rules when a reaction occurs reactants are removed from the soup and replaced with the resultant definitions. The syntax and semantics of the Join calculus is defined in the next section.

2.3.3 Join Syntax

The Join calculus (Fournet and Gonthier 1996) contains both processes and expressions called Join patterns. Processes are asynchronous constructs that produce no result. Expressions are synchronous and produce a result. Processes communicate by sending messages through communication channels. These communication channels can be described using Join patterns. A Join pattern definition has two or more left hand terms and a right hand side. The expression will not reduce until there are calls to all left hand terms. A body itself may contain another expression that may include a parallel composition of two or more other expressions.

The syntax of the Funnel (Odersky 2000) language is used in preference to the syntax originally defined by (Fournet and Gonthier 1996). This syntax is closer to that of the target domain object-oriented languages, in our case Java, as its syntactic components are consistent with other parts of the Java language specification. The Funnel variant (Odersky 2000) syntax of the

```
Expression          E => def D; E | x (y1….yN) | E & E
Definition          D => L = E | D, D
Left Hand Side      L => X(Y1…YN) | L & L
```

Figure 5. Funnel Variant of Join Calculus Syntax

[2] Polyadic Calculi have processes and channels which have identifiers

Join calculus is presented in Figure 5.

For example given the Join calculus expression presented in Figure 6 it can be seen that there are two expressions on the left hand side of **X(Y1)** and **X(Y2)** and a right hand side with a parallel composition of **xa(y1)** and **xb(y1)**. This means that when expressions **X1(Y1)** and **X2(Y2)** are available the body of the definition is evaluated, in this case the parallel composition of **xa(y1)** and **xb(y1)** expressions.

2.3.4 Join Semantics

In the previous section, it has been shown how the body of the definition is only evaluated when all expressions in the left hand side are available. This behaviour can be modelled using the CHAM (chemical abstract machine) semantics proposed by (Berry and Boudol 1992). The chemical abstract machine semantics are modelled on the idea of a soup of waiting chemicals that can react to generate new chemicals while removing old chemicals from the soup. As an example of the semantics of the chemical abstract machine a brief example is presented. Given a machine with one possible reduction **def A&B = C&D** it is assumed that the soup already contains the left hand side expressions **A & R & F & S & C**. Consequently, the state of the CHAM is as shown in Figure 7. Further if the expression **B** is added to the soup, the soup will contain all the terms on the left hand side of the reduction rule. The terms will then react and generate all the terms on the right hand side of the reduction rule removing the terms on the left hand side of the reduction rule from the soup. This is shown in Figure 8 where the addition of term **B** creates new terms (via the reduction rule) **D** and **C** in the soup and the terms **A** and **B** are removed.

Join's dynamic nature emerges when there is the possibility for multiple reductions to occur depending on the combination of Join expressions appearing. If one were to augment the definition above with an additional definition **def A&Z = X** it can be seen that on a call to **A** depending on whether **Z** or **B** were previously called the choice of which definition to evaluate would occur. If **Z** is waiting (called previously) in the CHAM then the second definition would evaluated, **A** and **Z** would be removed from the CHAM and replaced by **X**. Alternatively if **B**

```
def  X1(Y1) & X2(Y1) =
     xa(y1) & xb(y1) ;
```

Figure 6. Example Join Calculus Expression

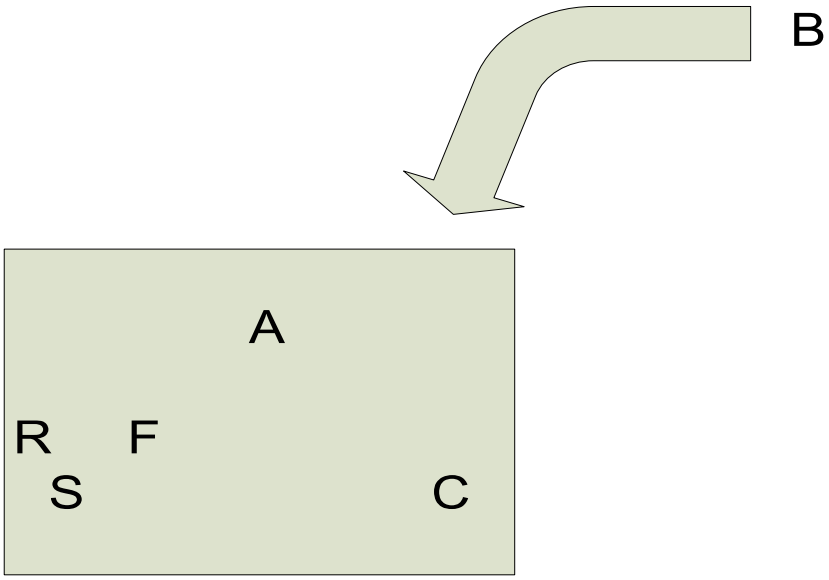

Figure 7. Addition of B to CHAM

was waiting in the CHAM then the first definition would be evaluated removing **A** and **B** from the pool, replacing them with **C** and **D**. One alternative situation that can occur is; what happens if **Z** and **B** are both waiting in the CHAM and **A** is then called. In this situation, a non-deterministic reduction is possible in that either the first definition or second definition could equally be reduced but not both. The calculus does not specify what would occur in this case.

2.3.5 Synchronous Names

Combining the Join semantics from the previous section with blocking semantics a mechanism of flow control can be achieved. These blocking fragments of the definition are called synchronous names. With this addition to the semantics of Join calculus, message passing can

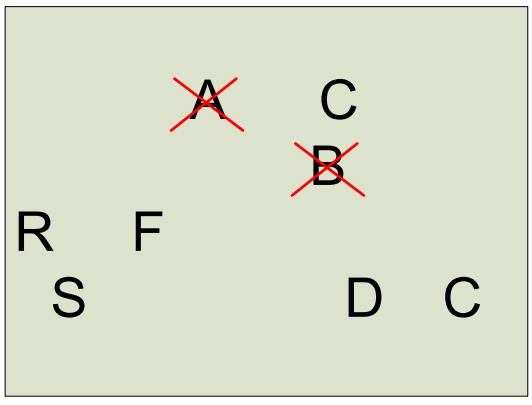

Figure 8. Operational Semantics of the Join Calculus

be supported in a language level implementation. This idea is sympathetic to the object-oriented paradigm.

2.3.6 Join Calculus vs. Other Calculi

The Join calculus gives an ideal alternative to the current popular concurrency semantics in programming languages via its locality and reflection. Locality allows us to limit reductions to only one site in the calculus rather than all sites. The CHAM semantics in the previous section omits the locality as the reductions are entirely completed within a global pool. If this were to be implemented in the object-oriented paradigm, a CHAM would need to be created in each object in order to give locality. Reflection on the other hand allows us to define reductions dynamically when the evaluation occurs. In this thesis, we will implement locality via localization of CHAM's to the object level. We will not implement reflection other than that implicitly offered by that of reflection libraries of the base language.

Fessant also points out an advantage of the Join calculus over other calculi.

> *Processes communicate by sending messages on channels or port names. Messages carried by channels are made of zero or more values and channels are values themselves. By contrast, with other process calculi (such as the pi-calculus and its derived programming language Pict), channels, and the processes that listen on them are defined by a single language construct. This feature allows considering (and implementing) channels as functions, when channels are used as functions. (Fessant and Conchon 1998)*

Odersky (Odersky, Zenger et al. 1999) argues that Join Calculus better expresses the idea that most modern applications are reactive in their interfaces and concurrent in their implementation. The pi-calculus (Milner, Parrow et al. 1992) and the fusion calculus (Parrow and Victor 1998) whilst expressing interaction between entities well do not represent sequential expressions sufficiently. It has been pointed out that complicated type systems are needed to address these problems.

2.3.7 Summary

In this section, the Join calculus was identified as a promising candidate for the expression of concurrency that would be suitable for incorporating into a mainstream object-oriented language. It is shown that Join's high-level abstraction is based on the conjunction of atomic actions and expression of synchronization explicitly in a way that is not likely to breach the modularity required by object-oriented paradigms. This strongly motivates the choice of Join as the basis for a high-level concurrency extension.

2.4 The Object Oriented Paradigm

In this section, what defines the object-oriented paradigm is examined. The review particularly emphasizes aspects that interact with concurrency. Some of the basic concepts of the object-oriented paradigm are described. It is shown how the object-orientated paradigm has information hiding as one of its central concepts. The requirements for a programming language to be object-oriented are looked at and what advantages this gives are examined. It is shown how object-oriented languages provide a framework in which the only information an object shares with another object is the information passed via a message passing mechanism such as method calls. The idea of message passing in the language and how encapsulation requires us to limit the access of member variables in classes is scrutinized. Finally, the family tree of object-oriented languages is examined. For a comprehensive survey of object-oriented languages consult (Sutherland 1999).

Object-oriented languages were first proposed by (Dahl and Nygaard 1966) as a way of representing real world concepts. The primary object-oriented language structure is the class, that is an augmented record that also contains the operations that work on the data contained within that record. Instances of classes can be created which are known as objects. Each object's code operates on the data within the instance object. In this way, the code is associated with specific data rather than as is the case with non object-oriented languages, where the code is generic and data is passed to it.

Before object-oriented languages, most software analysis and design focused on process. Process designs were either data centric[3] or control centric[4]. This data centric or control centric approach tended to become problematic and was limited to small to medium sized programming projects. These limitations and non-scalability issues became known as the *software crisis* (Dijkstra 1972; Booch 1987). At this point software engineering was more an art with the differences between good projects and bad projects being mainly due to the skill of the team. In an attempt to improve this situation a methodology of design that would avoid the scalability issues of the process model techniques was pursued. Developers needed to bring reuse and closer representation of the real world to software engineering. This closely matched the idea of the object-oriented paradigm that was proposed by the Simula language (Dahl and

[3] Data centric applications can be described as programs that are designed to serve data from large data storage systems for example databases.

[4] Control Centric applications can be described as command style programs designed to interact with other systems or the outside world.

Nygaard 1966). The object-oriented paradigm offered abstraction via "class hierarchies" that allowed control of cohesion between components in the system. This was a necessary requirement for the increased complexity of projects being developed.

An important concept of software engineering is the idea of high cohesion and low coupling (Stevens, Myers et al. 1982). A highly cohesive module will require outside modules as little as possible. A loosely coupled module will be as constrained as possible from communication with other modules in the system. The two principles of high cohesion and low coupling work together to decrease the complexity of software hence reducing potentially hard to localise errors. In the object-oriented paradigm, encapsulation is the method in which these two principles are achieved. Encapsulation hides attributes from other objects (modules) via modifiers. Modifiers allow the programmer to set the visibility of methods and attributes. In object-oriented languages, it is customary to restrict visibility of internal methods and attributes so that they cannot be seen outside the object in question. In this way, these issues of high coupling are avoided such as having various objects access any attribute or method they wish. If arbitrary access to the objects state were allowed the application would be highly coupled and hence unnecessarily complex. This reduction in accessibility leads to a requirement of a more formal mechanism of object communication known as message passing. This starkly contrasts with the majority of concurrency abstraction implementations that are highly coupled and have low cohesion. Concurrency abstractions tend to be non-localized where concurrency calls can be made from anywhere in the code to anywhere else in the code. An example of this is semaphores where up and down calls can be made anywhere. Monitors address some of these deficiencies by associating locks and queues to scopes. However, monitors have even been shown to be a problem (Ramnath and Dathan 2003) where they increase the coupling of more complicated patterns. Within an object-oriented program, no object should be able to directly manipulate another object's attributes and hence state. The way that one object seeks to change the state of another's object is via mutator methods. These method's explicit purpose is to allow another object to request a change of an object's internal state. If an object needs the contents of another object, it is done via an accessor method. In this way, the object can protect itself against being placed in states that it should not be. This reduction in coupling reduces the complexity of the interaction between different modules of the system hence increasing the scalability of software development. Any concurrent mechanism implemented in an object-oriented language should support this implicitly.

2.4.1 Advantages of the Object Oriented Paradigms

The advantages of the object-oriented paradigm can thus be summarized as:

1. Data Abstraction: This concept allows complicated representations to be modelled in the language whilst still maintaining readability. The object-oriented paradigm achieves this via inheritance of classes. In this way, generic concepts are represented high in the inheritance structure whilst concepts that are more specific are added progressively as one descends through the inheritance tree.

2. Encapsulation: Bundles related data and methods into neat packages that then provide a form of modularity[5]. Encapsulation also provides support of information hiding via interfaces (definition of the methods within a class). In this way, the user of the class does not need to know the implementation details of the class they are using. In fact, the implementation can be arbitrarily changed at any time with the user of the class not even knowing.

3. Polymorphism: Quite often, it is a good idea to group similar objects of different classes so they can be treated in the same way. For example in a drawing program, representative classes may have for each type of drawing a widget such as circles or spheres. It would be useful if one could just refer to all these widgets as shapes and write methods that use shapes. In this way, one can add new types of widgets without having to write new methods for each. This helps support code reuse hence leading to more scalable engineering of software systems.

4. Reusability: By using the encapsulation facilities of object-oriented languages, an amount of reusability is acquired. The encapsulation of data structures with the methods, attributes with well-defined interfaces allows us to hide the implementation details from the user. This, together with high cohesion and low coupling, leads to code that can be reused in different projects. Furthermore, making use of the facility of polymorphism, code reuse is attained. That is, rather than writing a different method for each type of class; instead, a carefully designed inheritance structure will provide a generic class with generic methods. The methods are written to accommodate the generic class signatures any subclassed object can then be passed as the generic type.

[5] It is arguable that the Object Oriented paradigm is flawed in that the inheritance structure of most modern applications libraries is defeating encapsulation and high cohesion.

2.4.2 History of Object Oriented Languages

With so many object-oriented languages available, it is useful to see how the major languages relate to each other. Object-oriented languages have developed a long way since the introduction of the Simula language. A number of external sources have influenced the development of languages culminating in recent languages such as Java (Gosling and McGilton 1996), C++ (Stroustrup 1983) and Sather (Lim and Stolcke 1991). Figure 9 shows a tree illustrating the development of object-oriented languages. The tree shows the main languages in relation to the non-object-oriented languages they draw inspiration from and how they relate to each other. This tree is by no means complete[6] but shows some of the highlights and how they relate to each other. The Simula languages are the founding languages for object-oriented concepts. Simula can also be regarded as the first language to introduce true concurrent

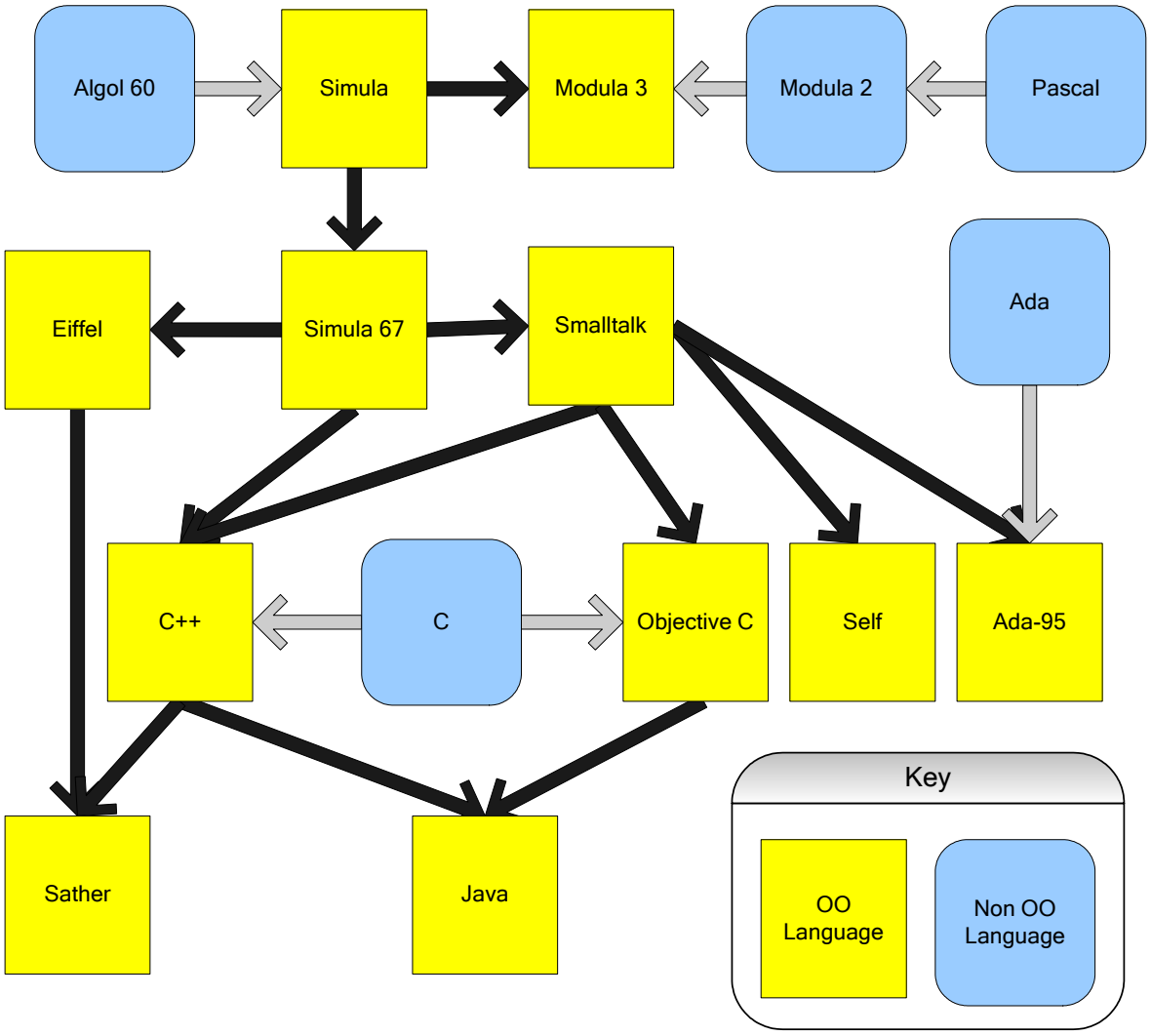

Figure 9. A partial family tree of object-oriented languages

[6] (Sutherland 1999) reports that at the time of his publication there were over 170 known Object Oriented languages

behaviour[7]. One might regard Simula as the watershed that spawned most of the modern languages that have gained wide spread acceptance today. It is interesting to reflect that most object-oriented languages since the original Simula language have had some form of concurrency. However, the most popular object-oriented language (C++) omitted concurrency support and only supplied the most low-level of primitives for synchronization. Java on the other hand supplies a concurrent class *Thread* that supports a single asynchronous method. However, even this class is still an addition to the language and not truly integrated into the language syntax.

2.4.3 Summary

It can be seen from the preceding sections object-oriented languages leverage the idea of information hiding, encapsulation, inheritance, and message passing to achieve the aim of building a paradigm that will support the continually increasing size of programming projects. The language design is elegant in data structure composition with control structures being subjugated to the data rather than data being supplied to the control structures. Communication is also more elegant with the concept that communication should only be achieved through accessor/mutator methods[8]. This section showed that languages based on the object-oriented paradigm have as a central design aim to restrict inter-object communication as much as possible. For consistency, communication should only be allowed via a tightly controlled message-passing construct. This ideal can be intuitively extended to threading to say that one should restrict inter-thread communication as much as possible. That inter-thread communication should only allow threads to communicate through safe message passing constructs rather than arbitrarily accessing shared variables. Unfortunately, concurrency is generally not well implemented in object-oriented languages. Concurrency tends to be minimally integrated with a lot of dependence on external libraries.

In the next section, the concepts of the fusion of object-oriented languages and concurrency will be introduced. An overview of some object-oriented languages is presented along with how concurrent object-oriented languages can be categorized.

---

[7] Simula offered via co-routines a simple form of concurrency. Co-routines are similar to sub-routines except they save control state between calls.

[8] Generally these intra-object communications are called message passing.

2.5 Concurrent Object Oriented Languages

In this section, existing approaches to the expression of concurrency in a selection of object-orientated languages will be surveyed. The selection includes some research languages in addition to the three mainstream languages C++, Java, and Ada95. This section starts by suggesting a number of categorizing attributes in which concurrent object-oriented languages can be differentiated. The categorization of actor-based languages vs. non-actor based languages is examined first. Following this, a categorization of how closely concurrency is integrated into the language is shown. The final categorization is that of mainstream vs non-mainstream languages. A number of concurrent object-oriented languages that have already been proposed are reviewed. The definition of what is a mainstream object-oriented programming language and how these mainstream languages have handled the issues relating to the expression of concurrency are reviewed. Finally, selected concurrent object-oriented languages are reviewed with emphasis on the classification of concurrency. The outcome of this survey is the realization that for most mainstream languages there has been little attempt to encapsulate threads or integrate concurrency and as a result, many object-oriented specific problems that researchers have identified are amplified.

Object-oriented designs do not necessarily make concurrent programming easier. A poorly designed concurrent object-oriented program can easily obscure the behaviour of threads running in parallel. Unlike processes in operating systems, which are protected by memory management software (other than those explicitly given all privileges), Java for example uses a type system to protect users from executing unsafe operations. However, the Java type system does not protect the user from concurrent access to shared variables. For example, programming a thread pool using only monitors can be a non-trivial task. The programmer needs to pass references to a job dispatcher. This job dispatcher via some central registry of workers finds which threads are idle and signals a thread that a job is waiting. The worker thread then collects the job and runs it returning the outcome via some shared variable. Implementations of this pattern can be quite difficult with shared data being vulnerable to corruption due to double updates if the author is not careful to fully protect shared data.

There are a great number of concurrent object-oriented languages available today. Languages such as Java (Steele, Gosling et al. 1995), ADA (Wellings, Johnson et al. 2000) and C++ with PThreads (Stroustrup 1983) are popular languages. These languages have varying levels of support for concurrency/interaction, ranging from difficult low-level external library support such as C++ to low-level usable concurrency as is the case with ADA and Java. In the

following sections, these languages will be covered categorizing them by their particular approaches to concurrency implementation.

A major feature that defines the object-oriented paradigm is the idea of class inheritance. Class inheritance is where a class takes the attributes/methods from another class in addition to the ones it provides. This allows a generalized class to be written and then classes that are more specific are written to express domains that are more specific. With respect to concurrency, inheritance causes problems in expression. The inheritance anomaly is where redefinitions of methods are required to maintain the integrity of concurrent objects (Matsuoka, Wakita et al. 1990). When method names and signatures are known, and method bodies are abstract, the signature of a parent class does not usually convey any information regarding its synchronization behaviour. This behaviour is usually defined within the body (Fournet, Laneve et al. 2000). With this restriction in most object-oriented languages an inheritance anomaly can exist if a subclass method cannot be added without modification in some form of the superclass methods. This of course is the reverse of the design methodology in object-oriented languages where subclasses modify the behaviour of the superclass by overriding the superclass methods. If the superclass has to be rewritten for the subclass, the design paradigm breaks down. Much of the research into concurrent object-oriented languages has been carried out in an attempt to reduce the impact of the inheritance anomaly. (Crnogorac, Rao et al. 1998) shows that none of these languages can actually fully solve the problem, as there appears to be a fundamental incompatibility between concurrency and object-oriented paradigms. For this reason, most research aims to minimize the problem rather than eliminate it. This thesis does not aim to solve the inheritance anomaly, but tries to contribute to reducing the likelihood of occurrence. The thesis highlights how low-level synchronization expressions that are poorly integrated into an object-oriented language require them to be placed in the body of methods. This approach of placing constraints within the body is an enabling mechanism for the creation of inheritance anomalies (Pons 2002). Thus, it can be concluded that there is a definite argument to investigate how high-level concurrency can be added to mainstream languages. With this alternative approach, it may also be possible to reduce the number of inheritance anomalies via moving constraint information to the inheritable part of the class rather than the method body.

In the following sections, we will introduce three categorizations of concurrent object-oriented languages. These categorizations are actor categorization (Section 2.5.1), implementation categorization (Section 2.5.2) and mainstream categorization (Section 2.5.3). A number of languages are then classified using the categorizations in section 2.5.4. In section 2.5.5 we

show that a gap exists in the field of concurrent object-oriented languages that Join Java intends to fill.

2.5.1 Actor versus Non-Actor Based Concurrent Languages

The first categorization shows how parallelism and synchronization are integrated within the language.

1. Actor based languages (Agha 1986). In these languages, every object is also a thread and objects. A true active object language will make the thread active the moment the object is created. The first actor based language was Plasma (Hewitt 1976) originally named Planner 73 (a non-object-oriented language based on Lisp). Following Plasma there have been a number of implementations of Active Object Languages. Plasma II (Salles 1984) evolved later with support for parallel interpreters. Still later Alog added extensions for logic(Salles 1984). Alog evolved into Smart (Salles 1989) which added support for distributed interpreters. Figure 10 gives a brief description of the evolution of active object languages from Plasma up to languages that are more recent.

2. Non-actor based languages. In these languages, threads are separate from the objects. In this way, several threads may be active within a single object at the same time. This mechanism allows objects that do not need to be active to have low overhead. In many implementations, the language has specialized libraries that allow code to run in parallel. An example of this category is Java (Gosling and McGilton 1996). In Java, if one wishes to produce a thread, a call to the library is made or the asynchronous method of the thread class is overridden. Initially when the first object-oriented language was proposed, it supported concurrency via co-routines that were conceptually quite similar to that of sub-routines from older style imperative languages. The idea of separating the objects from the threads can be considered a passive object approach to concurrent object-oriented languages. The disadvantage of this approach is that the programmer has to explicitly create threads when they deem necessary. There is a separation of threads and objects in the language to the programmer. This leads to cognitive overheads as the programmer tries to define where threading is required and where it is not required then designing a mechanism for communication between the threads.

Actor based languages whilst not necessarily being object-oriented share a great number of similarities with that of the object-oriented paradigm. The term active object language will be used when referring to actor-based languages that are within the object-oriented paradigm. In the following sections, the thesis concentrates on the active object variety of actor-based languages. However, this section will cover the most important languages in the entire set of actor-based languages for completeness.

2.5.2 Integrating Concurrency into Object Oriented Languages

In concurrent object-oriented languages, there are several ways of integrating concurrency into the language. These approaches have a direct effect on the ease of use and safety of

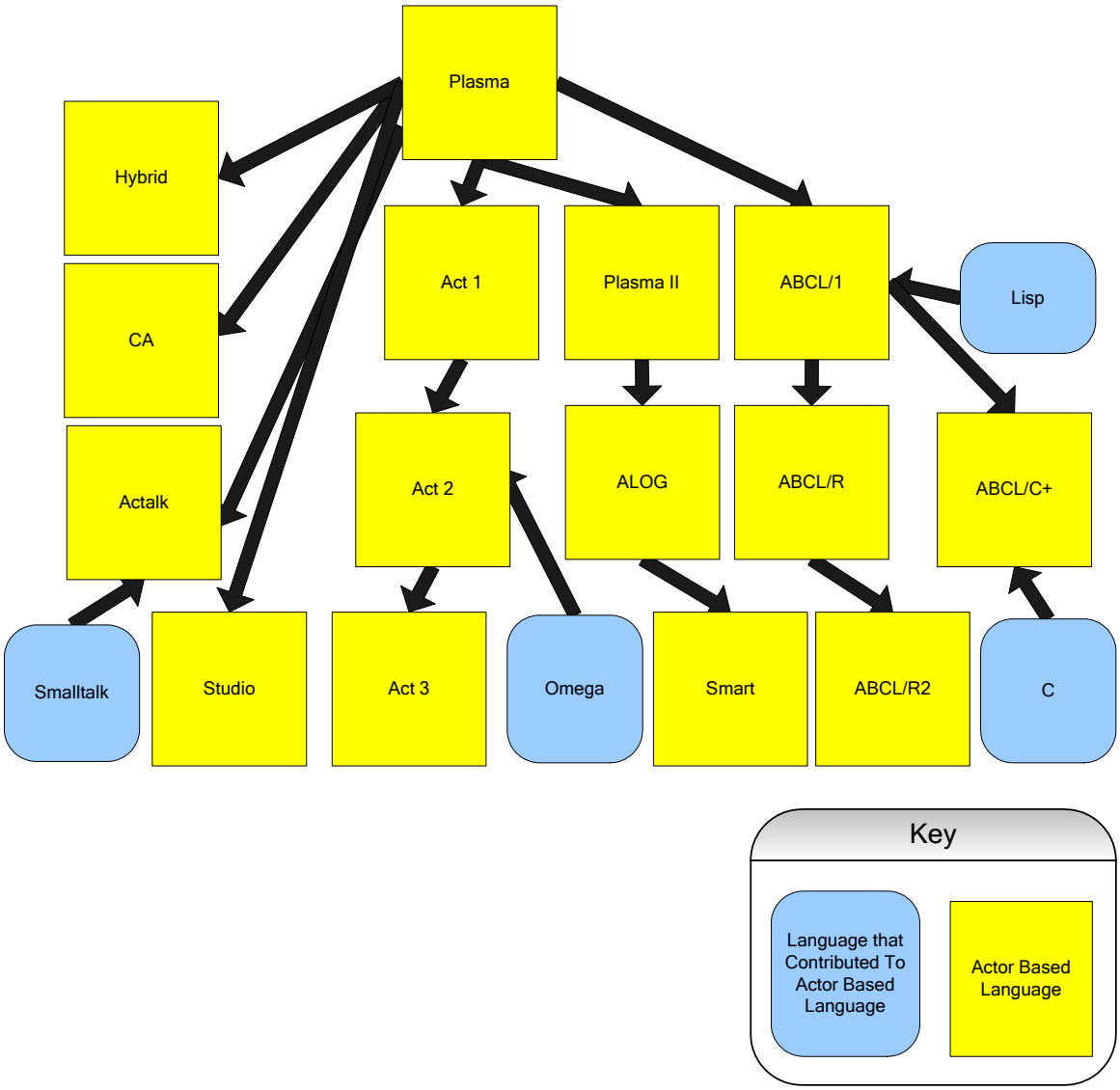

Figure 10. Evolution of Actor Based Languages

concurrency within the language.

1. Library approaches. Concurrency and synchronization features are embedded via a predefined set of libraries. This approach is taken by JCSP (Welch 1999) and CSP for Java (Hilderink, Broenink et al. 1997) both which retrofit CSP semantics to Java via library calls. These types of extensions are normally quicker to develop however lack the advantages of language level integration. For example, errors in the use of libraries cannot be detected by the compiler leading to long debug cycles for application development.

2. Superset. This is where an existing language is expanding with syntax and semantics to support concurrency. For example, Timber (Black, Carlsson et al. 2001) adds concurrency and imperative object-oriented semantics to Haskell (Jones, Hughes et al. 1998; Jones 2003). These implementations tend to gain wider acceptance as they leverage existing user knowledge to allow for quicker understanding of concepts. However, they are usually slower than the base languages as they add features that may break the optimizations in the base language's compiler.

3. Design a completely new language. Create a new language with semantics of concurrency embedded into the language. An example of this language is Funnel (Odersky 2000) where the language was specially designed to illustrate a small set of features including concurrency. This is the ideal solution, as no compromises need to be made in implementation. These implementations have the disadvantage that there are no programmers who will initially understand the language. It is then harder to get the language to be accepted as a mainstream language.

2.5.3 Defining Mainstream Languages

For the purposes of this thesis, mainstream languages are defined as any language that is used by a reasonable proportion of non-academic users. That is, those languages that are used by a significant proportion of industry programmers. For example, C++, Ada and Java would all be regarded as being mainstreams language due to their widespread use.

2.5.4 Implementing Concurrent Object Oriented Languages and Extensions

Concurrency has been long recognized as being necessary in any modern programming language. Right back at Simula-67 and Algol-68 threads and simplistic synchronization primitives have been made available in languages. Later languages like OCCAM (Inmos 1984) have implemented small improvements such as monitors (Hoare 1974) and CSP style constructs (Hoare 1980). Java has made concurrent programming using threads widely available to mainstream programmers. However, this situation has just re-emphasized what many-experienced concurrent programmers already knew, that concurrent programming is inherently difficult to get right. In the following paragraphs, some of the more well known approaches to implementing concurrency in object-oriented languages will be covered.

MuC++ (Buhr 1992) a language which extended C++, was designed to give programmers access to the underlying system data structures. It implemented low-level operations such as real-time monitors, timeouts dynamic-priority scheduling and basic priority inheritance. It uses a translator and runtime kernel that supports concurrency using a shared memory model. The language modifies the C++ syntax but does not modify the compiler. A pre-processor converts the program into a normal C++ language file. A significant disadvantage of this is when a compilation error occurs; the compiler will give error messages in terms of the translated code not the extension language.

Another language, ACT++(Kafura, Mukherji et al. 1993) is one of a large number of languages based upon the idea of actors (Agha 1986). This language is implemented using a library rather than a language extension. Again this language is based on the C++ language. However, as it is a library extension it does not modify the syntax.

Synchronization rings (Holmes 1995) rather than being a specific language was proposed as a language independent model of synchronization. More specifically synchronization rings are a library of behaviours that support the composition of concurrent behaviours. This language independent model has been implemented in two languages C++ and Java. The language independence whilst being flexible suffers from not being closely integrated with the language.

The Tao (Mitchell 1995) model (and later implementing languages) primarily aimed to overcome some of the problems involved with the inheritance anomaly. The prototype language is implemented as a language extension with a pre-processor that generates C++ code

after passing through a translator. The language achieves its aim via the novel approach to inheritance anomaly avoidance. However, the prototype suffers from the same problem that other translation languages have. That is errors are given in terms of the translated language rather than the extension language.

A number of implementations of CSP (Reppy 1992) have been created for object-oriented languages (Demaine 1997; Hilderink, Broenink et al. 1997,Welch 1999; Bakkers, Hilderink et al. 1999). Most of these CSP style extensions are in the form of libraries that are added to the language. Demaine's (1997)'s work on Java communicating sequential processes is a good example. In his extension, events are regarded as first class objects that are treated as normal objects in Java. The extension (a Java package) uses a number of primitives for sending and receiving these events. Demaine also thought it important to supply an *alt* or non-deterministic choice primitive operator as well. The language extension is based upon the work done by (Reppy 1992) which in turn was based upon work by Hoare (1985). Similarly (Hilderink, Broenink et al. 1997)'s implementation uses a library to introduce CSP style semantics into Java. Whilst these approaches are usable, it would be imagined that the CSP primitives should really be implemented at the language level.

The first actor based language was proposed by (Hewitt 1976). The language (originally named Planner 73) was based on Lisp syntax but supplied in addition to the existing multitude of parenthesis even more parenthesis and brackets. This made the language somewhat hard to interpret when reading code. From Figure 10 above it can be seen that from Plasma a large number of languages were spawned in several branches including the Act series of languages. Act 1 (Lieberman 1981), Act 2 (Theriault 1983) adopted features from the Omega language (Attardi and Simi 1981), The Plasma designer returned with Act 3 (Hewitt 1985) which was an domain specific variant of the Act series of languages. Plasma 2 introduced a parallel interpreter (Salles 1984). Salles then further extended Plasma 2 with further support for logic in ALOG (Salles 1984). This was then further extended into Smart which contained support for distributed computing on virtual machines (Salles 1989).

A set of descendants of the Plasma language was the ABC series of languages ABCL/1 (Yonezawa 1990), ABCL/R (Watanabe and al 1988) ABC language paradigm with reflection and finally ABC/R2 (Yonezawa 1992) which merged features from Hybrid into the Common Lisp semantics. One problem with these languages is that they mostly used Lisp as the host language. These languages proved to be unpopular outside academia. To remedy this a

mainstream language variant of ABC was created called ABCL/c+ (Doi and al 1988)) which used C as the host language. In a number of ways, this extension to C was synonymous to the C++ extension to C.

A number of independent languages were created to explore facets of actor-based languages in relation to object orientation. A short selection of these are presented with their more memorable characteristics. Concurrent Aggregates (Chien 1990) was aimed at being a more restrictive and pure implementation of the object-oriented paradigm with actors (an active object language). It also added features such as continuations and first class behaviours to the language. At about the same time some implementers favoured Smalltalk (Goldberg and Robson 1983) created an extension called Actalk (Briot 1989). A number of other independent active object languages were created that explored active object ideas such as Lucy (Kahn and al 1990) and Studio (Hadjadji 1994). These languages, whilst being excellent academic explorations of the facilities of actor based languages tended to be of little use in mainstream programming as they were frequently hosted in either purpose built narrow domain languages or more commonly in the Lisp language.

A small number of actor-based languages were created that merged with the mainstream languages C or C++. These languages merged the ideas of the original Plasma language with the syntax and semantics of the host language. For example, Hybrid (Nierstrasz 1987) abandoned the Lisp syntax style in favour of C++ syntax style with static typing. Another C++ language Act++ (Kafura 1989) was designed to be a full concurrent extension of C++ based on the active object paradigm. Synchronous C++ (Petitpierre 1998) was proposed more recently. This extension contained a real time kernel that allowed each object to contain a thread of control that could be interrupted at will by other active objects. The ideas of Synchronous C++ were ported to Java with Synchronous Java (Petitpierre 2000) two years later. Both languages were implemented via concurrency libraries.

Triveni is a process-algebra based design methodology that combines threads and events in the context of object-oriented programming (Colby, Jagadeesan et al. 1998). The Triveni system is not really a language but a framework that can be added to any language via a library. One of these extensions is the JavaTriveni (Colby, Jagadeesan et al. 1998) library. The framework supplies two classes of combinators: standard combinators from process algebras such as parallel composition, and pre-emptive combinators (Berry 1993). In Triveni, communication is done via events that are in fact named communication channels. The framework also has a

specification based testing facility that allows testing for safety properties. However, Triveni's major contribution is the pre-emptive combinators that allow another process to abort or suspend processes at runtime and in combination with asynchronous communication that earlier frameworks did not possess. While Triveni is an interesting approach however, it does have some disadvantages. Firstly, the framework is entirely implemented as libraries and consequently does not support the extension at the language level. This means that the users have to call library functions to make use of expressions that are more naturally expressed as language primitives. For example, the parallel operation that executes a number of jobs concurrently is expressed as an array of new jobs passed to the constructor of the parallel object. This is non-intuitive, as one would expect such a low-level operation to be expressed as a syntactic operation rather than a library operation. Of course, this could be done with Triveni however, that would lose some of the language independence the designers were seeking when designing the framework. The second disadvantage is the differences between it and the platform on which it sits (for example Java). To be ultimately usable the extension should fit as closely as possible with the paradigm of the platform on which it is placed. Again, this was a conscious design choice as the developers of Triveni were aiming for a platform independent framework.

In ADA (Guerby 1996), tasks are integrated into the language via class style syntax. Synchronization is via a rendezvous mechanism. Rendezvous is a mechanism where one thread sends a message to another thread via two primitives, accept and entry. An ADA rendezvous occurs when control reaches an accept statement in one task and another task executes the corresponding entry call statement. One of the negative features of ADA rendezvous is that it is prone to dead lock (Levine 1998). This is generally attributable to the lack of locality, which makes it hard to identify dead lock prone code.

The JoCaml language was proposed by (Maranget, Fessant et al. 1998; Fournet, Laneve et al. 2000) it is an extension of the objective-Caml language with Join calculus semantics. The authors claim that the language includes high-level communications and synchronization channels, failure detection and automatic memory management. JoCaml was a second language implementation of the Join calculus authors (Join language (Fournet and Maranget 1998)) JoCaml is implemented via a concurrency library added to the Caml Language. It also modifies the bytecode set of the Caml language in order to make the extension work more easily. This language extension is a more interesting extension but is implemented in an unfamiliar language

to mainstream programmers. It would thus be necessary to translate the ideas to a production system in order to gain more ready acceptance of the ideas.

Bertrand Meyer extended the Eiffel (Meyer 1988) language to support concurrency and synchronization. His extension, concurrent Eiffel (Meyer 1997) was a straight forward syntactic addition to the language. He achieved concurrency via adding a keyword **separate** that indicates that the object has its own thread of execution. These threads of executions communicate via method invocations in each thread's own methods. If the method is not ready to reply it will block the caller until the reply is ready. Despite the fact that the base language Eiffel is popular in the academic area and some restricted commercial domains(such as telecommunications), Concurrent Eiffel cannot be regarded as a mainstream language.

Concurrent ML (Reppy 1992) is an extension based upon SML/NJ (George, MacQueen et al. 2000). The language supports dynamic thread creation and typed messages passing channels. Threads are implicitly first class in the language, which means that synchronisation abstractions can be dynamically created. The language also provides a number of other features such as stream I/O and low-level I/O being integrated into the language itself. This language however, could not be considered mainstream.

C++ in reality provides no support for threading or synchronization. Threading is achieved via operating system specific libraries that do not integrate closely with other language features. Synchronization is achieved through either data structure sharing or pipes. The danger of this approach is the libraries are not necessarily platform independent, which means that programs have to be modified for every target platform. Secondly, other libraries and language components that threaded programs may use may themselves not be thread safe. For instance a member of the class may be declared as static which means only one instance exists for all classes. In normal cases, this may be acceptable however; if the data is accessed and modified in a multi-threaded context, it is possible to corrupt the contents of the field with unpredictable results. This is a problem, as generally the application programmer has no access to the source code. This means they have to either risk possible data corruption or rewrite their own versions of the libraries. Generally, the solution to this problem is to create a wrapper class that protects the class using some form of locking mechanism (such as the OS semaphores of UNIX based C and C++ languages). This of course increases the complexity of the application being programmed hence reducing maintainability.

Java supports multithreading via the use of a special thread class that in essence implements an asynchronous method. To make a threaded program the user subclasses the thread class and then overrides' the run method. Synchronization is achieved via a monitor embedded into every object in the program. Programmers make use of these monitors to protect data structures whilst other threads are modifying the same data. A deeper description of the semantics of Java concurrency can be found in Chapter 3. It is easy to, firstly circumvent the monitor lock by accident or, secondly to lose track of the locality of locks which leads to the possibility of deadlock. It has also been noted that non-method level locking leads to greater chances of inheritance anomalies (Pons 2002)

In this section, a selection of the most relevant object-oriented languages and extensions were covered. It can be seen that actor based and more specifically active object languages are an interesting solution to the issues involved in concurrency and object-oriented languages. It could also be presumed that the active object paradigm would be more expensive computationally than that of non-active objects presented due to the number of active threads being implicitly larger. It can be seen how academic languages whilst having superior concurrency mechanisms are not widely used. Counter to this is the mainstream group of languages, which are more conservative in their approach to concurrency, yet are widely accepted. Consequently, through these languages it can be seen that there is an apparent gap between the promise of the academic languages and the reality of the mainstream languages.

2.5.5 Categorization of Concurrent Object Oriented Languages

In this section, we summarize the languages we have examined using the categorization that were identified earlier. Table 1 shows the method in which concurrency is added to the language. Table 2 shows whether the language is considered mainstream or non-mainstream. Finally, Table 3 shows whether the language is an actor based language or non-actor language.

Language	Library Based Implementation	Retrofitted	New Language
ABCL/1		✓	
ABCL/C+		✓	
ABCL/R		✓	
ABCL/R 2		✓	
Act 1			✓
Act 2		✓	
Act 3		✓	
ACT++	✓		
Ada			✓
Alog		✓	
C++	✓		
Concurrent Eiffel		✓	
Concurrent ML		✓	
Hybrid		✓	
Java	✓		
JoCaml	✓		
MuC++		✓	
OCCAM			✓
Plasma 2		✓	
Plasma/Planner 73			✓
Simula			✓
Smart		✓	
Studio			✓
Tao(model)	✓		
SyncRings (model)		✓	
CSP (Models)	✓		
Triveni	✓		

Table 1. Summary of Concurrency Integration of Languages

In Table 1, it can be seen that the majority of languages are retrofitted languages. A significant proportion are library based in that their concurrency is not explicitly implemented syntactically in the language. A number of assumptions were made in this table regarding whether particular language features are library based, new language or retrofitted. For example, Java whilst being a new language with some concurrency features integrated into the language (synchronized keyword) is in fact a library implementation. We only consider the language to be non-library based if it can create and synchronize threads without resorting to system libraries.

Language	Mainstream	Non-Mainstream
ABCL/1		✓
ABCL/C+		✓
ABCL/R		✓
ABCL/R 2		✓
Act 1		✓
Act 2		✓
Act 3		✓
ACT++		✓
Ada	✓	
Alog		✓
C++	✓	
Concurrent Eiffel		✓
Concurrent ML		✓
Hybrid		✓
Java	✓	
JoCaml		✓
MuC++		✓
OCCAM		✓
Plasma 2		✓
Plasma/Planner 73		✓
Simula		✓
Smart		✓
Studio		✓
Tao(model)		✓
SyncRings (model)		✓
CSP (Models)		✓
Triveni		✓

Table 2. Summary of Mainstream vs. Non-Mainstream Languages

In Table 2 it can be seen that the majority of languages are non-mainstream. This makes sense as industry users are generally unwilling to move from the languages they are familiar with unless necessary. If we were to examine the mainstream languages mentioned here, we can see that they tend to be iterations of existing languages. For example, C evolved to C++ and Java adopted a large proportion of the syntax from C++.

Language	Actor Based	Non-Actor Based
ABCL/1	✓	
ABCL/C+	✓	
ABCL/R	✓	
ABCL/R 2	✓	
Act 1	✓	
Act 2	✓	
Act 3	✓	
ACT++	✓	
Ada		✓
Alog	✓	
C++		✓
CA	✓	
Concurrent Eiffel		✓
Concurrent ML		✓
Hybrid	✓	
Java		✓
JoCaml		✓
MuC++		✓
OCCAM		✓
Plasma 2	✓	
Plasma/Planner 73	✓	
Simula		✓
Smart	✓	
Studio	✓	
Tao(model)		✓
SyncRings (model)		✓
CSP (Models)		✓
Triveni	✓	

Table 3. Summary of Actor vs. Non Actor Based Languages

Finally, Table 3 gives a summary of actor vs. non-actor languages. In the next section, we extend the categorizations presented so far to cover that of intra-process communication.

2.5.6 Intra-Process Communications

Another important characteristic of concurrent systems is that of intra-process communications. Whilst a number of different mechanisms exist, this thesis will categorize these intra-process communications mechanisms into three types using (Andrews 2000) criteria.

1. Message passing; Communication is via channels formed between parallel threads. Channels do not necessarily have to protect themselves, as they are generally purpose built between particular threads. An example of this is or CSP (Hoare 1980) and OCCAM (Inmos 1984).

2. Shared Resource; Threads communicate via data that is accessible by all. This "shared data" can be read and altered concurrently. Shared data has to be protected from corruption therefore; some form of multiple access protection is required. Protection is usually via a locking mechanism such as monitors (Hoare 1974) or semaphores (Dijkstra 1968).

3. Coordination; Communication is via a shared space (tuple space) that dissociates interaction of the components from the components themselves. The most well known example of coordination languages is that of Linda (Gelernter 1985; Carriero and Gelernter 1989). Whilst this mechanism is similar to the shared resource category it is possible to separate languages based on coordination as it is a higher-level mechanism for accessing the shared resources.

Language	Channel	Shared Resource
C++		✓
Concurrent Eiffel	✓	
Concurrent ML	✓	
Java		✓
JoCaml	✓	
MuC++		✓
OCCAM	✓	
SyncRings (model)	✓	
CSP (Models)	✓	
Triveni	✓	

Table 4. Summary of Communication Mechanisms for Concurrent Languages

In the previous sections, a number of examples of message passing and shared resource implementations were covered. These examples are summarized in Table 4. If we were to compare this table to the languages in Table 2 we would find that mainstream languages tend to be shared resource languages. Coordination is omitted in this table, as it tends to be adapted to a language rather than a feature of any language on its own. For example, Linda has been adapted to Java in the JavaSpaces (Sun 1998) API.

2.6 Related Work

Initially when the Join calculus was announced there was one main design that was based on this work called Functional Nets(Odersky 2000). This used some of the concepts of the Join calculus and Petri-nets(Peterson 1981) and applied them to functional languages. This implementation later evolved into the language Funnel(Odersky 2000). The Join Java language itself used the ideas of the Join calculus but adopted the syntax of Funnel. The work of Nick Benton, Polyphonic C#[9] at Cambridge followed the same development ideas as that of Join Java. This language has a number of similarities with that of Join Java. Benton has acknowledged Join Java was a simultaneous discovery in his latest paper.

> *The work that is most closely related to Polyphonic C# is that on Join Java [Itzstein and Kearney 2001, 2002]. Join Java, which was initially designed at about the same time as Polyphonic C#, takes almost exactly the same approach to integrating join calculus in a modern object-oriented language.(Benton 2004)*

2.6.1 Similarities between Join Java and Polyphonic C#

Join Java introduces two main concepts to Java, that of Join methods and asynchronous methods. Polyphonic C# also adds two main concepts Chords and asynchronous methods. Effectively with some minor syntactic differences, the Chord is an identical structure to that of a Join method. Asynchronous methods in Join Java have a **signal** return type. Asynchronous methods in Polyphonic C# have an **async** return type.

2.6.2 Differences between Join Java and Polyphonic C#

Whilst these two languages have a number of commonalities, there are also a number of significant differences between them. The major differences between Join Java and Polyphonic C# are described below.

1. Each implementation makes use of a different base language. The Join Java extension is based on the older Java language. Polyphonic C# is based on the more recently released C# language.

2. Join Java has a richer semantic for modifying the behaviour of ambiguous reductions of Join methods. This allows the user to specify whether the matching behaviour is sequential or non-deterministic. This is specified via a modifier that is added to the

[9] Polyphonic C# has recently been merged with a new language called Cω.

class declaration. The user sets the modifier (**ordered**) on the class to specify whether the pattern matcher uses random selection or first declared priority to select the appropriate pattern in the event of two or more possible ambiguous reductions of Join methods. Polyphonic C# does not offer this facility. The Polyphonic C# implementation is a purely non-deterministic implementation. This would make a number of design patterns (see Chapter 5) more difficult to implement, as there is no priority constraint mechanism.

3. To avoid issues with the inheritance anomaly and keep the implementation as simple as possible Join Java restricts inheritance by making the class final. Join fragments, however, can be inherited either via interfaces or via extending classes that have abstract Join methods as abstract method signatures. On the other hand, the Polyphonic C# language attempts to allow inheritance but in response to the issues involved with the inheritance anomaly has a number of rules to try to reduce the impact. For example, in Polyphonic C# if any method of a chord is overridden all chords must be overridden. This is synonymous with the typical solution to the inheritance anomaly where the superclass synchronization mechanism must be re-declared in the subclass in order to ensure that inheritance anomalies do not develop.

4. In Join Java, when declaring Join methods the first fragment can be either asynchronous or synchronous but all further methods must be synchronous. In Polyphonic C#, only one method can be synchronous but it can be any method.

5. The research for Join Java explored various pattern matchers. The polyphonic C# research did not pursue this avenue.

6. Join Java does not need to restrict **ref**/**out** operators in the Join methods as the base Java language does not support passing of references to stack frame variables. This difference is more a function of the base language than the extension.

7. The research into the usefulness of Join Java covered a number of design patterns in an attempt to have a true evaluation of the usefulness of the extension. The Polyphonic C# research did not pursue this avenue instead pursuing a more formal approach to evaluation of the language.

2.6.3 Summary

Whilst the Polyphonic C# and Join Java languages are superficially quite similar there are a number of significant differences. These differences can be categorized into engineering differences and research directions. Firstly, the engineering differences between building a Join calculus extension to Java and C# are significant. The Java JVM is much more restrictive at the byte code level forcing the extension designer to be quite creative in the implementation of the compiler. For example the lack of boxing and unboxing operations. The .net framework on the other hand has a large number of additional language features such as boxing and unboxing which allows easier translation of the extension. Also Java does not allow pointer arithmetic which means that there is a level of indirection that can sometimes reduce performance. For example storage of waiting parameters can be complicated in the pattern matcher. Secondly, the research directions were quite different. The Join Java approach was to better support programmers trying to implement design patterns and higher-level concurrency semantics. Consequently a thorough coverage of design patterns was undertaken which yielded a number of issues that could only be solved using priority constraints (**ordered** modifier).

2.7 Conclusion

In this chapter, the idea of abstraction was initially studied. In the first section, a model of abstraction that suggests that by increasing the level of abstraction of concurrency one can decrease the difficulty in creating safe concurrent code was proposed. The Join calculus, the basis of our extension to Java, was suggested as a possibility in the next section. The Join calculus showed promise as a good model of synchronization for higher-level object-oriented languages. The next section briefly examined the paradigm of object-oriented languages taking special care to point out features that imply design decisions relevant to the addition of higher-level concurrency and synchronization. It was discovered that the paradigm is built upon the ideas of encapsulation and inheritance. Encapsulation is supported by locking state information of objects away from other objects insisting that they communicate via message passing. This implies that if concurrency is available in an object-oriented language it should follow the same design principle of protecting data and insisting on some form of safe message passing between active threads. The next section examined a number of concurrent object-oriented languages both academic and mainstream. The languages are categorized using three classifications, actor vs. non-actor, level of integration of concurrency, and mainstream vs. non-mainstream languages. The features of the languages were summarized in addition to the communications mechanisms they use. It was found that the academic languages promise many interesting features yet are rarely adopted. Alternatively, it was found that mainstream languages are generally conservative in their concurrency mechanisms. It was suggested that there is room for modifying existing mainstream languages at the syntax level to incorporate some of the higher-level concurrency semantics of process calculi. It can be concluded that there needs to be more work on adopting features from clean academic designs into object-oriented languages that have already been accepted. In the final section, we examined the related work Polyphonic C#

3

Join Java Syntax and Semantics

The most powerful designs are always the result of a continuous process of simplification and refinement.

(Kevin Mullet)

Table of Contents

3.1 Introduction

In this chapter, the concurrency syntax and semantics of the Join Java superset will be described. The chapter begins with an examination of the interesting anomalies that can occur when writing concurrent Java programs. These anomalies suggest avenues for improvement in the language's concurrency semantics. In the next part of the chapter, Java is examined in light of the Join calculus. The final part of the chapter gives a description of the proposed new concurrency extension that we call Join Java.

3.2 Java Language Semantics and its Deficiencies

In this section, the semantics of threading and locking in Java are examined. Portions of the following material are drawn from the Java language specification (Steele, Gosling et al. 1995).

3.2.1 Java Concurrency

There is no specific language syntax support for threads in Java. There is only low-level support for synchronization via monitor style locking. Instead, the language supplies a specialized thread class that either is subclassed (see Figure 11) or has code passed to it (see Figure 12). These threads may be implemented in a number of ways either via hardware support (true concurrency) or much more commonly via time slicing on a single processor. Each of these implementations can lead to differing evaluations of the same code based on the policies of time slicing or hardware implementations. The Java specification is extremely restrictive on the implementations of the threading model. This leads to JVM's that do not implement the entire specification in order to aid optimizations (Sun 1996; Sun 2002). The Java virtual machine does not specify the conditions under which threads may be pre-empted therefore Java application threads can be pre-empted at any time. In other words, threads can be suspended to allow another thread to continue executing its run method. If a thread is midway through modifying a shared data structure when it is pre-empted, there is the possibility of double update occurring. This is where data is corrupted by two threads simultaneously changing the value simultaneously. There are also optimizations in the JVM that lead to reordering of instructions (Sun 1996). This effectively means that it is impossible to predict the result of a multithreaded program in Java without additional language support. This support is via the *synchronized* keyword introduced in the next section.

In Java, there are two methods of creating asynchronous code. The first method illustrated in Figure 11 shows how by subclassing the class *Thread* an advantage can be gained via inheritance to get concurrent code. However, there are a few limitations to this approach. Firstly, no parameters are permitted to be passed to a thread specification other than via the constructor. A wrapper method usually is constructed that will accept parameters and sets shared variables. The thread then gets the original methods that get the same values. Secondly, the subclass cannot extend any other class due to Java's single inheritance restriction. These problems are mostly a direct result of the thread mechanism not being fully integrated into the language. This leads to awkward unstructured workarounds to achieve what the software engineer wants. The second method, shown in Figure 12 remedies some of these problems by

```
public class MyThreadVer1 extends Thread{

   public void run() {
      //some code to be run in parallel
   }

   public static void main(String[] argv) {
      MyThreadVer1 x = new MyThreadVer1();
      x.start();
   }
}
```

Figure 11. Using subclassing in standard Java to achieve multithreading

passing an object containing the multithreaded code. This solves the second problem of single inheritance but is still awkward for parameter passing to the threaded code. Alternative Java methods are presented in chapter 5, for example Figure 62 *t1* and *t2* methods.

Both methods do not really support parameter passing. Instead, they use either a shared variables technique as would be the case with Figure 11 or, as would be the case in Figure 12 the addition of get/set methods to pass parameters into the thread. The problem with this approach is that the shared variable is potentially unprotected if the code is not constructed carefully. The second approach, using set/get methods to set a variable in the thread is safer. However, when a thread is created, the programmer must make sure to set the variables before calling the start method for the thread. Again, there is possibility that if the code is not constructed carefully it will lead to undesirable consequences. An example of this type of solution is presented in Figure 13. There are a number of ways this code can be inadvertently made unsafe. Firstly, the *synchronized* keywords are necessary on both the set and get methods. As described in the following section *synchronized* will allow only one thread to access any synchronized section of code that is protected by the same monitor. If the programmer omits a *synchronized* on one of the methods this would mean that more than one thread can modify the

```
public class myThreadVer2 {
   public static void main(String[] argv) {
      Thread x = new Thread(new Code2Run());
      x.start();
   }
}
class Code2Run implements Runnable {
   public void run() {
      //some code to be run in parallel
   }
}
```

Figure 12. Using interfaces in standard Java to achieve multithreading

shared variable concurrently leading to the possibility of the program entering an expected state. Secondly, if the thread object does not protect the shared variable via access modifiers any external thread can access the data-structure directly bypassing the synchronization mechanism altogether. It should be noted that this implementation is simpler than in practice as generally it is considered unwise to synchronize on the threads monitor. Finally, the complexity of these mechanisms become even more unwieldy if one considers communications between the thread creator and the thread after the thread is created (see chapter 5). Further description of these issues can be found in (Lea 1998) and the JSR-166 (Lea 2002) specification.

In this section, a brief overview of the threading mechanism in Java was given. It was shown how there are two methods of achieving concurrency in the language. Some of the issues involved with passing arguments into threads were also described. In the next section, the synchronization construct in Java that allows the threads to pass information safely will be examined.

3.2.2 Synchronization

The mechanism for serializing access to data-structures is based on an honour system in which all accessor/mutator of the shared data structure must be synchronized. If one of these accessor/mutators' does not follow the synchronization mechanism then the entire data structure is open to corruption. This section will give two examples that show how weak Java is in relation to enforcing good concurrent programming at compile time. The first example is the straightforward omission of synchronization. The second example relates to changing of the

```
public class myThreadVer3 extends Thread{
   private SOMETYPE sharedVariable; //must be set
   public void run() {
      //get the shared variable via set get method
      //some code to be run in parallel
   }
   synchronized public void setParam(SOMEOTHERTYPE value) {
      sharedVariable.data = value;
   }
   synchronized public SOMEOTHERTYPE getParam() {
      return sharedVariable.data;
   }
   public static void main(String[] argv) {
      myThreadVer3 x = new myThreadVer3();
      x.start();
   }
}
```

Figure 13. Accessor Mutator Implementation

identity of the monitor inside the protected section of code.

The first example of the circumvention of synchronization in Java is the omission of synchronization. When creating threaded programs in Java any shared variables need to be protected either by synchronized blocks or synchronized methods. Figure 14 below gives an example of a variable *X* that is being modified in synchronized methods *safe1* and *safe2*. This implies that only one thread is permitted to access the method at any one time. However, a common error is to omit one or more synchronized modifiers in the code. This is shown in the example where *safe2* has no synchronized keyword. If this occurs, more than one thread can modify the shared variable leading to corrupted shared data. In the example, this means that one thread can access *safe1* and any number of threads can access *safe2*.

The second example of the circumvention of synchronization in Java is the modification of the monitor lock object reference. Figure 15 shows an example of how someone can either intentionally or unintentionally break the locking semantics of Java. In the example, a protected section of code is synchronized on the object *lockingObject.* Half way through the synchronized section of code the programmer changes the object that the reference *lockingObject* points to. From that point on the code is unprotected and any other thread may enter the code section as the lock has been replaced with the new object *newObject*. An analogy of this program is locking your house and having a burglar replace the entire door (including the lock) in order to open it. Consequently, it is important to remove access to the locking object itself from the user. The moral is that the shared variables must be packed with the lock that protects them. This is exactly what the synchronized modifier for a method does. The monitor is associated with the object, which holds the shared variables. In likelihood, these problems may not occur frequently in practice. However, it does illustrate some obvious

```
public class BreakSync {
   SomeObject X = new SomeObject();
   //…

   synchronized public void safe1() {
       if(X.someCondition)
          X.someMethod();
   }

   public void safe2() {
       if(X.someCondition)
          X.someOtherMethod();
   }
}
```

Figure 14. Omission of Synchronized Keyword

```
public class BreakSync {
   Object lockingObject = new Object();
   static int counter=0;

   public void broken() {
      synchronized(lockingObject) {

           //various code

           lockingObject = new Object();
           //from this point code is no longer protected
           //code here that modifies supposed
           //protected data could corrupt the data structure
      }
   }
}
```

Figure 15. Dangerous Modification of Monitor Object

deficiencies that would probably spawn more dangerous and subtle problems.

There are two possible solutions to this problem. Firstly, the language designer could ban the modification of a reference to a locking object in the section of code that it locks. This approach would complicate the compilation process, as the abstract syntax tree would need extra decorations in addition to checks for the code dealing with entry and exit and assignment of the locking object. Similarly, the programmer could also make the object *final* so that modification is flagged by the compiler. The second solution to this problem is to hide access to the locking object from the user. This is a further abstraction of the design ideal that Java already possesses in which it stops the programmer from directly modifying the monitor within an object. If the programmer is denied direct access to the object on which the locking is conducted, there is no chance of unfortunate situations like the previously presented one developing. However, one could argue that this denial of access to the locking object may restrict the usefulness of the synchronization mechanism. Consequently, if one was to deny access to the locking object the replacement mechanism needs to be very flexible. One way to do this is to abstract the mechanism from being a purely locking device to a communications medium. For example, a communications channel abstraction could be created that would mean the locking is implicitly done for the programmer.

3.2.3 Wait/Notify

The thread blocking semantics in Java are implemented via the wait/notify operations on the monitor. When a thread is waiting on a monitor it will release its locks and suspend itself until a corresponding notify on its monitor occurs. The problem with these low-level semantics is that they can be called from anywhere in the environment as long as the caller acquires the

monitor lock. This leads to a situation synonymous to that of the gotos (Dijkstra 1968) in old imperative languages. Because there is no locality for the wait and notify, these methods can be called from anywhere as long as the monitor lock is acquired. This leads to situations where encapsulation violations are generated as objects modify the state of other objects. This is a violation of one of the core principles of the object-oriented paradigm. Notify also is non-deterministic in that when it is called it will pick a single waiting thread and wake it. The thread that is woken is arbitrary and dependant on the platform implementation. To avoid the non-deterministic race condition it is suggested that the programmer set a condition and a wait in a while loop that tests the condition. Then all threads wake up via a notifyAll call and check their individual condition flags. They either then go back to sleep or continue on the basis of the condition. It would be advisable to try to avoid this low-level method of passing control between threads as it is non-localized and leads to highly coupled software. High coupling defeats the encapsulation ideal of the object-oriented paradigm. A possible solution to this problem is to create a special object that hides its lock behind a private interface. The user then calls public lock and unlock methods. This approach leads to awkward mechanisms as locks are now separated from the code. Because it is not making use of the monitors in the local object, rather using what is effectively a library this approach suffers from all the issues of a library approach to concurrency. Finally, as the lock is removed from the local code scope, automatic scoped locking and unlocking is no longer available, leading to potential dead locks. In this section, it has been shown that there are significant weaknesses in the Java concurrency semantic that could be improved.

3.3 Principles for Improving Java Concurrency Semantics

In this section, the thesis investigates the requirements for extending the concurrency in Java. The high-level requirements for the Join Java extension are examined. With these high-level requirements a novel approach to improving the synchronization and inter-thread communication of Java can be proposed.

3.3.1 High-Level Requirements

A primary motivating factor in this research is to make it easier, safer, and more natural for a programmer to write multithreaded code. In this research, a number of characteristics that contribute to achieving that aim are identified. The requirements are;

1. Faithfulness: The primary priority was to be faithful to the original language paradigm. It was seen in section 2.5 that the object-oriented concepts of Java such as encapsulation, inheritance, and intra-object message passing are critical to the paradigm. Consequently, any new features added to the language should support and/or complement these features as much as possible.

2. Increased robustness against programmer error: This is achieved by providing a mechanism that can be checked at compile time. Higher-level abstraction (see section 2.2) should also encourage designs that do not have low-level problems such as omission of locks (see section 3.2) on shared variables.

3. Performance: An important requirement is that the performance of the extension is not markedly less than that of a standard Java implementation running on the same particular hardware/software architecture.

4. Minimalist design: Make minimal changes to the target language and make the extensions as isolated as possible within the language. That is make as few strategic syntactic and semantic changes to the language as possible. By making minimal changes to the language, any person picking up the new extension for the first time will have an easier time understanding the extension. It has been seen in section 2.5.5 that mainstream programmers prefer languages based on existing mainstream languages. This would mean that programmers would be more likely to adopt the extension.

5. Backward compatibly: The extension should be able to accept a standard Java program and compile to standard byte code. Additionally a program written with the extension

should compile back to standard byte code. The advantage of this is that any program written in the extension language or the standard language and compiled with the extension compiler will run on any standard Java virtual machine. The disadvantage of this approach is that the extensions have to be translated into standard byte code. This also leads to some performance decrease, as you are not able to optimize the JVM for the extension.

6. Message passing mechanism that complements the method call technique of serialized applications: In this way, changes to the language are minimized whilst being faithful to the original languages paradigm. The locking paradigm popular in a large proportion of concurrent languages is counter intuitive to the language paradigm in which communications between entities (in most cases objects) should be as restricted as possible. Implicitly, locking mechanisms in a large proportion of industry languages can be considered to be global variables in which different threads of execution dip in and out as they like. More importantly, each thread of execution is expected to obey access guidelines, for example they should only use synchronized accessor/mutator methods to modify the global data structure. If the thread of execution misbehaves and accesses the data structure directly **all** synchronization is invalidated. In object-oriented paradigms, a core concept is that entities should tightly control the access to their state. Generally, in serialized applications this is done by method calls and access modifiers. In concurrent applications, this same ideal should hold with a communications channel being formed between two entities (threads) to send information.

7. True superset: A language extension should not interfere with the base language method of communication between threads. Consequently, any extension should be a separate syntactic structure to the original mechanism to avoid confusion for the programmer.

8. Locks for communications channels will be hidden from the programmer: Hiding the locking mechanism will reduce the possibility of locking mechanisms being circumvented and hence improve the safety of the code. This criteria is limited by the seventh criteria. Consequently, the language should be designed so that the modification of the lock is unavailable when using the extension method of synchronization. However, for backward compatibility reasons the access to the locking mechanism in Java has not been modified.

3.3.2 Extension Decisions in Intra-Process Communications

Using the presented high-level requirements, an existing language development library and existing concurrency semantic were chosen. By using technologies that already exist, the possibilities of flaws in the design of the extension are reduced and the complexity of the language design simplified.

One decision made was the selection of the high-level semantics for concurrency and, more specifically, communication to be used in the extension. There are number of interesting approaches already existing that fall into three categories relating to communication (Andrews 2000). The three categories are *Shared Variables*, *Message Passing,* and *Coordination*. By coordination, Andrews means, a mechanism based on shared tuple spaces. The author goes on to introduce three additional categories however; these categories are more related to parallelism of code or abstract models. With respect to language integrated communications constructs the categories both Ada (via protected types), and Java (via synchronized variables) use shared data variable models for communication whilst CSP (Hoare 1980) and Occam (Inmos 1984) (among others) use synchronous message passing. Although Java already supplies a message-passing mechanism via remote procedure call, (RPC) this mechanism is chiefly used as a distributed mechanism rather than a parallelised code mechanism. Secondly, the RPC mechanism is via libraries and one of the aims of this thesis was to integrate more closely the extension into the language and semantics. The third category of integration is coordination in which languages such as Linda (Gelernter 1985) and JavaSpaces (Sun 1998), a derivative of Linda, use a tuple space to coordinate the interaction between different language domains. In the extensions design, some of the concepts of the coordination languages are borrowed and then applied to message passing at the syntax level of the language. For example coordination mechanisms use tuples to dynamically choose communications channels. If the extension could make use of this dynamic nature an extremely flexible intra-process communication mechanisms could be introduced.

3.3.3 Concurrency Semantic Choice

Join calculus gives an excellent mapping to the requirements. The calculus was designed to have the ability to form communications channels when coupled with method signatures rather than the customary continuation passing style of other calculi. The design of the Join calculus also implies that the synchronization locks for the communication channel can be hidden inside the channel structure. This will increase the safety of the language structure avoiding the

possibility of the programmer changing the identity of the lock whilst the lock is being used. An additional advantage of the Join calculus is the dynamic channel formation via the variation of members of a reduction (see section 2.3.2). This matches with the coordination style channel structure required in section 3.3.2. Another requirement was that the design should make minimal changes to the existing language implicitly. This is also a requirement of the calculus as well. Consequently, whilst parameterised Join calls and synchronous method calls are added the extension did not allow multiple synchronous calls in the same method (see Section 3.4). Another requirement for integrating Join with Java is that the fundamental idea of explicit synchronization is preserved. This can be done via the extension specifying the channel formation rules explicitly in the syntax.

3.4 Join Java Language Semantics

Given the apparent weaknesses of the Java approach already presented it could be argued that some of the opportunities presented by formal semantics such as the Join calculus should be used to improve Java's concurrency semantics. In this section, such an extension is introduced. First, the changes made to the Java language are specified. Next, the syntax and semantics of the new language are informally examined. Finally, some of the semantic issues arising from the extension are investigated.

3.4.1 Changes to the Language

In this section, the syntax and semantics of the Join superset of Java are introduced.

Join Java makes three syntactic additions to Java.

1. Addition of Join methods for synchronization and channel construction.
2. Addition of a signal return type for creation of asynchronous (threaded) methods.
3. Addition of ordered modifier to classes to specify deterministic reductions in Join methods.

Using these three additions, the mechanisms of synchronization (section 3.4.1.5), dynamic communications (section 3.4.1.3) and thread creation (section 3.4.1.6) in Join Java are made straightforward. Synchronization and dynamic communication is only slightly more complex than writing and calling standard Java methods. The syntax of the language extension is provided in Figure 17.

3.4.1.1 Join Methods

A Join method (see Figure 16) in Join Java gives the guarded process semantics of the Join calculus to Java. That is the body of a Join method (See JoinMethod in Figure 17) will not be executed until all the fragments (See IdentFrags in Figure 17) of a Join method are called. If a Join method is defined with pure Java return types such as **void** or **int** the first fragment has blocking semantics. If the return type of the leading fragment is the new type **signal** the

```
int fragment1() & fragment2(int x) {
   //will return value of x
   //to caller of fragment1
   return x;
}
```

Figure 16. A Join Java Method

```
ClassDefn      = [JoinClassMods] class Ident { ClassDefnItems }
ClassDefnItems= [ClassDefnItem] [ClassDefnItems]
ClassDefnItem = VarDecs | BlockDefn | MethodDecl | JoinMethod
JoinMethod     = [Modifier] ReturnType IdentFrags { Statements }
ReturnType     = signal | void |UserDefined | BaseType
IdentFrags     = Ident(ParamList)[&IdentFrags]
Ident          = any legal Java identifier
JoinClassMods = any legal class modifier and ordered
Modifier       = standard Java method modifiers
BlockDefn      = standard Java block definition
VarDecs        = standard Java variable declarations
MethodDecl     = standard Java method declaration
ParamList      = standard Java method parameters
Statements     = standard Java method body
UserDefined    = standard Java class types
BaseType       = standard Java base types

A Join method can be placed at any point that
a normal Java method can be legally placed.
```

Figure 17. Join Java Language Extension Syntax

fragment is asynchronous (an early return type). Trailing Join fragments are always asynchronous. When a fragment is asynchronous, this means they will not block the caller of the fragment. A non-Join Java aware class can call methods in a Join Java class even if the return type is **signal**. In the case of a **signal** return type the caller will return immediately as if it called an empty **void** method. In Figure 18, an example of a Join Java method declaration within a Join Java class is presented.

The Join method would be executed when calls are made to all three fragments (*A()*, *B()* and *C(int)*). A call to method *A()* will block the caller at the fragment call until fragments *B()* and *C(int)* are called due to the requirement that a value be returned of type **int**. When all method fragments have been called the body of the corresponding Join method is executed returning the **int** value to the caller of *A()*. The message passing channel in the example is therefore from the caller of *C(int)* to the caller of *A()* as there is an integer value passed from the argument of *C(int)* to the return type of *A()*. The call to *B()* only acts as a condition on the timing of the message passing. One thing to note is that the fragments *A*, *B* and *C* do not have method bodies

```
final class SimpleJoinPattern {
   int A() & B() &C(int x) {
      //will return value of x
      //to caller of A
      return x;
   }
}
```

Figure 18. A Join Java Class Declaration

of their own. The invocation of any single fragment does not necessarily invoke the method body. Only when a complete set of Join fragments that forms a Join method have been called does a Join method body execute. Static Join Java methods are in principle allowed.

3.4.1.2 Join Method Pattern Matching

A single Join method is a straightforward evaluation; however, Join fragments can take part in different Join methods as is illustrated in Figure 19 below. It can be seen that the Join fragment *A()* takes part in three patterns. Also the asynchronous fragment *D()* takes place in two Join methods. Semantically this means that the first method that is completed (has calls to all its Join fragments) will be executed. A situation that can occur is when a choice has to be made between two or more simultaneously completed Join methods. For example if *B()*, *C()* and *D()* were called followed by *A()* there is a choice in which Join method to run. Two alternatives in handling this situation exist; a non-deterministic (pseudo-random) selection or, some predefined selection policy. In the prototype extension, both options are provided. The default behaviour is to select a pattern randomly. However, the programmer can modify the behaviour of the Join methods by using the class modifier **ordered** in which case patterns are executed in the order in which they are declared**.** As Java's JVM is non-deterministic in relation to thread time slicing it is impossible to design a contention strategy for Join Java. Consequently, in the event of concurrent method invocations it is left to the JVM to decide which thread has precedence. When there are multiple calls to a leading fragment and there are no fragments available to complete the Join method the calls are blocked and then queued internally by the pattern matcher. When a fragment arrives that completes the Join method the blocked calls are released in FIFO order.

3.4.1.3 Dynamic Channel Formation

Parameterized Join fragments can also be used to create channels. For example in Figure 20, a

```
final class SimpleJoinPattern {
   void A() & B() {
   }
   void A() & C() {
   }
   void A() & D() {
   }
    signal D() & E() {
  }
}
```

Figure 19. Shared Partial Patterns

caller to input1 will pass a message to the waiting caller of output via Join method 1. However, the channel creation becomes dynamic when one considers the second Join method which has a different second Join fragment (input2). Now if there is a call to input1 waiting when output is called then Join method 1 executes. However, if there is a call to input2 instead then Join method 2 will be executed. These decisions are made at runtime via a "pattern matcher".

3.4.1.4 Matching Behaviour

The default behaviour simulates non-determinism by randomly selecting one of the simultaneously completed Join methods. If the **ordered** keyword is used simultaneously completed patterns are selected based on their definition order in the source code. Any number of other policies could be added such as "fair" scheduling.

3.4.1.5 Synchronization

One of the problems identified earlier in Java is the ability for the programmer to change the identity of the lock midway through a synchronized block of code. This is done when a user uses the synchronized block syntax to lock a section of code. Consequently, in the extension the necessity of using this access to the lock is removed from the user. This is done by enclosing the locking mechanism within the channel creation handling code of the Join methods. It should be noted that the synchronized keyword could still be used to protect the content of the method from multiple accesses. However, synchronization can be omitted if the programmer was to use a Join fragment to act as a synchronizer (see Chapter 5).

3.4.1.6 Asynchronous Methods

A Join fragment with a **signal** return type indicates that the fragment is asynchronous. Any fragment with a **signal** return type specifies that on being called a thread will be created and started. Figure 21 shows an example declaration of a thread with argument *x*.

```
final class SimpleJoinPattern {
   //Join method 1
   int output() & input1(int parameter) {
      return parameter;
   }

   //Join method 2
   int output() & input2(int parameter) {
      return parameter;
   }
}
```

Figure 20. Channel Example Code

```
class ThreadExample {
   signal thread(SomeObject x) {
      //thread code that uses parameter x
   }
}
```

Figure 21. Thread Example

3.4.2 Type System

The type system in Java was left largely unchanged by the modification. The only modification to the type system is the **signal** return type. This was done by reusing the **void** type's characteristics except for the timing of the return. Consequently, to the type system a call to a **signal** return type Join Java fragment will appear to be a call to an immediately returning **void** method. Internally the **signal** thread creates (or reuses) a thread to create the asynchronous executing code immediately returning to the calling method.

3.4.3 Relation between Join Java and Java

In this section, the interaction of the Join Java extension and the Java language are investigated. Issues regarding inheritance, overloading, interfaces and polymorphism are covered.

3.4.3.1 Inheritance

Join Java classes can inherit other classes just like standard Java. In the current extension, inheritance of Join classes is disabled. Effectively a Join class is declared final. The main reason this has been done is to concentrate on the expression of concurrency and the communications mechanisms. If inheritance were allowed, one would have to deal with the inheritance anomaly. These problems have also been looked at by (Fournet, Laneve et al. 2000) and has been showed to be a difficult problem to solve. A future modification of the compiler could be made to the pattern matcher so that it would check for superclass Join pattern matchers and import those to the base class. This is discussed in section 8.3.

3.4.3.2 Overloading

Join Java supports ad-hoc polymorphism, in which the signatures of methods are defined by both the name and the arguments. This allows programmers to use the same object-oriented features that they normally use in the context of the rest of the language. For example, Join fragments may take part in several Join methods if their signatures and names are identical. Figure 22 shows an example of a class with five distinct Join fragments. The first *A* fragment in

```
final class SimpleJoinPattern {
   void A() & B() //Join method one
   {
   }
   void A(int value) & C() //Join method two
   {
   }
   void A() & D() //Join method three
   {
   }
}
```

Figure 22. Polymorphic Join Java Fragments

the first pattern and third pattern are the same, however, the *A* fragment in the second pattern is different as the signature is different.

3.4.3.3 Interfaces

Join methods are normally an implementation structure consequently they are not available in interfaces. However, Join fragments can be specified in an interface. This allows a user to define the Join methods separate from the definition of the Join fragments. This would mean that you could have dynamic loading of channel definitions. In Figure 23 an example of an interface (*ChannelInterface*) for some Join fragments is presented. Two possible

```
interface ChannelInterface {
   public void b();
   public int a();
   public signal c();
   public signal d();
   public signal f();
}
class Implementation1 implements ChannelInterface{
   public int a() & c() {
        return 0;
   }
   public void b() & d() {}
   public int a() & f() {
        return 0;
   }
}
class Implementation2 implements ChannelInterface {
   public int a() & c() & d() {
        return 0;
   }
   public int a() & c() & f() {
        return 0;
   }
   public void b() & c() & f() {}
}
```

Figure 23. Interfaces in Join Java

```
class ExamplePluggableChannels {
      …
      ChannelInterface channels;

      public static void main(String[] argv) {
            …
            if(someValue) {
                channels = new Implementation1();
            } else {
                channels = new Implementation2();
            }

            …//behaviour dependent on runtime condition above
            channels.f();
            channels.c();
            channels.b();
      }
}
```

Figure 24. Using Interfaces in Join Java

implementations (*Implementation1 and Implementation2*) of dynamic channels are also shown. At runtime, the implementation to use for channel creation can be selected dynamically. Figure 24 gives a short example of using the interface to polymorphicaly select channel creation behaviours at runtime. It can be seen that at runtime the pattern matcher is selected based upon the value of the Boolean variable *someValue*. If the value of the variable is true, the *Implementation1* channel definition is used otherwise *Implementation2's* definition is used. This consequently affects the behaviour of the matching later in the program.

3.4.3.4 Polymorphism

As can be seen in the previous sections and Figure 24 a form of polymorphism is possible with Join enabled classes just as in any normal object-oriented language. Implicitly Join methods themselves are polymorphic, that is the Join fragments that are called at runtime decide which Join methods are eventually evaluated. For example in Figure 25 a runtime decision on which pattern is evaluated based on the previous calls to Join fragments can be seen. When *A()* is

```
class PolymorphismJoinJavaExample {
      //Join method one
      void A() & B() {
          //some code
      }

      //Join method two
      void A() & D() {
          //some code
      }
}
```

Figure 25. Polymorphism in Join Java

called it will depend on whether *B()* or *D()* was called previously as to which Join method is completed. In the former case Join method one will be called, in the latter case Join method two will be called.

3.4.3.5 Inheritance Anomaly

A well known problem with combining concurrency and object-oriented paradigms is the so called inheritance anomaly (Matsuoka, Wakita et al. 1990; Matsuoka and Yonezawa 1993). Simply put, inheritance anomalies lead to the necessity to re-declare a base class method's coordination mechanisms in a derived class. Join Java does not attempt to solve this problem, as it was not the primary focus of this work. At initial inspection, it seems that Join Java neither encourages nor discourages instances of inheritance anomalies. However, having synchronization code in method signatures should make identification of inheritance anomalies easier.

3.4.3.6 Dead Locks

Join Java does not solve the dead lock problem directly. It is still possible to write code that creates a dead lock. In Figure 26 a straightforward dead lock is illustrated. Join method one cannot proceed beyond the call to *C()* until a call to *D()* is made. But the call to *C()* is blocked waiting for a call to *D()* which occurs after it finishes. Of course *C()* cannot finish hence dead locking occurs. Conversely, a similar situation occurs on Join method two.

With the explicit definition of synchronization, the compiler should be able to warn of possible dead lock situations. This could be done by creating a dependency graph of Join fragments and the calls within them and then do a topological sort on the result. Any cycles (which would stop a topological sort) would become immediately apparent and reportable to the programmer as warnings of possible deadlocks.

```
class DeadLockingJoinJava {
      void A() & B() {  //Join method one
         C();
         D();
      }

      void C() & D() {  //Join method two
         A();
         B();
      }
}
```

Figure 26. Deadlock in Join Java

3.4.4 Summary

In this section, the informal syntax and semantics of Join Java were examined. It was shown that with a small number of simple changes to Java a superior mechanism for synchronization can be introduced. The changes better reflect the object-oriented nature of the language. The behaviour of the matching of Join methods and how this leads to dynamic channel formation was also examined. Finally, the issues surrounding the interaction of Join Java and the object-oriented paradigm were examined.

3.5 Conclusion

In this chapter, the concurrency, syntax and semantics of the Join Java superset was described. The chapter began with an examination of the interesting anomalies that occur when writing concurrent Java programs. It was found that even though Java's concurrency mechanism is sufficient it still has problems. Programmers can arbitrarily change lock identities within the body of a supposedly protected region of code. This can lead to code that looks protected being unprotected. Another problem is that synchronization has to be used at the correct places otherwise; any single omission invalidates all other synchronized code. These anomalies suggested avenues for improving the language's concurrency semantics. Using these problems as a starting point, a number of high-level requirements for the extension were then identified. These requirements identified criteria that were critical to the usability of the extension. The chapter then showed how Java could make use of some of the novel features of Join calculus to create a superior synchronization mechanism. The next part of the chapter gave a description of the proposed new concurrency extension that is called Join Java. Finally, the chapter examined the interaction of the Join Java extension to other features of the language.

4

Implementation

A good scientist is a person with original ideas. A good engineer is a person who makes a design that works with as few original ideas as possible.

(Freeman Dyson)

Table of Contents

4.1 Introduction

In this chapter, the implementation of the Join Java compiler is covered. Structurally the extension is divided into two components. The first component is the translator that extends the standard Java compiler to support the new syntax and semantics of the extension language. The translator converts the Join Java code into standard Java byte code, which can then be executed on a standard Java Virtual Machine. The second component of the language is the pattern matcher. The pattern matcher forms a library of classes that are used by the translated code to handle the dynamic reduction of Join method calls. This chapter consequently divides the implementation description into two sections. This chapter demonstrates the feasibility of implementing the Join Java extension in the Java language. The prototype language extension also allows performance issues to be investigated more thoroughly. At the end of the chapter, some of the issues that were discovered in implementing the extension are covered.

4.2 Compiler Choice

When writing this language extension a choice needed to be made whether to create an entirely new compiler, modify an existing compiler or use an available extensible compiler. The major disadvantage of writing a new compiler for a language extension that extends a rich language such as Java is the considerable time required to get a standard compiler to a stable level. The main advantage is that once the base compiler is written the extensions tend to be easier to add, as the developer is very familiar with the entire environment. A major disadvantage of modifying an existing compiler is that it may be extremely difficult to add extensions if the base compiler has no provision for the extensions. Rewrites of the entire compiler front end are frequently required. The other strategy is to adopt an extensible compiler. These compilers give us the advantage that they are designed to be extended thus do not suffer from the headaches involved with modifying a standard compiler. The main disadvantage is usually a performance penalty due to the generic nature of the compiler architecture. The extensible compiler was chosen to implement the Join Java extension. In the following sections we justify that decision.

With respect to engineering the extension, three major requirements for our compiler were identified. The ability to:

1. Translate the extension into standard byte code for JVM portability.

2. Translate the extension intermediate abstract syntax tree (AST) into standard Java code for debugging purposes.

3. Use Java itself as the implementation language of the compiler.

A number of lesser requirements such as access to the designer of the base compiler were also identified. At the time of the design of the language extension, there were limited possibilities that fit these criteria. The core Join Java compiler was based upon the extensible compiler developed by (Zenger and Odersky 2001). This compiler provided an excellent mapping to the requirements mentioned above. Later a number of further possibilities became available such as JSE (Bachrach and Playford 2001) and Maya (Baker and Hsieh 2002) which could also have been used.

4.3 Translator

In this section, an overview of the extensible compiler is provided. The Join Java compiler is based on an existing Java compiler that has a modifiable architecture specially designed with extensibility as a primary design objective. In the first half of this section, a brief explanation of the extensible compiler is given in order to better explain our extensions. More detailed information is available from (Zenger and Odersky 1998). In the second half of this section, a description of how these features have been used to implement the Join Java language is given.

With this extension, it has not been necessary to modify the core architecture of the extensible compiler. All the modifications are in the form of subclassing of existing program structures that are by their nature specific to each version of the language being compiled.

4.3.1 The Extensible Compiler Architecture

The object structure of the extensible compiler(Zenger and Odersky 1998) was specially chosen to make extensions easy by confining each new language feature to extending a limited number of existing classes. The extensible compiler is the work of Matthias Zenger at EPFL (Zenger and Odersky 1998; Zenger and Odersky 2001). The language that the compiler supplies (in this case the standard Java language) will be referred to as the base language. The new language that is being implemented will be called the extension language (in this case Join Java). The extensible compiler was chosen in the belief that extending an existing compiler designed with extensibility in mind would be easier to prototype than starting with an existing standard compiler.

The fundamental idea behind the extensible compiler is that given a base language implemented in the extensible compiler the addition of new language structures is achieved by subclassing of existing program structures. The extensible compilers architecture is similar to many other compilers. It has the standard lexer, scanner, parser[10] and semantic analyser phases in the front end and code generator and optimizer phases in the back end. For details of these standard elements see any of the more popular standard texts such as (Aho, Sethi et al. 1986). Each phase creates, modifies or uses an abstract syntax tree. The lexer, scanner and parser together create the abstract syntax tree, the semantic analyser attributes it and the backend uses it to generate byte code. Figure 27 illustrates the modules of the base compiler without translation. In the standard compiler there is a standard syntactic analyser that creates the abstract syntax

[10] These three are sometimes called the syntactic analysis phase together

tree, the semantic analyser that checks and attributes the abstract syntax tree, and the backend, which turns the abstract syntax tree into byte code via the class writer.

In the following section, abstract syntax trees are examined and an explanation of how the base compiler shown below in Figure 27 is modified to translate a program compiled in the extension language to standard byte code.

4.3.1.1 Extensible Abstract Syntax Trees

For the compiler to be extensible not only does the compiler architecture need to be extensible, so do the data structures on which it works. In the case of the extensible compiler, the abstract syntax tree is constructed in an object-oriented manner. The abstract syntax tree is generic

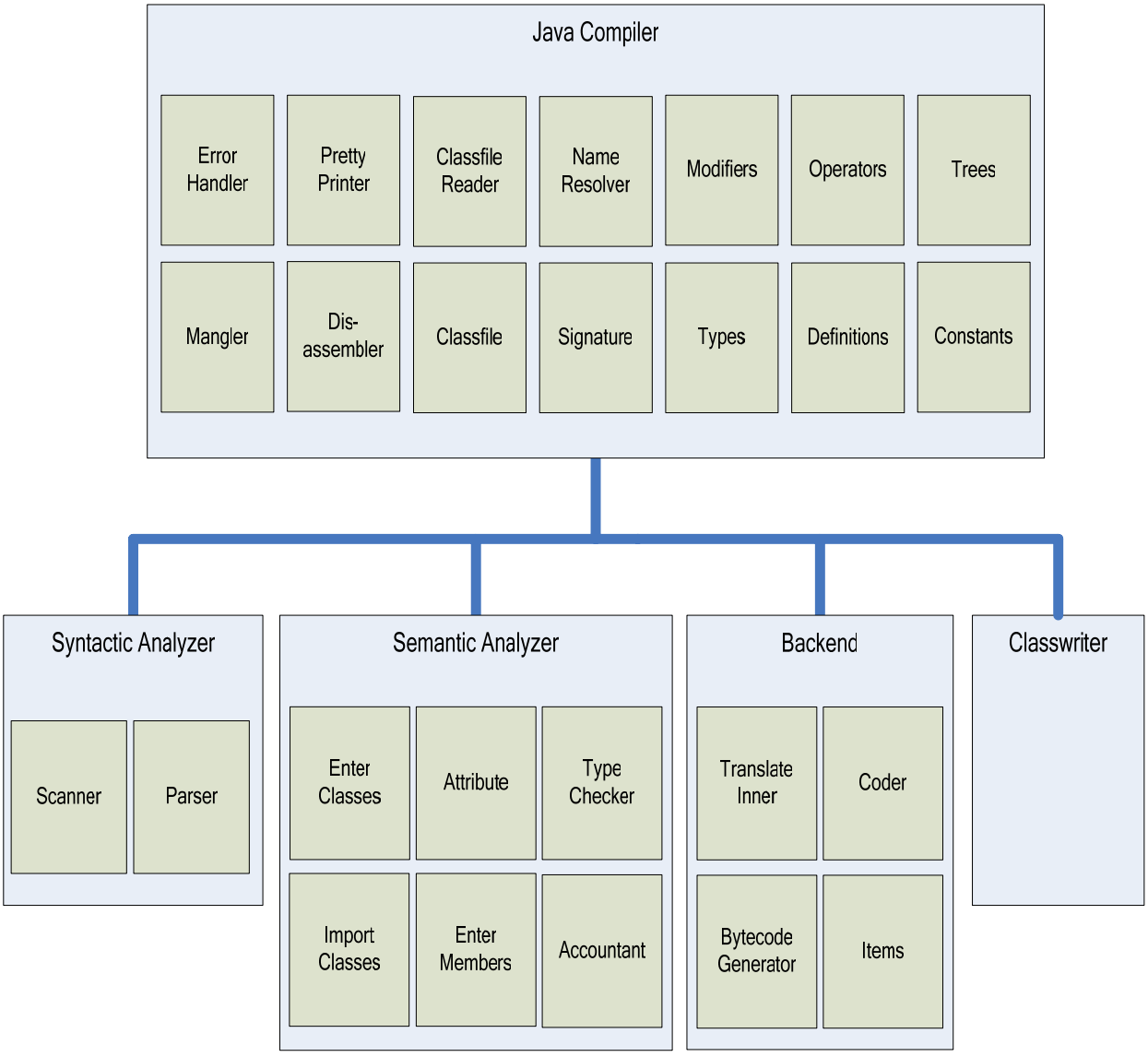

Figure 27. Stages of Extensible Compiler without Translation

enough to store nodes that are designed after the base compiler was written. That is all nodes have a parent type *Tree* whilst each node represents the particular type (eg compilation unit, method, statement etc…) of the tree node. This means the polymorphic design of the abstract syntax tree data structure allows the tree to store any subclass of the tree node allowing extended components to be added later. Therefore, to make an addition to the data structure a subclass node representing the new language feature must be created to attribute the tree for the Join Java abstract syntax tree. To make changes to the abstract syntax tree there also must be appropriate tree processing functions in the compiler architecture. It is therefore necessary to subclass components of the base language syntactic and semantic analysers. In this way whenever the semantic phase comes across an extension component it knows what to generate in the extension abstract syntax tree.

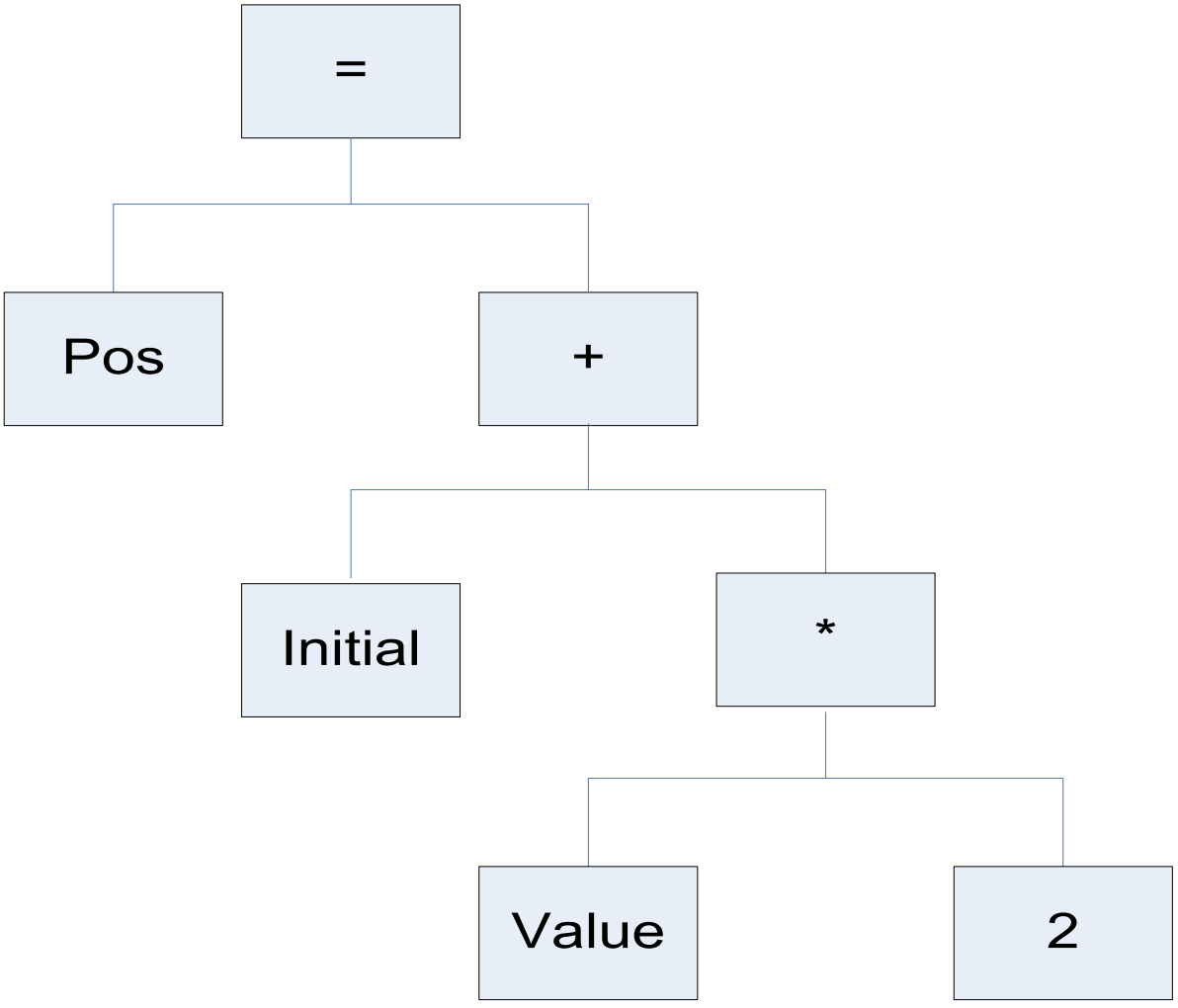

Figure 28. Abstract Syntax Tree before Syntactic Analysis

4.3.1.2 Syntactic Analysis

The syntactic analysis phase of the extensible compiler is similar to most of the other compilers. A language definition file is written and a look ahead left to right (LALR) parser generator (Hudson 1996) is used to generate the scanner and parser code. To make an extension the programmer obtains the language description file (CUP file) for the base language (with the base language parsing code) adds the extension language components and regenerates the parser and scanner phase code. The new scanner/parser can then generate an un-attributed/undecorated abstract syntax tree for the extension language using the extended abstract syntax tree. A simple example of one of these trees is shown in Figure 28. In the example, a simple extension of a language that supports multiplication as well as the addition of the base language is illustrated. The example, whilst being simple shows how the process of translation occurs. The changes to the Java grammar for Join Java are given in section 4.3.2.

4.3.1.3 Extension Semantic Analysis

The semantic analysis phase constructs an attributed tree decorated with type information. It then performs type checking, scope checking, and construction of symbol tables for classes, methods, variables and constants. When compiling a program written in the extension language the semantic checker uses additional checks designed by the extension writer to make sure the code obeys the language rules for the extension. In our example illustrated in Figure 29 it can be seen that the tree has been decorated with type information. However, in the base language the multiplication operation is not supported so the tree needs translation.

4.3.1.4 Translation

The translator phase converts the attributed extension abstract syntax tree into an attributed base language abstract syntax tree. The abstract syntax tree output by the semantic analyser is read by a translator. In the base language version of the compiler, this phase does nothing other than pass over the tree. The translator is provided for the sole purpose of allowing compiler writers implementing language extensions to sub-class the translator. The extension language writer creates a subclass of the translator that in the event of encountering an extended component of the language converts the node into base language components semantically equivalent to the extension language. In our example this means converting the multiplication nodes of the tree into addition nodes semantically equivalent to what the multiplication nodes are trying to express. The modified tree is illustrated in Figure 30 where the multiplication node has been replaced with addition nodes that achieve the same aim. This means that from this point on, for the purpose of compilation, the abstract syntax tree is a base language tree not an extension tree.

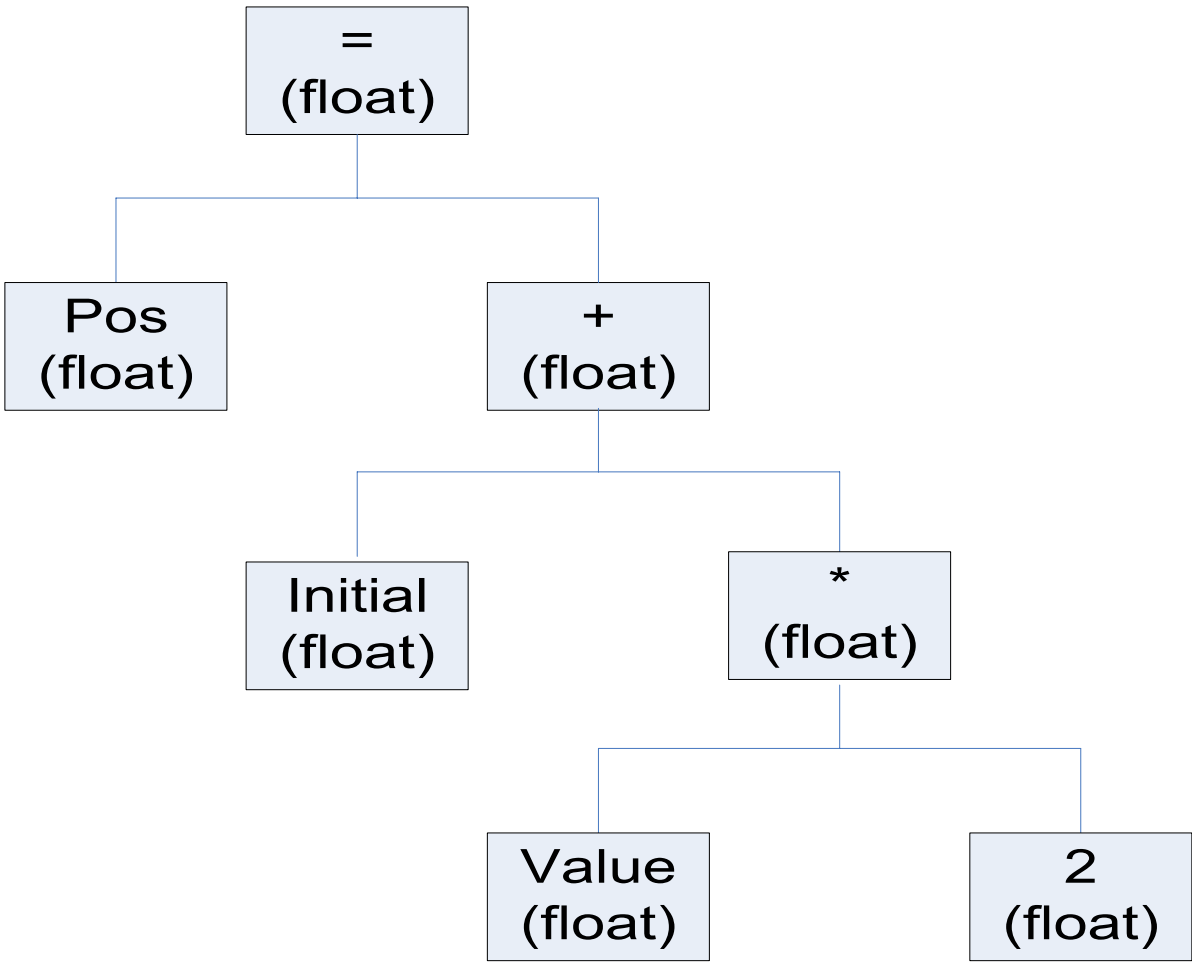

Figure 29. Abstract Syntax Tree after Semantic Analysis

This has the advantage that the back end does not need to be changed if standard Java byte code is to be output.

The resultant tree after the translation phase generally needs to be run through a standard Java semantic analyser (called a silent semantic analyser) to check for translation errors. This phase checks to make sure the translator has generated a correct base language abstract syntax tree. The silent semantic analyser also attributes the newly translated nodes of the tree. After the silent semantic analyser has finished, the abstract syntax tree is passed to the backend for code generation. Our running example is illustrated in Figure 31.

4.3.1.5 Backend

As the backend of the existing compiler is not altered in the case of Join Java this will not be discussed in this chapter. For more details see (Zenger and Odersky 1998).

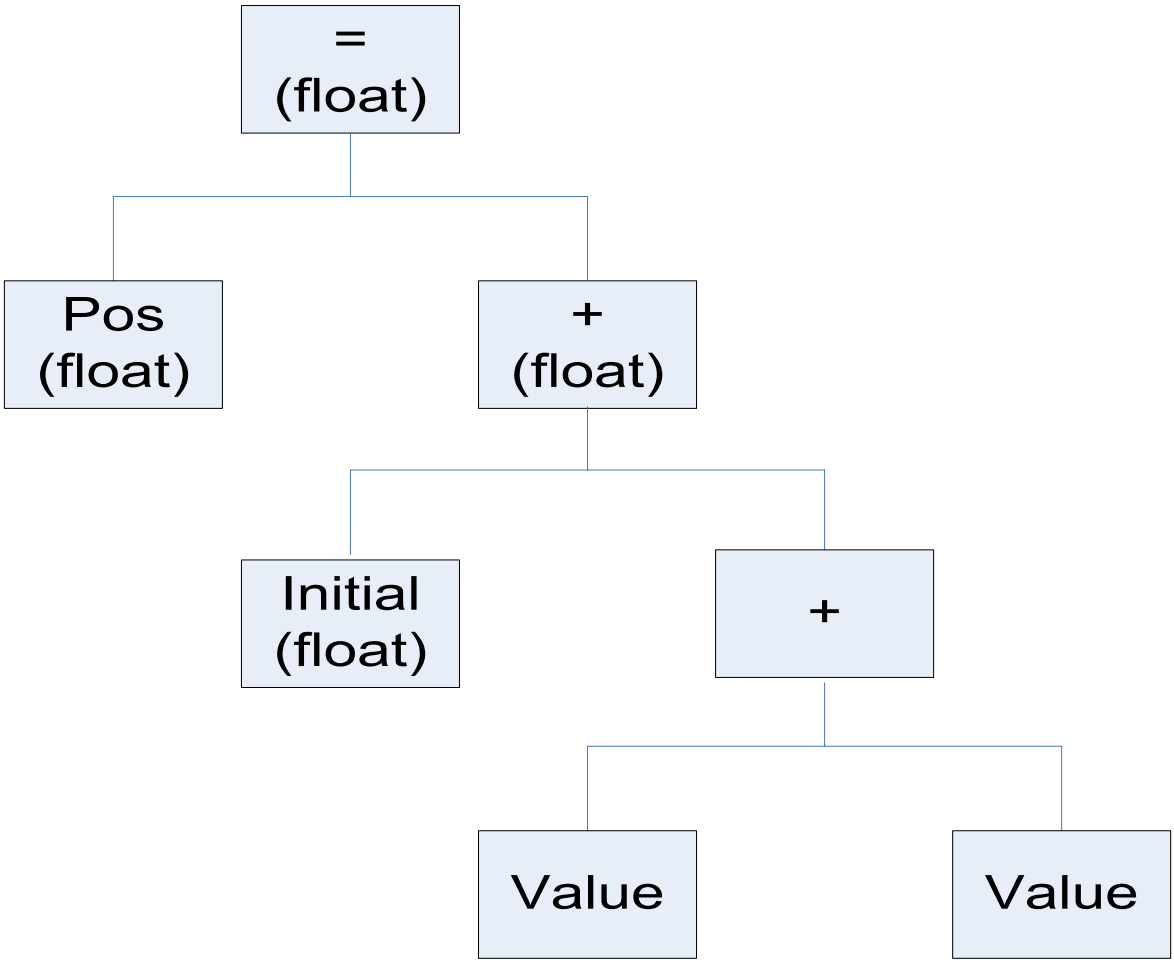

Figure 30. Abstract Syntax Tree after Translation before Silent Semantic Analysis

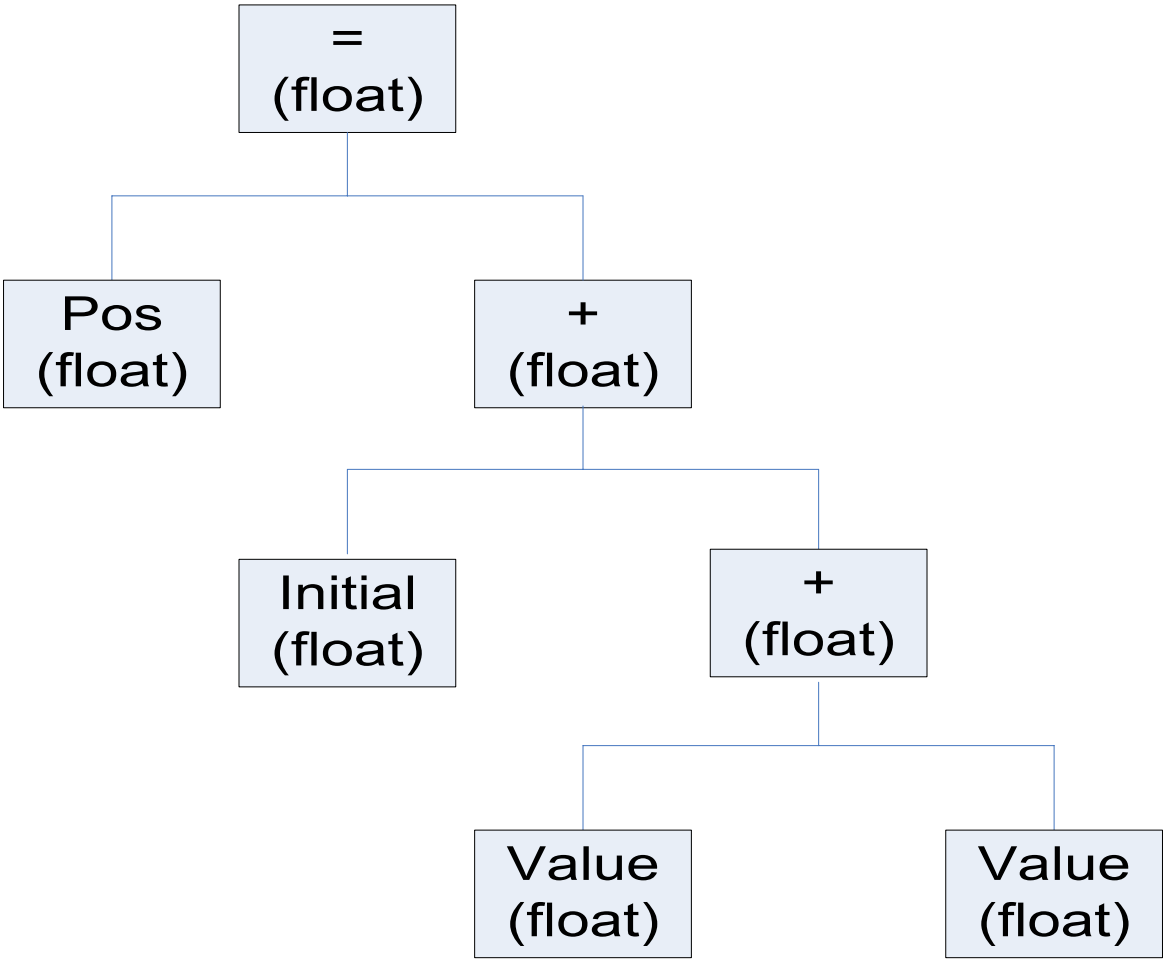

Figure 31. Abstract Syntax Tree after Silent Semantic Analysis

4.3.2 Changes to the Extensible Compiler for Join Java

In Join Java, there are a number of changes that were required to be made to the base compiler. Firstly, the language semantics needed to be extended to support the idea of asynchronous methods. Secondly, the language syntax was extended to support Join Java methods and finally code generation to support pattern matching between Join method fragments was required. This extension adds to the base compiler, new syntactic analysis phase, a new translation phase as well as an extra semantic analysis phase. It was also necessary to add a number of subsidiary classes to support the extensions to the abstract syntax tree and tree walkers for the modified structures. These changes are shown in Figure 32 in yellow. In the following section, these additions will be described in detail.

In this section, all the changes needed to implement Join Java in the extensible compiler are described.

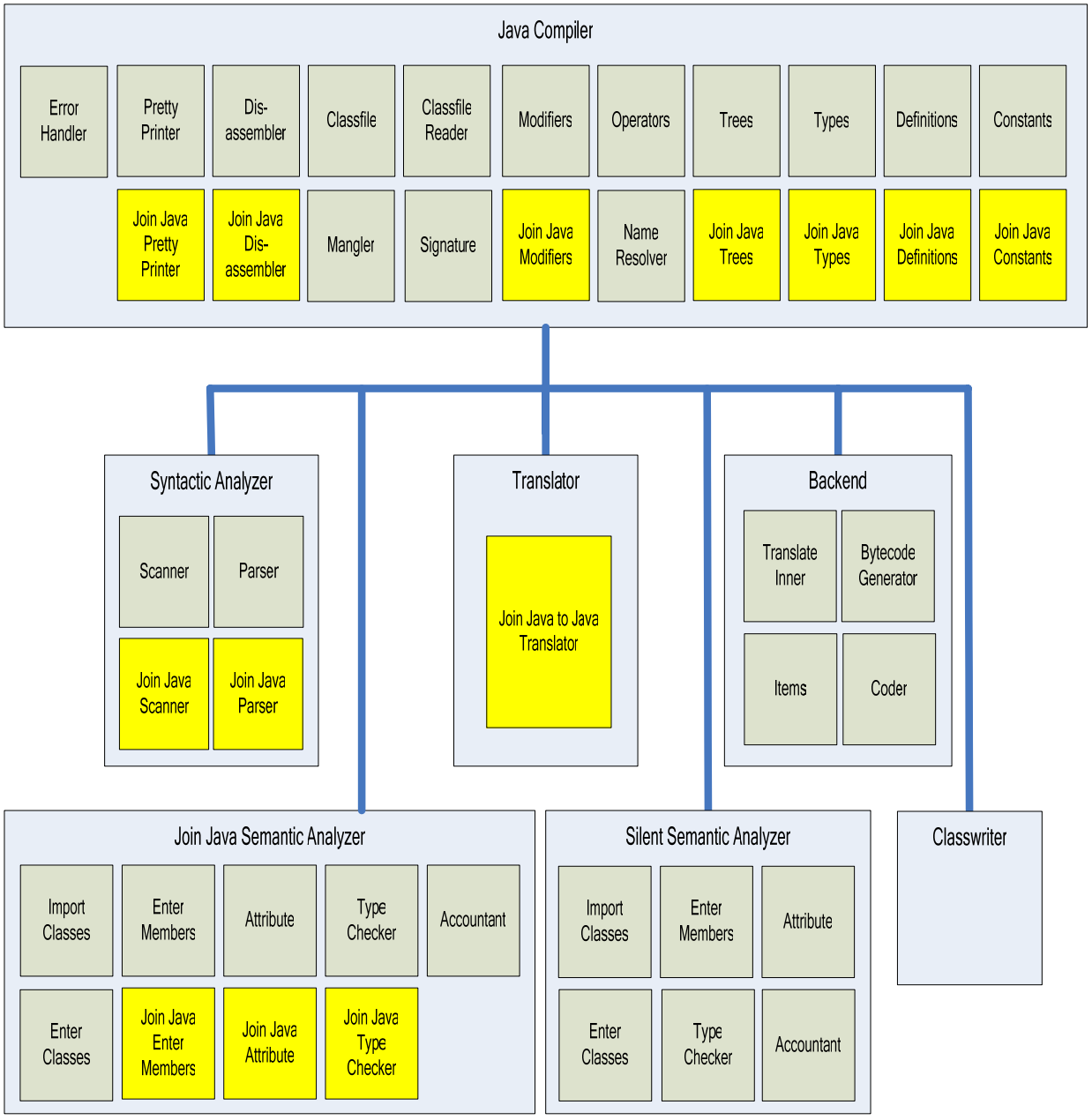

Figure 32. Structure of the Join Java Compiler

The translator was subclassed to allow it to process Join Java elements on the abstract syntax tree and produce a pure Java syntax tree. To assist in the explanation of the translation an example of a small Join Java program is used. This *HelloWorld* Program will be used to illustrate the translation. Figure 33 shows the Join Java code. The program starts with two threads, the first *thread1*, which takes a string "helloworld" and passes it to the second thread *thread2* via a Join Method; the second thread then prints the string out to the console.

```
class HelloWorld {
  signal thread1() {
    call2("helloworld");
  }
  signal thread2() {
    System.out.println(call1());
  }
  String call1() & call2(String value) {
    return value;
  }
}
```

Figure 33. Simple Join Java Hello World Program

In order to compile and translate this to Java byte code an abstract syntax tree, syntactic analyser, semantic analyser and a translator will need to be created. In the case of Join Java, these are method nodes, return types and some modifiers for the class nodes that must be subclassed. These parts are designed and added to the standard Java syntax tree to generate the Join Java syntax tree. The compiler architecture is then extended to handle the new tree structures. Finally, a translator is produced to convert the extension language abstract syntax tree into the base language abstract syntax tree. These alterations are now discussed referring to the *HelloWorld* example program above.

4.3.2.1 Extended Abstract Syntax Tree

To store the superset extension of Java it is necessary to create additional node types for the extensions' abstract syntax tree. This extension required the nodes representing classes and methods to be subclassed and new nodes representing Join methods, Join fragments and Join classes to be created. The most important change is the node representing methods within the abstract syntax tree. The original structure of the tree was modified to allow a number of Join fragments to comprise the method signature rather than the customary single method signature. The original method signature was modified to be a Join fragment for the Join method. In this way, the Join fragments can be used in other classes without those classes needing to know about Join Java. The side effect of this is that less code needs to be modified when the Join fragments are called.

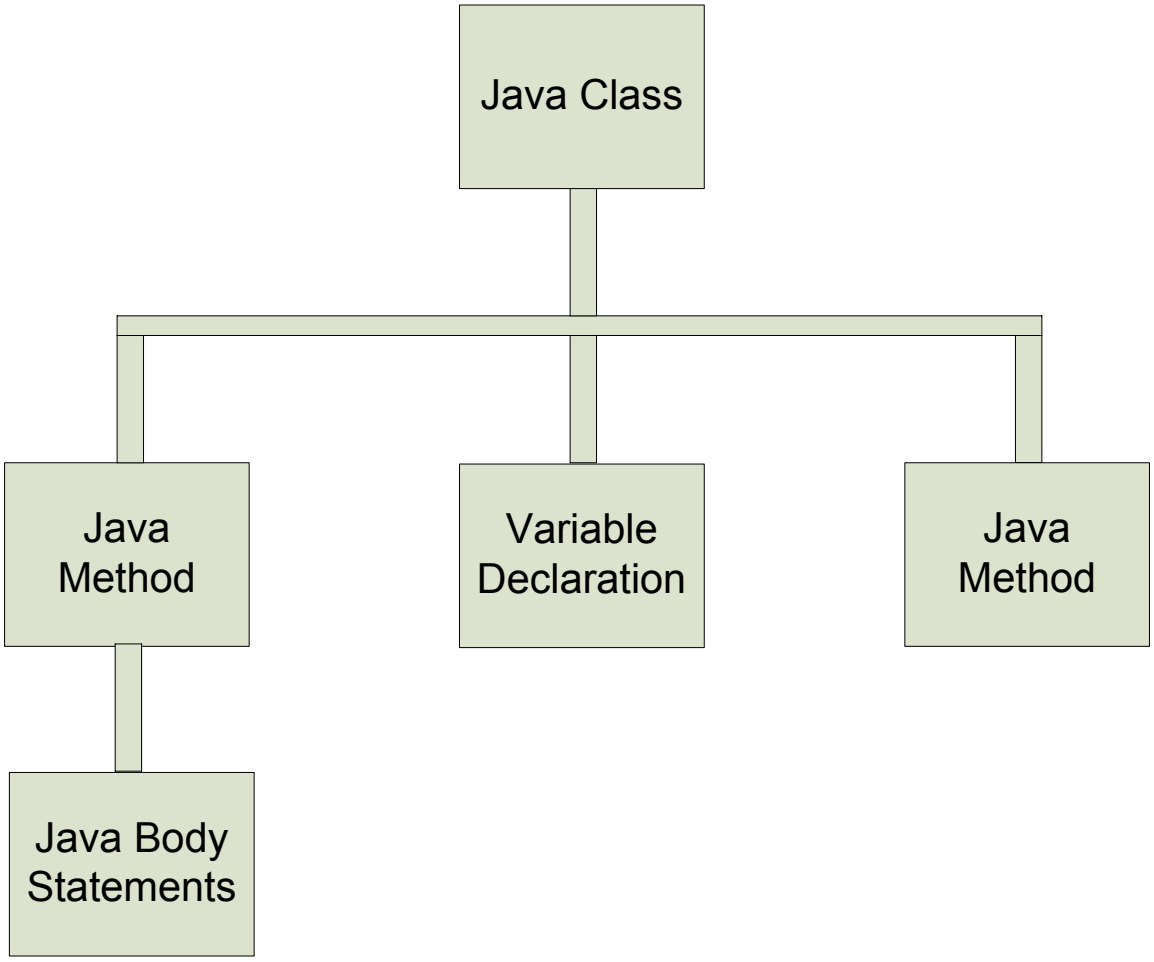

Figure 34. Standard Java Abstract Syntax Tree Example

In Figure 34 an example of a simple abstract syntax tree of Java is shown. The modifications to that tree to support Join Java methods is shown in Figure 35. It can be seen that there are two main changes visible. Firstly, a class node is now a Join Java class node that supports not only standard methods but Join Java methods as well. Secondly, the Join Java method whilst still having a body, now has additional branches representing each of the fragments that make up the compound method signature. The final point to note on the modifications of the tree structure is that the Java method node itself is reused by Join Java with some minor changes to represent a Join Java fragment.

The Join Java class node differs from a standard Java class node in two ways. Firstly Join Java classes have an extra modifier **ordered** that affects the evaluation of Join Java patterns in the event of multiple reduction possibilities (see the semantic analysis and pattern matching sections). Secondly, a Join Java class node may have an additional branch type that is a Join method node.

The Join Java method node has a number of differences from a standard Java method node. The most important difference is that the Join Java method is composed of a number of method signatures (within sub-nodes), unlike Java that only contains one method signature. To distinguish this from the signature of the entire Join Java method they are referred to as fragments. The first fragment is stored in the Join Java method fragment for efficiency and backward compatibility. However, the second and subsequent fragments are stored within sub-nodes called Join Java Fragments. The return type of the first fragment becomes the primary return type of the Join Java pattern. The return types of the sub-node Join Java Fragments are always **signal** which is explained in more detail in the following sections.

The final major abstract syntax tree structural change is the Join Java Fragment node. These are

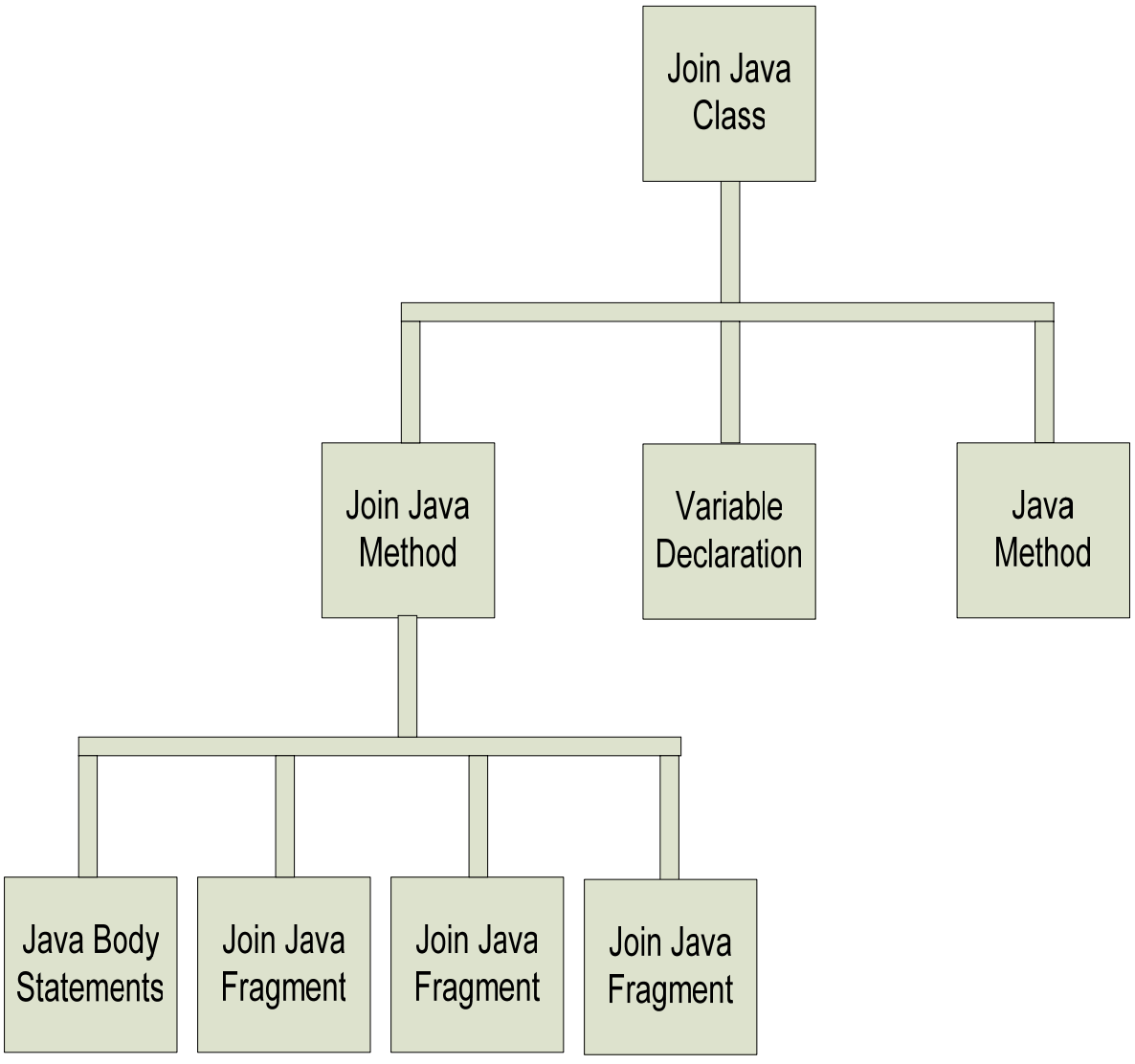

Figure 35. Join Java Abstract Syntax Tree Example

functionally very similar to standard Java method nodes in the standard Java abstract syntax tree. However, there are a few differences. Join Java fragments must have a return type of **signal** and have no body of their own as they take part in Join Java method signatures.

4.3.2.2 Syntactic Analyser

The syntactic analyser converts the source code representation of the Join Java program into an unattributed abstract syntax tree. In the language extension, the standard Java grammar specification is modified so that the grammar represents the additions/changes of the extension language. The language definition was extended to include additional class modifier ***ordered***, the additional return type ***signal*** and new compound method signature for representing Join patterns. The Join Java grammar changes made to the Java grammar are presented in Figure 36. The parser and lexer are generated from this CUP specification using a standard parser generator such as JavaCup (Hudson 1996). Using the example from Figure 33 it can be seen that the syntactic analyser would accept the signal return types and the compound method as they are valid expressions in the grammar illustrated in Figure 36. After the syntactic analyser pass has finished the compiler will hold a Join Java abstract syntax tree representing the example program.

```
ClassDefn      = [JoinClassMods] class Ident { ClassDefnItems }
ClassDefnItems= [ClassDefnItem] [ClassDefnItems]
ClassDefnItem = VarDecs | BlockDefn | MethodDecl | JoinMethod
JoinMethod     = [Modifier] ReturnType IdentFrags { Statements }
ReturnType     = signal | void |UserDefined | BaseType
IdentFrags     = Ident(ParamList)[&IdentFrags]
Ident          = any legal Java identifier
JoinClassMods = any legal class modifier and ordered
Modifier       = standard Java method modifiers
BlockDefn      = standard Java block definition
VarDecs        = standard Java variable declarations
MethodDecl     = standard Java method declaration
ParamList      = standard Java method parameters
Statements     = standard Java method body
UserDefined    = standard Java class types
BaseType       = standard Java base types

A Join method can be placed at any point that
a normal Java method can be legally placed.
```

Figure 36. Join Java Additions to Java Grammar

4.3.2.3 Semantic Analyser

The first semantic analyser phase in addition to doing the standard checks and attribution for the Java nodes of the abstract syntax tree, also handles the checking and attribution of the Join Java nodes. Consequently, in the compiler extension a walker class passes over the abstract syntax

tree using the Java semantic analyser for standard Java code and the Join Java semantic analyser for Join Java code.

As part of the semantic analysis phase, the nodes in the tree are attributed with type information. This language extension has minimal type additions to Java. The main type change being the introduction of an asynchronous return type **signal.** This type has a number of similarities to the standard Java **void** type in that they both indicate an absence of return information. The only difference being that **void** means that the method returns no information other than the timing of the return. **Signal** on the other hand does not even return the timing of the method. Consequently, in our language due to the similarity of the two types the Join Java extension maps **signal** to **void** in the internal representation. The major benefit is that other classes that are not Join Java aware may call **signal** return methods without difficulty. An alternate approach would be to make a modifier called asynchronous which makes the method a thread. This avoids the issues with modifying types. However, this was not done for a number of reasons;

1. If the language were extended to allow synchronous fragments anywhere in the Join method signature one would have to place the asynchronous modifier in front of most Join fragments leading to clutter in the method signature.

2. By definition, all asynchronous methods will have to be **void**. It does not make sense to allow asynchronous modifiers for non-**void** return types. Consequently, if this was implemented as an asynchronous modifier wherever asynchronous was used a **void** must always follow. This proves to be an awkward syntactic design

The Join class node needs standard Java style semantic checking in addition to the checks for the Join methods and Join method fragments. One of the design criteria of Join Java is that it be as sympathetic to the base language as possible. Java supports ad-hoc polymorphism in that the signature of a method is composed of the identifier and the types of the parameters. In Join Java the same idea has been used. A Join Java method is defined by the identifier of each Join fragment plus their respective parameters. This makes symbol entry into the symbol table complicated but makes the extension language more like the base language. A related check that needs to be done is that the return types of method fragments must match the return types of any other fragment with the same signature. This can be complicated especially when the return type difference is **signal** to **void**. The structure of the parser tends to hide the differences between these two types. Consequently, once the semantic checking phase has finished there

should be one entry per fragment signature in the symbol table even if the fragment is used in several Join methods. This is important for the next phase of the translation where calls to the pattern matcher are generated. The semantic checking phase must also check for modifiers in the class definition. The Join class modifier, **ordered** changes the behaviour of the pattern matcher. Consequently, the semantic analysis of this modifier is simply checking that it is only used on the class definitions not on the method definitions.

4.3.2.4 Translation

The major phase of the compiler extension is the translation phase. This phase crawls across the Join Java abstract syntax tree converting it into a Java abstract syntax tree. When it has finished it will leave a semi attributed Java abstract syntax tree. To simplify the presentation of the extensions the less complex extensions are described first and progressively more sophisticated features of Join Java are then described. The order of presentation of features will be;.

1. Asynchronous method calls.

2. Join methods with a **signal** return type (asynchronous Join method with object parameters)

3. Join methods with non-**signal** return types (synchronous Join method with object parameters and return type)

4. Passing base type parameters to Join method as opposed to object types.

The tree manipulation in the translator can require a number of different modifications to the tree. It could require one tree node being replaced by another tree node, one tree node being replaced by a sub-tree or alternatively a node being deleted. Consequently, the tree manipulation functions take the form of pruning and replacement functions that edit the abstract syntax tree. As these functions will not be covered in this thesis consult (Zenger and Odersky 1998) for further information. Translation happens in the following way. A tree walker is supplied which moves over the tree detecting sections of the tree that are Join Java specific. When a Join Java region is detected, the sub-tree is pruned replacing the section of the tree with the functionally equivalent Java sub-tree.

When the compiler detects Join methods within a class it modifies the inheritance structure of the class and generates a number of support methods. The translated Join Java class is made to inherit the *JoinInterface* interface so that the pattern matcher (for more information about the pattern matcher see Section 4.4) can make call backs at runtime to notify waiting threads.

In addition to the implementation of the Join interface, a number of methods are also created. Firstly, an initializer method is generated that creates an instance of the pattern matcher (see section 4.4) passing a reference to the Join Java object. This reference is of type *JoinInterface,* which allows the pattern matcher to make call backs to a generated method within the body of the Join Java class. The *initJoin* method then tells the pattern matcher about the Join methods within the class. The initializer is illustrated in Figure 37, where the method firstly checks for the existence of a pattern matcher and if it does not exist, creates one. Once the pattern matcher exists, it then tells the pattern matcher what matching behaviour to use. This is done by checking the status of the **ordered** modifier, if the **ordered** modifier is present sending a true as a parameter to the *setOrdered* method of the pattern matcher. The *initJoin* method also tells the pattern matcher what patterns are present in the class. This information is passed by the pattern matchers *addPattern* method. The *addPattern* method has two arguments the first argument is a list of Join fragment identities (integer values) that take place in the pattern and the second is whether this method is synchronous (**false**) or asynchronous (**true**). If a Join fragment appears in more than one Join method, its identity will appear in each methods' *addPattern* call. Consequently looking at the translated HelloWorld program from Figure 33 the initJoin method fragment zero is the *thread1* Join fragment, fragment one is the *thread2* Join fragment, and fragments two and three are the *call1* and *call2* fragments respectively.

```
private join.system.joinPatterns all$;
synchronized void initJoin$() {
   if (all$ != null)
      return;
   join.system.joinPatterns alllocal$ =
            new join.system.joinPatterns(this);
   alllocal$.setOrdered(false);
   alllocal$.addPattern(new int[]{0}, false);  //thread1
   alllocal$.addPattern(new int[]{1}, false);  //thread2
   alllocal$.addPattern(new int[]{2, 3}, true);  //call1&call2
   alllocal$.noMorePatterns();
   all$ = alllocal$;
}
```

Figure 37. Hello World Initializer Method in Translated Code

```
   public java.lang.Object dispatch$(
          final join.system.returnStruct retStruct)
          throws join.system.joinTranslatorException {
      switch (retStruct.patternCompleted) {
         case 0:
            new java.lang.Thread() {
               public synchronized void run() {
                  $_thread1();
               }
            }.start();
            break;
         case 1:
            new java.lang.Thread() {
               public synchronized void run() {
                  $_thread2();
               }
            }.start();
            break;
         case 2:
            return call1_$_call2(
              (java.lang.String)retStruct.getObjectArg(3, 0));
         case -1:
            break;
         default:
            throw new join.system.joinTranslatorException(
              "Compiler Error: [JTrans0]: Unexpected case Dispatch");
      }
      return null;
   }
}
```

Figure 38. Hello World Dispatch Method in Translated Code

The second method that is generated is the dispatch method. This method acts as the index of Join methods within the class. Because the pattern matcher is dealing with various base and object parameters along with potentially different return types a generic structure must be created. To make the pattern matcher as simple as possible signatures are replaced with an integer index. When a reduction (pattern match) occurs the pattern matcher returns an index to the completed pattern. When a call occurs to a Join fragment the pattern matcher packages up the arguments into a storage structure (*returnStruct*) and waits for a completion to occur. A completion takes place when all fragments required for a Join method are available. When that completion occurs the *returnStructure* is modified to contain all the parameters of the Join fragments along with the index of the completed pattern. That *returnStructure* is then passed to the dispatch method by the translated method (see later). Each case in the dispatch method identifier equates to the order of *addPattern* calls in the *initJoin* method.

In Figure 38, above the dispatch method has three cases representing the three methods from the example code previously shown in Figure 33. The case zero relates to the method *thread1*, case one to *thread2* and case two to *call1()&call2()*. The -1 case is for finishing asynchronous methods that are not required to be blocked. The remaining case *default* relate to error checking for the translation and debugging. When a pattern has a **signal** return type the call to the method body is wrapped in an anonymous inner thread. This creates the asynchronous behaviour of the **signal** return type. If the return type is **void** the method is simply called. If the return type is an object the *returnStructure's getObjectArg* method is called to retrieve the object from the structure. The return value is then returned to the caller which is the translated version of the Join fragment.

The third generated method is a notify method, (Figure 39) which acts as a synchronized access point for the pattern matcher to modify the object's monitor. This method could be better designed in an optimized version of the compiler however; it works for proof of concept with reasonable speed.

```
public void notifyJoin$() {
   synchronized (all$) {
      all$.notifyAll();
   }
}
```

Figure 39. Hello World Notify Translated Code

The final translated code for the language extension is the Join methods/fragments to Java methods conversion. For every Join fragment, a new Java method is created. In Figure 40 the Join method *call1() & call2(<String>)* translation is shown. The translated methods *thread1* and *thread2* would be similar to *call2* and hence are omitted for brevity.

```
String call1() {
   if (all$ == null)
      initJoin$();
   join.system.returnStruct retval$ =
        all$.addSynchCall(
        new java.lang.Object[]{}, 2, this);
   while (retval$.ready != true) {
      try {
         synchronized (all$) {
            all$.wait();
         }
      }  catch (java.lang.InterruptedException ex) {
         Throw new
             join.system.joinTranslatorException
                 ("Interrupted Exception …");
      }
   }
   return (String)dispatch$(retval$);
}

void call2(String value) {
   if (all$ == null)
      initJoin$();
   join.system.returnStruct retval$ =
        all$.addCall(new java.lang.Object[]{value}, 3);
   dispatch$(retval$);
   notifyJoin$();
}

String call1_$_call2(String value) {
   return value;
}
```

Figure 40. Hello World Join Method Translated Code

The content of the translated Join methods is dependent on the synchronicity of the Join fragment. The simpler translation of the two is presented first. If a method is asynchronous, for example *call2* that has an asynchronous return type, there is no requirement for return messages to be sent back to the caller. Consequently, the translation is simpler and needs only to check for the availability of a pattern matcher then pass the arguments to the pattern matcher. If the addition of this fragment call to the pattern matcher results in a pattern being completed of which the fragment is the first fragment then the *returnStructure* that is returned from the pattern matcher will have a completion information including the identifier of the completed pattern. If the fragment is not completing a pattern or it is not the first fragment of the completed pattern the pattern matcher will return a -1 identifier which tells the dispatch method to do nothing and end.

The other possible translation of a Join fragment is if the Join fragment is synchronous, that is it has a return type of **void**, a base type or a object type. In this case the caller needs to wait on the completion of these fragments. The translated methods are more complicated as they have to

exhibit blocking semantics. As in the asynchronous method, the first step in the translated code is to check for the existence of a pattern matcher. The method then adds the call to the pattern matcher that immediately returns a *returnStructure* containing information about the success or failure of the match. If there is no match the *returnStructure* field *ready* is set to false and the translated method enters a while loop that blocks until a *notify* occurs. Eventually when this fragment is completed by other fragments being called, the *ready* field is changed to true and the pattern matcher calls *notify* and the dispatch method is called with the completed *returnStructure*. The dispatch method will then call the body of the Join method. The method body method is constructed by appending the names of all the fragments with _$_ between them and then collecting all the parameters of the various fragments and making them the parameters of the method body method. This can be seen in Figure 40 where the method *call1_$_call2* contains the method body code and has the parameters of all the fragments. The return type of the translated method is the same return type as the Join method that was translated. The dispatch method will also return an object that is cast back to the type of the return type of the method. This is how the return values are passed back out of the pattern matcher to the caller of the pattern matcher.

The following sections show specific translation examples. Firstly, single fragment asynchronous methods are covered followed by asynchronous multiple fragment Join methods. Following this, base type parameter passing translation is covered. Finally, an example program with multiple patterns sharing some of the same fragments is illustrated. For conciseness only the relevant parts of the translation in the examples is provided.

4.3.2.4.1 Asynchronous Method Example

In the first example program illustrated in Figure 41 a single method with **signal** return type is shown. This is the equivalent of the thread1 in our running *HelloWorld* example. Semantically the **signal** return type indicates an asynchronous method. Consequently, in the code it can be seen that whenever the *x* method is called the dispatch method wraps the call in a thread and starts it.

```
class Example1 {
   signal x() {
      System.out.println("Example1");
   }
}
```

Figure 41. Asynchronous Method Source Code

Selected code from the translation is shown in Figure 42. It can be seen that when the *x* method is called the translated code is executed which adds the call to the pattern matcher. This immediately matches as the pattern has only that fragment in it. Once the call to the pattern matcher occurs a *returnStructure* is returned containing the index of the completed pattern. This is then passed to the dispatcher which then executes the body of the method (named *$_x*) in a thread.

```
void x() {
   if (all$ == null)
      initJoin$();
   join.system.returnStruct retval$ =
                all$.addCall(new java.lang.Object[]{}, 0);
   dispatch$(retval$);
   notifyJoin$();
}

void $_x() {
   System.out.println("Example1");
}


//from dispatch method
      case 0:
         new java.lang.Thread() {
            public synchronized void run() {
               $_x();
            }
         }.start();
         break;
 …
 return null;
```

Figure 42. Asynchronous Method Translated Code

The pattern matcher is invoked in this code as there may be other patterns that contain this fragment (eg via using **ordered** modifier). The code could be optimized to detect the situation where fragments do not appear elsewhere and do not use the pattern matcher dispatcher to execute the code. This would speed up the program, as it would no longer need to go to the pattern matcher to check for a match. However, in the first version of this compiler the optimizations were kept to a minimum in order to allow for further extensions. A second performance issue that may be noticed is the superfluous notify call at the end of the translated method. This is placed in the code for situations where asynchronous methods are completing patterns that have a waiting synchronous fragment. In this example, the call is unnecessary and a performance penalty is paid. Even with minimum optimizations, the compiler has acceptable performance penalties (see Chapter 7).

4.3.2.4.2 Asynchronous Join Java Method Example

The next translation example is an asynchronous Join method with **signal** return type. In the example in Figure 43 a Join method containing two fragments ***x*** and ***y*** is shown. The meaning of this is, when a call to both **x** and **y** occur run the body of the method asynchronously. The important portion of the translation for this is presented in Figure 44.

```
class Example2 {
   signal x() & y() {
      System.out.println("Example2");
   }
}
```

Figure 43. Asynchronous Join Java Pattern Source Code

In this translation two Join fragments can be seen; ***x*** and ***y*** are translated exactly as the previous example. However, in this case the method body will not be executed until both Join fragments are called, at which time the last called asynchronous method will go into dispatch with a completed pattern identity. The dispatch method wraps the call to the method body in a thread and starts the thread returning immediately.

```
   void x() {
      if (all$ == null)
         initJoin$();
      join.system.returnStruct retval$ =
         all$.addCall(new java.lang.Object[]{}, 0);
      dispatch$(retval$);
      notifyJoin$();
   }
   void y() {
      if (all$ == null)
         initJoin$();
      join.system.returnStruct retval$ =
         all$.addCall(new java.lang.Object[]{}, 1);
      dispatch$(retval$);
      notifyJoin$();
   }

   void x_$_y() {
      System.out.println("Example2");
   }


   //from dispatch method
         case 0:
            new java.lang.Thread() {
               public synchronized void run() {
                  x_$_y();
               }
            }.start();
            break;
```

Figure 44. Asynchronous Join Java Pattern Translated Code

4.3.2.4.3 Synchronous Join Java Method Example

One of the most difficult aspects of the translation phase is the handling of fragment parameters and return values. In Java, there is no convenient way of handling variable parameters and variable types. In less strictly typed languages one may bypass the type system by taking liberties with parameter lists by disabling the type recordings (such as C's **void*** type). With Java however, this is not a possibility consequently one must use a combination of casting for objects and boxing for base types both of which are expensive operations. Neither of these operations are significantly optimized in the existing JVM (Gosling and McGilton 1996; Sun 1996) in fact boxing is not a supported instruction as it is in more recent virtual machines such as that used in the Microsoft .Net architecture (Schanzer 2001). In the next two examples, parameter passing is examined and how it is handled in the Join Java translation. In the first example, reference type passing is examined.

```
class Example3 {
   String x() & y(String val) {
      System.out.println("Example3"+val);
      return val;
   }
}
```

Figure 45. Synchronous Join Java Source Code

Figure 45 shows an example Join method containing two fragments one of which has a parameter. There is also a return type of String for the first fragment *x*. This means the caller to *x* is blocked until the entire Join method is completed by a call to *y(<string>)*. Consequently, as has already been seen in previous examples for the synchronous methods the caller is put into a wait loop until the call to the asynchronous method arrives. This is demonstrated in Figure 46, which shows the important parts of the translation of the code in Figure 45. The caller of the asynchronous method will have an immediate return as soon as the call is registered in the pattern matcher. In the *y* call, the parameter is stored in an object array and passed to the pattern matcher. The pattern matcher then stores this argument until a pattern is completed that involves the *y* method. The other Join fragment *x* when it is called will block in the while loop if there is no *y* waiting. When a *x* and *y* call are both waiting the *returnStruct* for *x* will become ready and contain the completed pattern number (due to the pattern matcher) and the parameter from *y*. This *returnStruct* is then passed to the dispatch method that then calls the method body. The arguments of the Join fragments are retrieved from the *returnStruct* via calls to *getObjectArg*. The translator then casts the objects to the original reference types.

```
void y(String val) {
   if (all$ == null)
      initJoin$();
   join.system.returnStruct retval$ =
        all$.addCall(new java.lang.Object[]{val}, 1);
   dispatch$(retval$);
   notifyJoin$();
}

String x() {
   if (all$ == null)
      initJoin$();
   join.system.returnStruct retval$ =
        all$.addSynchCall(new java.lang.Object[]{}, 0, this);
   while (retval$.ready != true) {
      try {
         synchronized (all$) {
            all$.wait();
         }
      }  catch (java.lang.InterruptedException ex) {
         throw new
           join.system.joinTranslatorException("…");
      }
   }
   return (String)dispatch$(retval$);
}

String x_$_y(String val) {
   System.out.println("Example3" + val);
   return val;
}

//From dispatch method
case 0:
   return x_$_y((java.lang.String)retStruct.getObjectArg(1, 0));
```

Figure 46. Synchronous Join Java Translated Code

4.3.2.4.4 Base Type Parameter Example

Whilst reference types in the previous section can all be stored as an *Object* irrelevant of the specific type (as they are all sub-classes of *Object*) base types pose an interesting problem in translation. There is no way of referring to all the different base types as a single type. Consequently, in the translation, boxing of base types must be done otherwise each type must be separately handled throughout the code. This exercise unfortunately becomes expensive in runtime if there is no support for boxing at the byte code level. Boxing requires boxing objects to be created for every base type. This defeats the reason for having base types (an efficiency concern) in the language (Gosling and McGilton 1996). In a final version of the compiler, it would be possible to implement separate code for each of the base types handling each one separately. This is not wise in a prototyping version though as one would need to change seven different segments of code for every minor change to the extension specification. In Figure 47,

an example Join Java program is shown that makes use of base types for both a parameter and a return type. Figure 48 shows the relevant code that is different between the base type versions and previous code examples. You can see when the method is called the integer argument is boxed in the Java wrapper class for storage in the pattern matcher. The boxed Integer is then treated just like the Objects in the previous examples. For the return value when the dispatch method is executed the return value is boxed in the wrapper class and returned back to the caller of dispatch. The caller then unboxes the return value and returns it to the caller of the synchronous Join fragment.

```
ordered class Example4 {
   int x() & y(int val) {
      System.out.println("Example3");
      return val;
   }
}
```

Figure 47. Base Type Join Java Source Code

4.3.2.4.5 Shared Fragment Example

The last example in the translation part of this chapter to be covered is the repeated fragment translation. In this example (presented below in Figure 49) the Join fragment *y(<int>)* is used in two Join methods. Consequently, it is interesting to see how this affects the translation. This is shown in Figure 49.

In the example, the translated methods are not shown, as the methods are identical to the previous examples. The interesting feature of this translation is in the *initJoin* method and the *dispatch* method. In the initJoin method in Figure 50 the situation where there are three Join fragments *x, y(<int>)* and *z*. They are identified as 0, 1 and 2 respectively. In the *addPattern* method calls you see the first Join method *x() & y(<int>)* is entered as 0,1. The second pattern *z() & y(<int>)* is entered as 2,1. This shows a situation where a Join fragment is used within two Join methods. The second interesting translation is the dispatch method where the dispatch method now has two possible Join methods to call. It can be seen that there is a choice between which two method bodies to execute depending on what Join method is chosen by the pattern matcher.

```
   void y(int val) {
      if (all$ == null)
         initJoin$();
      join.system.returnStruct retval$ =
         all$.addCall(new java.lang.Object[]{
                           new java.lang.Integer(val)}, 1);
      dispatch$(retval$);
      notifyJoin$();
   }

   int z() {
      if (all$ == null)
         initJoin$();
      join.system.returnStruct retval$ =
           all$.addSynchCall(new java.lang.Object[]{}, 2, this);
      while (retval$.ready != true) {
         try {
            synchronized (all$) {
               all$.wait();
            }
         }  catch (java.lang.InterruptedException ex) {
            throw new join.system.joinTranslatorException("…");
         }
      }
      return ((java.lang.Integer)dispatch$(retval$)).intValue();
   }

   int z_$_y(int val) {
      System.out.println("P1");
      return val;
   }

   //from dispatch method
    case 0:
        return new
           java.lang.Integer(x_$_y(retStruct.getIntArg(1, 0)));
```

Figure 48. Base Type Join Java Translated Code

```
class Example5 {
        int x() & y(int val) {
                System.out.println("P0");
                return val;
        }
        int z() & y(int val) {
                System.out.println("P1");
                return val;
        }
}
```

Figure 49. Repeated Join Fragment Usage Source Code

```
synchronized void initJoin$() {
   if (all$ != null)
      return;
   join.system.joinPatterns alllocal$ =
        new join.system.joinPatterns(this);
   alllocal$.setOrdered(false);
   alllocal$.addPattern(new int[]{0, 1}, true);
   alllocal$.addPattern(new int[]{2, 1}, true);
   alllocal$.noMorePatterns();
   all$ = alllocal$;
}
public java.lang.Object dispatch$(final returnStruct retStruct)
         throws join.system.joinTranslatorException {
   switch (retStruct.patternCompleted) {
      case 0:
         return new java.lang.Integer(
                x_$_y(retStruct.getIntArg(1, 0)));
      case 1:
         return new java.lang.Integer(
                z_$_y(retStruct.getIntArg(1, 0)));
      case -1:
         break;/* inserted for safety */
      default:
         throw new join.system.joinTranslatorException("…");
   }
   return null;
}
```

Figure 50. Repeated Join Fragment Usage Translated Code

4.3.2.5 Silent Semantic Analyses

After the translation phase is complete, an incomplete attribution of the abstract syntax tree is available. A silent semantic analyser is required to finish attribution of the tree. At the end of the silent semantic phase, the abstract syntax tree will contain a valid Java program. In the base compiler, the semantic analyser is the last phase of the front end. The extensible compiler will run a silent semantic analyser over the Java abstract syntax tree using the base language rules. This will make sure the translation from the extension language to the base language is correct. It is also necessary in this phase to attribute the newly translated sections of the abstract syntax tree. Once this phase is complete, the abstract syntax tree will contain a semantically correct Java attributed abstract syntax tree.

4.4 Pattern Matcher

The second major component of the Join Java extension is the pattern matcher. The objective of the pattern matcher is to implement the synchronization and dynamic channel formation semantics of the Join calculus in the language. The pattern matcher forms the core of the runtime system. The pattern matcher is used at runtime to decide which Join calls are matched together to execute Join method bodies. The pattern matcher is in the form of a library that integrates closely to the compiled code from the Join Java compiler.

4.4.1 Application Programmer Interface for the Pattern Matcher

In this section, the interface between the compiler and its run time system is described. The Join Java extension for the compiler resolves to Java bytecode but this is not sufficient to support the execution of Join Java programs. The language in fact makes use of a runtime system that handles the decision of what Join methods to execute at runtime. There are a number of different ways one could have approached to implement the pattern matcher. One option, extending the Java virtual machine, with new bytecode to support Join method matching. However, this means a special Java virtual machine(Gagnon 2002) would have had to be created. The second option was to generate pattern-matching code within the Join Java compiler. Whilst this is a superior implementation for performance and compactness of code, this method was not used because it slows the prototyping of alternate compilation strategies. Rather a runtime library was written in standard Java. This allows the pattern matcher to be easily changed for experimentation. Another design decision that needed to be made is whether to use a threaded or unthreaded model. With an unthreaded model for the pattern matcher, the processing of the arriving fragment is done in the runtime of the caller. If the pattern matcher were a separate thread it would have to be notified immediately, when any fragment was called. There is no guarantee that a further thread would not pre-empt the pattern matcher, delaying fulfilment of the completed pattern. Therefore, for simplicity the non-threaded event driven approach was selected.

The pattern matcher has two phases. In the initialization phase, the compiler passes a list of all the patterns to the pattern matcher. It also passes behaviour requests to the pattern matcher (see **ordered** modifier). When the compiler has completed passing patterns, it then signals the pattern matcher that it has finished setting patterns. The second phase, the operational phase accepts Join fragment calls and evaluates if there are any completed Join methods.

Each Join fragment call provides only partial information for the construction of a channel. It is the roll of the pattern matcher to assemble the fragments to form completed Join methods. The pattern matcher itself does not perform any operations on the data at the inputs of the channels. The pattern matcher is completely abstracted from the rest of the program. It simply acts as a mechanism for matching Join fragments to Join methods. It does not contain anything other than the identities of the fragments, the patterns and a mechanism to associate waiting arguments to the call. Thus, the pattern matcher can be viewed as an event based scheduler responding to the arrival of fragments/events associated with the Join methods. Another way of viewing the pattern matcher is a mechanism for marshalling the parameters required for the body of each Join method.

When the pattern matcher is asked to process a new Join fragment it is clear that there may be zero, one or more possible patterns that will be completed by the fragment (see Figure 49). For example, *X* and *Z* Join fragments have been called and a call to the *Y* Join fragment arrives for the following patterns *X&Y* and *Z&Y*. At the high level, there is no guidance at what the pattern matcher should do in this situation. Two possibilities here you either define the priority of matching, for example the order the Join methods were defined in the class or patterns with shorter lists of fragments are selected first. Alternatively, the pattern matcher may make a non-deterministic choice.

The pattern matcher implements a search operation. Therefore, it will always take some finite time to find a match. The speed of the search is dependent on the number of patterns and the number of fragments in each pattern. Some optimizations such as branch and bound pruning can possibly reduce this search time. The speed is also dependent on the data-structures and implementation of the algorithms. An important practical characteristic of the pattern matcher is graceful degradation of performance. That is, the pattern matcher performance should degrade proportionally to the number of patterns and fragments. However, some optimizations tend to reduce this predictability. It is important that the pattern matcher take the same amount of time for each pattern match otherwise, program performance will be erratic. This will then lead to programs that have seemingly random delays. As will be explained later, the linear search time implementations usually imply a fixed limit on the number and complexity of patterns that can be registered due to the processing overhead. The need for the pattern matcher to maintain state information about the callers of the blocking methods makes the search problem somewhat unusual requiring an especially constructed solution.

In the remainder of this section firstly, the API for the pattern matcher is described; this is the method calls and data structures expected by the translated Join Java code. Finally, different implementations of the pattern matcher are described.

4.4.1.1 joinMatcher class

This runtime library is encapsulated in a class called *joinMatcher* that provides a number of methods for the compiled Join class to call at runtime. This handles the pattern matching semantics between Join methods. A typical method is *addPattern*, which defines a specific Join method to the pattern matcher. The present version of the compiler creates one pattern matcher per instance of a class that contains Join methods.

Table 5 shows a summary of the most important methods and where they relate to the process of pattern matching. In the table, methods from both the initialization and operation phases are shown. The most important method call of the initialization phase is the *addPattern* method. This method takes a list of integer values that represent the identities of the fragments. Once all the methods are added, the *noMorePatterns* lets the pattern matcher know that it can start optimizing for the specific combination of patterns.

In the second phase (the operation phase), there are two methods, *addCall* which is called by asynchronous Join fragments and *addSyncCall* which is called by synchronous Join fragments. Inside the pattern matcher, these methods store the parameters for the current call then check with the pattern matcher to see if any Join methods are now complete. If a Join method is complete then the appropriate return structures are modified. That is each return structure of each fragment of the completed Join method is modified. All of these return structures are set to be ready. The completed pattern is set to -1 on all the return structures except for one, which is called the primary return structure. The primary return structure is set to the correct completed Join method identity and the parameters from all the return structures are moved into it. The choice of which fragment to use as the primary return structure depends on whether the Join method is synchronous or not. If the completed Join method is synchronous, the primary return structure is the first fragment of the pattern that is the fragment that is in likelihood blocking its caller. If the completed Join method is asynchronous then the last call that resulted in the completion is used. The -1's on the other lead to the dispatch methods aborting in all the other fragment cases.

Summary of Pattern Matcher (joinPatterns)	
Initialization Phase	
constructor	*Argument: JoinInterface* Creates the pattern matcher
setOrdered	*Arguments: boolean true or false depending on the reduction policy* Tells the pattern matcher whether to use non-deterministic reduction or to use the order of the definitions as the deciding factor.
addPattern	*Arguments: int array containing fragment identities* *boolean true or false representing synchronous status* Add a Join method signature into the pattern matcher. The first argument is a list of fragment identities. The second argument contains a boolean flag indicating the synchronicity of the Join method.
noMorePatterns	*Arguments: None* This method simply tells the pattern matcher that all the Join method signatures have been entered. The Join pattern matcher can then finalize its internal data structures. This is also the time that the pattern matcher can optimize.
Operational Phase	
addCall	*Arguments:Object array containing fragment arguments* *int fragment the fragment identity* At runtime whenever an asynchronous Join fragment call occurs this method is called. The first argument transports the boxed versions of the parameters the second parameter contains the identity of the fragment. This method must be synchronized.
addSynchCall	*Arguments: Object array containing fragment arguments* *int fragment the fragment identity that caused the call* At runtime whenever an synchronous Join fragment call occurs this method is called. The first argument transports the boxed versions of the parameters the second parameter contains the identity of the fragment. This method must be synchronized.

Table 5. Pattern Matcher API Summary

4.4.1.2 JoinInterface Interface

The Join interface is implemented by all Join classes in order to allow a call back from the pattern matcher to the Join Java class. This allows the pattern matcher to reawaken a waiting thread when an asynchronous Join method is complete. This interface is illustrated in Table 6.

Summary of Join Interface (JoinInterface)	
notifyJoin$	Method will make a synchronized call to *notifyAll* in order to wake up any threads waiting on the Join Classes monitor.

Table 6. Join Interface API Summary

4.4.1.3 returnStruct class

The *returnStruct* class acts as the storage mechanism for arguments along with a mechanism of returning the status of the fragment to the Join class. The class is illustrated in Table 7. The return structure consists mostly of unboxing methods that unwrap the arguments from their boxed versions stored in the argument lists. The dispatch method makes use of the *patternCompleted* field to figure out which Join method has been completed. The *patternCompleted* field can have a number of values including -1 indicating a Join method has been completed but is handled elsewhere or 0 indicating no completion and finally a positive integer representing the completed pattern. The ready field is used by the translated method in the Join Java class's wait loop.

Summary of Return Structure (returnStruct)	
patternCompleted (field)	Stores the identity of the completed pattern. Stores -1 if the pattern is completed by another return structure. Stores 0 while there is no pattern completed.
ready (field)	Stores the status of the fragment. If the fragment is part of a completed pattern it will become true.
Unboxing Methods getIntArg,getBooleanArg, getByteArg,getShortArg, getLongArg,getFloatArg, getDoubleArg,	*Arguments: int argument number.* Unboxing methods for the base types and the reference types. The parameter tells the return structure which parameter to return.

Table 7. Return Structure API Summary

4.4.1.4 JoinTranslatedException class

The final public class in the Join Java pattern matcher package is the Join translator exception class. This is used internally by the compiler to indicate translation and pattern matching errors. Exceptions can then be caught either inside the translated code or inside the pattern matcher for debugging.

4.4.2 Approaches to Pattern Matching

There are a number of approaches that can be taken in constructing a pattern matcher to achieve the semantics of the Join calculus. The simple approach to building a Join Java pattern matcher is to record the status of the waiting calls followed by a list of all possible reductions that are compared against the current state of the waiting calls. However, this approach breaks down, as search is quadratic on every call to the pattern matcher. The original Join language (Maranget and Fessant 1998) used a state machine where states are used to represent the possible patterns. However, state space explosion was a problem and they used state space pruning and heuristics

to trim the state space down. The second version of their language used a bit field to represent the status of the pattern matching. Each pattern reduction was compared with the current state of the calls via an XOR call. Whilst these approaches sped up pattern matching the solutions were not scalable beyond the predefined maximum size of the bit-field. The state space implementation consumed a lot of memory and the bit field solution was limited on the upper end by the max number of digits that could be represented and hence Join fragments the type could store. Two alternative ways of describing the data-structure are described below.

- Representative/ Structural. (bitmaps). A representative/structural solution usually is a direct encoding of the problem such as the naive solution presented previously or the bitmap solution from Maranget.

- State Space. (state machine). State space solutions try to represent all possible situations the pattern can be in and then navigate between them when Join fragment calls occur.

4.4.2.1 Tree Structured Implementation

In the pattern matcher for Join Java, a middle ground between the space complexity of a state-based solution and the time complexity of a linear solution was sort. One approach was by using a tree structure to represent the static structure of the patterns of a Join class. The idea of the approach is to limit the search space during the runtime of the program. The data-structure is consequently designed with the idea of a localized search in mind. In the data-structure, interior nodes represent patterns and leaves represent Join fragments. The root acts as an index to both leaves and interior nodes for fast access. In Figure 51, an example with three Join methods and six fragments is shown. The most interesting fragment is **B** as it is shared by two patterns **A&B** and **B&C&D**. This design allows us to trade the space against matching time. However, the search time is further optimized by only checking the part of the tree that is directly affected by the occurrence of the Join method fragment call. Using the example from the figure when a **C** is called only **B&C&D** is checked for completion. If **B** is called both **A&B** and **B&C&D** are checked for completions. This is achieved by connecting the leaves to multiple interior nodes so that when a fragment arrives it is quick to check if that new fragment completes any Join methods. In the pattern matcher a list of all fragments are stored in the root of the node so that when a call arrives the correct location in the tree can be immediately accessed without the need to traverse the data-structure. In this way, the pattern matching process can be optimized to search only the part of the tree that contains patterns affected by the

arrival of the new fragment. That is if a Join method call occurs it only checks patterns that contain that Join method fragment.

The Join calculus has a non-deterministic matching semantic on reduction of rules. However as related earlier, in the pattern matcher the semantics to support deterministic reduction have been extended. This is done via the **ordered** modifier. When the pattern matcher is in deterministic reduction mode it will match all possible patterns in the pool rather than the first randomly selected match. The pattern matcher will then choose which pattern to complete based upon the order in which they were defined in the Join class. The worst-case scenario for this pattern matcher is if a Join fragment occurs in every Join method. This will lead to every pattern being searched. This would make the algorithm $O(m^k)$ complexity (where m is the number of Join methods and k is the number of other Join fragments involved in the methods). This is not likely to happen as normally Join method fragments have locality. In other words, most Join fragments only take part in a few Join methods.

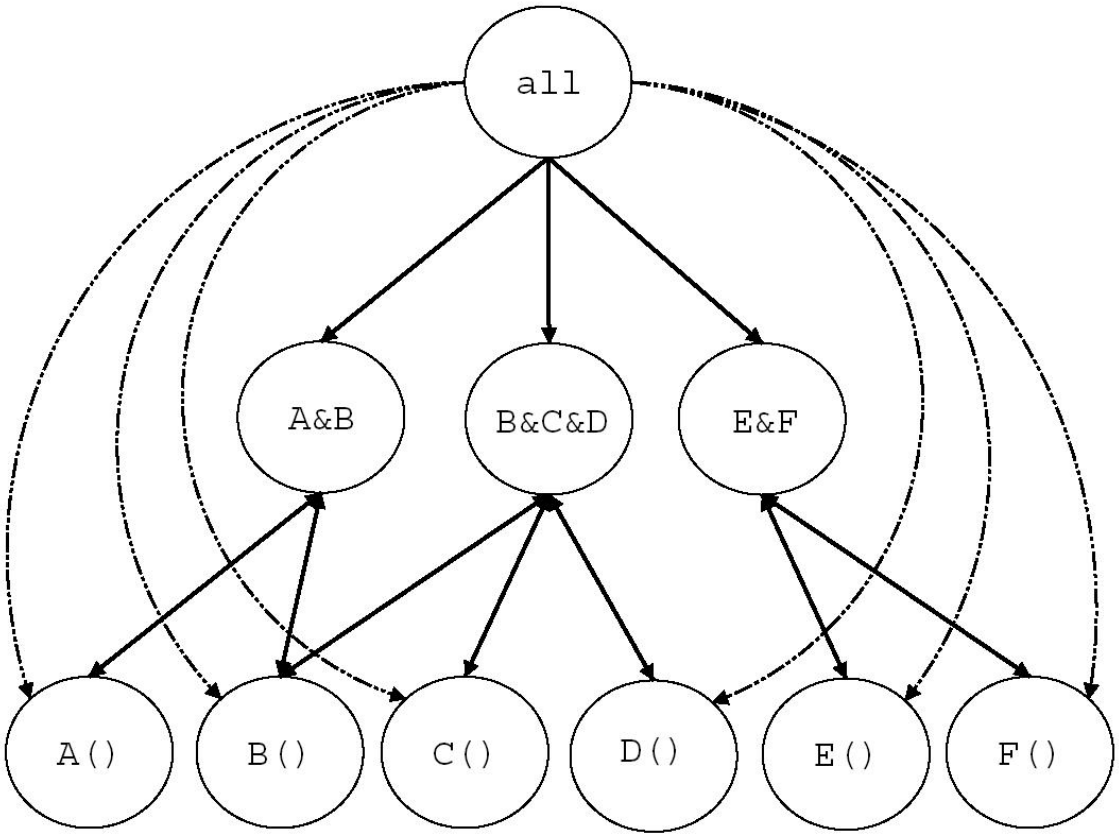

Figure 51. Internal Representation of Tree Pattern Matcher

4.4.2.2 Pre-calculated Table Lookup Implementation

The second major pattern matcher that was developed was designed to optimize the speed of execution for a limited number of fragments. This pattern matcher pre-calculated every possible state that the pattern matcher could exist in and in the event of a change in the state space would immediately know the appropriate pattern to execute. The state of the pattern matcher is expressed as a series of bits used to represent an integer value. This integer value gives a position in the pre-calculated array that resolves to a completed pattern. The array is thus expressed, as a linear array with a magnitude of 2^n where n is the number of fragments in the Join class. The state of the pattern matcher at any point in time can be expressed as a sequence of bits indicating the presence or absence of a particular fragment. For example, a Join class containing five fragments (**a** through **e**), there are 32 possible states *00000* through to *11111*. If there was an **a** and a **c** waiting the bit-field would be *10100*. The design of the pre-calculated pattern matcher is illustrated in Figure 52. In the event that more than one fragment

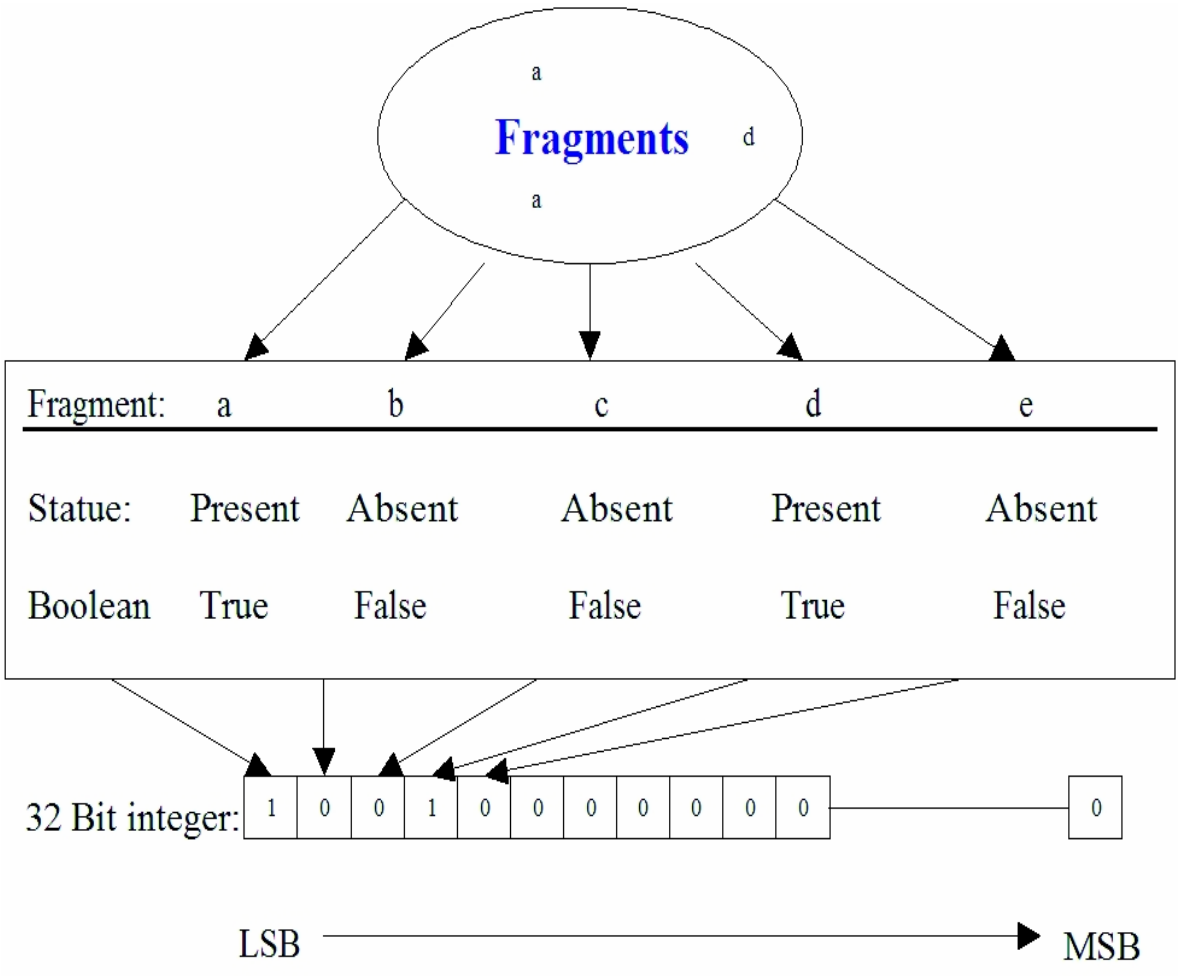

Figure 52. Pre-calculated Pattern Matcher

is waiting the bit field will still only be 1. Therefore 1 represents some (one or more) fragments waiting in the pattern matcher and 0 means no fragments waiting in the pattern matcher. Because the state can be converted into an index via trivial bit manipulation, retrieval of completed patterns is performed quickly. When initialization occurs (in the prototype when the first fragment is called), the pattern matcher calculates all possible states the pattern matcher could be in and calculates the resultant completed patterns from those states.

The major advantage of this approach is that the pattern matcher design has a O(1) delay when retrieving completed patterns. This is because after the pre-calculation of states is done no searching needs to be performed during runtime. The state of the pattern matcher is stored as a position in the pre-calculated array. Consequently, the time fluctuations other matching algorithms suffer from are removed.

This pattern matcher has two disadvantages. Firstly, In the event of a large number of fragments the pre-calculation period takes an increasingly long time to populate the array. However, if this is done at compile time there will be no penalty. The second disadvantage is the memory footprint of the pattern matcher is relatively large compared to other implementations. The number of fragments that can be handled by the pattern matcher is limited by the memory that can be allocated to the pre-calculated array. Benchmarking (see section 7.2) of the pattern matcher has indicated that performance is adequate in the majority of cases. Various optimizations could be created to improve the overheads of this pattern matcher design. For example, some form of caching strategy could be used to improve the pattern matching speed.

4.4.2.3 Optimizations

The previous section has described the two new approaches to implementation of a Join pattern matcher with their limitations. It is possible to reduce the penalty paid for the particular implementation by making optimizations that can either improve performance or improve memory usage.

Using Symmetry in Representative Pattern Matching

To improve the capacity of the representative and state space solutions a number of heuristics are possible. For example in the bitmap solutions, it would be possible to improve the (Maranget and Fessant 1998) approach by using symmetry to reduce the size and number of patterns. This could be achieved by making generic version of some patterns if they look

similar to each other. For example in Figure 53 two patterns reduce to expression1 and expression2

```
A&B -> e1
A&C -> e2
```

Figure 53. Symmetry before Example

Both these reductions are similar patterns the only difference being that the call to *C* or call to *B* defines which reduction is executed in the end. A new pattern fragment call *X* could be introduced which means either *B* or *C* and then generate the reduction in Figure 54. A decision (linearly) which expression to execute after the pattern match is complete must be then be made. Of course, this solution only reduces the size of the problem but does not eliminate it.

```
A&X -> e1 | e2
```

Figure 54. Symmetry after Example

State Space Pruning

Another approach similar to the symmetry optimization is the idea of pruning the state space. In the previous section, some state based solutions were examined. The state based solutions are straightforward methods of representing the pattern-matching problem however; these approaches are limited because of the problems involved in state space explosion. These problems could be reduced by using storage heuristics. For example in the (Maranget and Fessant 1998) state based solution, the author reduces state space explosion by collapsing some of the state space into states that represent several possible states. This reduces the memory footprint the state space has fewer nodes to search and hence increases the size of the real search space for little cost.

Limited Search in a State Space.

In the tree based Join Java pattern matcher a representational structure for the patterns is used. That is state is not maintained for every possible situation instead a representation is maintained for possible complete patterns. In this way, the solution space is small. Unfortunately, whenever a method call occurs it needs to check for the occurrence of a completed pattern. State space explosion is traded for a slightly longer matching time. However, representative structure can be further optimized by only checking the part of the structure that is directly affected by the occurrence of the Join method fragment call. That is if a Join method call occurs it only checks patterns that contain the Join method fragment.

4.4.3 Summary

In every implementation based on the Join calculus the critical factor to the performance of the language will be the pattern matcher. Whilst designing the various pattern matchers it was concluded that no single pattern matcher could ever efficiently solve all possible configurations of patterns and fragments. To this end, considerable effort was spent on this component of the compiler looking for novel solutions in order to increase the speed without compromising scalability, speed or memory size. This has proved challenging but it was felt that the prototype pattern matchers were a good start in this direction.

4.5 Issues Identified in the Prototype

When creating this extension a number of non-intuitive issues were encountered. The two main issues were firstly, implementation of multi-directional channels and secondly, the association of locks to parameters. These two issues are now described in more detail.

4.5.1 Multi-Directional Channels

The construction of the Join calculus implies the possibility of multi-directional channels. That is a messages can be passed two ways. An example of what a multi-directional program would look like is supplied in Figure 55. It can be seen that in this program the Join method has more than one synchronous Join fragment. This means that more than one method will be blocked waiting on a successful completion of a Join method. When there are calls to both A(int x) and B(int y) the Join method executes the Join method body returning the parameter y to A and x to B. This means that parameter y is passed from the caller of B to the caller of A. Conversely the parameter x is passed from the caller of A to the caller of B. This channel formation would make some of the more awkward pattern implementations (see Chapter 5) a lot more succinct.

Whilst it is possible to implement a multi-directional channel construct there are a large number of issues that become apparent. Firstly, the pattern matcher becomes a lot more complicated and hence slower as it needs to record more than one returnee for each pattern. The synchronous callers must also be blocked and one allowed to proceed into the method body. If all the threads are let into the method body a number or synchronization problems would occur. For example, the decision of which caller to actually do the processing needs to be decided. Presumably this would be the last caller to depart the Join method body. As each return statement is reached the runtime would have to pass the return value to the named blocked synchronous method before allowing it to continue. Secondly, the cognitive load to the programmer is higher as they now have to deal with multiple exit points for different threads in

```
final class MultiDirectionalChannelExample {
   int A(int x) & int B(int y) {
      //code that callers to A() and B() want to execute

      return y to A(int);//this returns control to A();

      //some more code that the caller to B() want to execute
      return x to B(int);//this returns control to B();
   }
}
```

Figure 55. Example Multi-Directional Program

a method. It is possible to simulate multi-directional with uni-directional channels, which reduces the necessity to implement them. For these reasons, multi-directional channels were not implemented.

4.5.2 Lock Parameter Association

A second subtle issue when implementing the Join Java extension is the association of locks with parameters. When a synchronous caller is blocked the parameters it passes via the Join fragment call must take part in the Join method body. An example program is shown in Figure 56. The program shows a situation where two threads (thread1 and thread2) are calling a Join fragment that passes in a single integer value (*param*). The value is returned to the caller as soon as a call to B is randomly generated by the thread3 method. If locks were not associated to

```
final class LockAssociationExample {
   signal thread1(){
      System.out.println("result =  "+A(3));
   }

   signal thread2() {
      System.out.println("result =  "+A(6));
   }

   int A(int param) & B() {
       return param;
   }

   signal thread3() {
       //a loop that randomly emits calls to B();
   }
}
```

Figure 56. Lock Parameter Association Example Code

the parameters in this example it would be possible that thread1 might print a 6 as the wrong lock is released. This issue is only a problem if a synchronous method has parameters and a return type other than **void**.

4.6 Conclusion

This chapter demonstrated the feasibility of implementing the Join Java extension in Java. Whilst investigating the implementation of the prototype language a number of interesting discoveries were made. Each point is covered below;

1. Even though the Join calculus implies multi-directional channels, they were not implemented due to two factors. Firstly, the difficulty of implementing the **return-to** construct from the calculus. This would involve implementing a method of generating some form of asynchronous return in which different callers could depart the method body at different times. Secondly, there is also the problem that programmers would not be used to the idea of multiple exits from a method body by different fragments. This is described in section 4.5.1.

2. In Join Java, it was necessary to allow base types to be passed as parameters for Join fragments. Unfortunately, by passing the parameters it was found that the pattern matcher had to store a number of different types. This also meant that processing in the translated code for the parameters must handle all the base types plus the object type. The only practical solution to this problem that maintains the prototypes ability to be modified easily was to use a boxing technique. Boxing means that when the parameters are passed into the pattern matcher they are wrapped in a holder class that in turn is manipulated by the pattern matcher parameter storage mechanism. This allows the same code to be used to manipulate all the base types and the object type. The unfortunate side affect of this approach is to slow down the method calls for methods containing base types due to the boxing and unboxing operations. If the Java virtual machine supported boxing and unboxing like that of later languages[11] this overhead could be minimized.

3. An important issue with pattern matching is ensuring that the arguments of a synchronous Join fragment are used in the method body that returns to the Join fragments caller. That is in the pattern matcher, arguments need to be associated with synchronous fragments locks. When a parameterized synchronous Join method call occurs, the caller is blocked until the Join method fragment completes. When a completion occurs, the parameters must be passed into the body of the Join method. If

[11] Microsoft's .Net framework provides optimizations for boxing and unboxing.

that Join method then returns a value then the value must be returned to the caller that was blocked. If this association of lock to the parameters of the Join method calls is not stored, results that do not relate to the parameters could be returned. This simple requirement complicated both the pattern matcher and the compiler implementations. This was described in more detail in section 4.5.2.

4. A choice that was faced writing the compiler was whether to use a generated pattern matcher or a generic pre-written pattern matcher. A generated pattern matcher will be faster as it can be optimized to the particular patterns in the Join class. However, the compiler itself will be more difficult to extend or modify. If a generic pattern matcher was used that is the same for all Join Java classes the pattern matcher could be changed at will to experiment with different optimizations. It also allows the compiler to be modified more easily.

This chapter has explored the implementation of the Join Java language extension. In the next chapter, a range of typical concurrent programming examples coded in Join Java language are provided.

5

Design Patterns

There are two ways of constructing a software design. One way is to make it so simple that there are obviously no deficiencies. And the other way is to make it so complicated that there are no obvious deficiencies.

(Charles Hoare)

Table of Contents

5.1 Introduction

In this chapter, the Join Java implementations of a number of common patterns in concurrency are examined. The first section describes design patterns relating to synchronization such as scoped locking, strategized locking, thread safe interfaces and double check locking optimization. The second section examines patterns that deal more with representation of concurrency such as, active objects, monitor objects, half-sync/half-async, leader/follower and thread specific storage. The final section examines a number of simple concurrency mechanisms that are commonly seen in concurrent applications, for example semaphores, timeouts, channels, producer/consumer, bounded buffer, reader writers and thread pools. Each of these patterns is implemented in both Join Java and Java. For each pattern, each approach is explained and there is some discussion on how they differ. It is necessary to show that Join Java is complete enough to express all patterns. However, in very few cases the Join Java expression of the pattern was shown to be longer than the Java solution. The implementations in this chapter will form a basis of comparison of both execution time and code length for Chapter 7.

5.2 Patterns for Synchronization

In this section all the design patterns for synchronization presented in (Schmidt 2000) are examined[12]. All the patterns presented in chapter four of the text are covered. In a practical sense, not all patterns presented here should be used in a real world situation. For instance, the monitor locking pattern is better implemented using the low-level monitor operations of Java. However, to prove the expressiveness of Join Java all the concurrency patterns are converted to Join Java.

The first pattern covered in this section is scoped locking. This pattern closely couples the locking mechanism to the scoping of a section of code allowing automatic acquisition and relinquishment of the locks. The second pattern is strategized locking that parameterises the locking mechanism in order to allow the manipulation of synchronization strategies in a first class manner. The third pattern is the thread-safe interface, which acts as a wrapper to unsynchronized code providing a totally separate mechanism for synchronization to that of the execution of the code. The final pattern covered in this section is the double-checked locking optimization which introduces a precheck in order to avoid repeated checking of locks with their related slowdowns. It should be noted that some patterns are not necessary in this language as the behaviour they present to the programmer are already integral to the Join Java language. For example the leader/followers pattern is implicit in the structure of the language as waiting Join method calls can represent waiting threads. The equivalent code in Join Java is shown for completeness and to show that the language is expressive enough to represent all the patterns presented by Schmidt (op cit).

5.2.1 Scoped Locking

The purpose of the scoped locking idiom is to make sure a critical section or method acquires a lock on entry and releases the lock on exit. The method of doing this is to create a class in which the constructor switches a lock on and the classes' destructor switches the lock off. A primary disadvantage of Java is that there is no real destructor method in the language due to the garbage collector. Java does provide finalizer methods. However, there is no guarantee that these finalizers will be executed when an object leaves scope. Consequently, the implementation of this pattern is impossible to achieve in a reliable manner.

[12] The thread specific storage pattern was not implemented because it avoids the problem of locking by duplicating data structures.

```
class Scoped {
        Scoped() {
                unlock();
        }
        synchronized void autoLockingMethod() {
                try {
                        lock();
                        System.out.println("Start Safe");
                        System.out.println("End Safe");
                } finally {
                        unlock();
                }
        }
        void lock() & unlock() {}
}
```

Figure 57. Join Java Scoped Locking Implementation

There is one work around to this problem as illustrated in Figure 57. Exception handling is used to ensure that a lock is released when a method body goes out of scope. This can be achieved using the finally clause for exceptions. Inside the finally clause (as you would normally do in the destructor) the lock is released. The Join method below simply acts as the locking mechanism. When the *autoLockingMethod* method calls the *lock* fragment it is blocked until an *unlock* fragment is called. In future when this pattern is used in future examples the finally code will be omitted for brevity.

The standard Java language provides this pattern via the modifiers of the method. The **synchronized** modifier acts to make the entire method synchronized (see Figure 58). Consequently, when this pattern is desired in a Java program its best to use the synchronized keyword due to its simplicity.

5.2.2 Strategized Locking

The strategized locking paradigm abstracts the synchronization mechanism from the code on which it operates. In different environments, different synchronization strategies may be needed. Consequently, what is needed is to replace the synchronization mechanism with a specific one for the purpose in mind. This implies the interface defines the Join fragments but

```
class Scoped {
  synchronized void autoLockingMethod() {
    //some protected code
  }
}
```

Figure 58. Java Scoped Locking Implementation

not the Join methods. Various implementations of the interface are then written. In this way, one can arbitrarily replace synchronization behaviours both at runtime and compile time. Figure 59 and Figure 60, show how to replace synchronization behaviours at runtime. The example is a simple traffic light system that changes its behaviour at night. In the daytime, the lights just alternate back and forward. At nighttime they use a sensor to detect when traffic is waiting on EW route and let them through else let NS traffic through. Firstly, the interface called *AbstractSynchronizer* is defined which describes two method headers *goNS* and *goEW*. Two implementation classes are then written with their respective behaviours. Finally, a traffic light control system is written which runs traffic lights using either the day or night behaviours. The control system has two threads one for each direction of traffic. Each thread takes a parameter that is the synchronizer. The most important part of the example to note is the main method where different synchronizers are being created dynamically. This is so that synchronization scheme can be changed at runtime altering the behaviour of the interaction between the two threads.

In Figure 61, Figure 62 and Figure 63 the equivalent Java code is presented. In the example, the closest possible code for comparison is generated. Consequently, the interface and the main method are identical to the Join Java version. The class *NightTimeSynchronizer* in the Java program is simpler than the Join Java one as you can make use of the object's monitor to simulate the sensor. However, the alternative implementation *DayTimeSynchronizer* which uses two locks does need to use additional checks and mechanisms to protect the locks. This becomes more complicated than the Join Java equivalent as the checking and setting must be implemented. An alternative implementation to this would be to use a single Boolean flag and alternate between each of the two lights. However, this code would be less readable and still does not solve the problem as it is scaled up to more lights in the intersection. This example shows that whilst the monitor approach is good for the simpler problems when it is scaled up to more complicated interactions the mechanism breaks down. This makes the code more prone to errors as it becomes more complicated and hard to read.

```
interface AbstractSynchronizer {
  public void goNS();
  public void goEW();
}
//Alternate cars through the intersection
class DayTimeSynchronizer implements AbstractSynchronizer {
  public DayTimeSynchronizer() {
    NS();
  }

  public void goNS() & NS() {
    System.out.println("Car Goes NS Daytime");
    //some delay
    EW();
  }
  public void goEW() & EW() {
    System.out.println("Car Goes EW Daytime");
    //some delay
    NS();
  }
}

//Allow cars from either direction if the hit the sensor (x)
ordered class NightTimeSynchronizer implements AbstractSynchronizer {
  public NightTimeSynchronizer() {
    x();
  }

  synchronized public void goEW()  & x() {
    System.out.println("Car Goes EW Nighttime");
    x();
  }

  synchronized public void goNS() {
    System.out.println("Car Goes NS Nighttime");
  }
}
```

Figure 59. Join Java Strategized Locking Library Code

```
class TrafficLights {
   signal t1(AbstractSynchronizer sync) {
     System.out.println("Starting EW Cars");
     for (int i=0;i<4;i++) {
          sync.goEW();
     }
     System.out.println("Ending EW Cars");
   }

   signal t2(AbstractSynchronizer sync) {
     System.out.println("Starting NS Cars");
     for (int i=0;i<4;i++) {
          sync.goNS();
     }
     System.out.println("Ending NS Cars");
   }

   public static void main(String[] argv) {
     System.out.println("Test");

     TrafficLights lights= new TrafficLights();

     AbstractSynchronizer day = new DayTimeSynchronizer();
     lights.t1(day);
     lights.t2(day);

     AbstractSynchronizer night = new NightTimeSynchronizer();
     lights.t1(night);
     lights.t2(night);

   }
}
```

Figure 60. Join Java Strategized Locking User Code

```
//this behaviour gives preference to the intersection with cars
//road only allowing
//traffic when the sensor (X) is tripped

class NightTimeSynchronizer implements AbstractSynchronizer {
   public NightTimeSynchronizer() {
   }

   //we can use the objects monitor as the sensor result
   synchronized public void goEW() {
     System.out.println("Car Goes EW Nighttime");
   }

   synchronized public void goNS() {
     System.out.println("Car Goes NS Nighttime");
   }
}
```

Figure 61. Java Strategized Locking Implementation Code

```
interface AbstractSynchronizer {
   public void goNS();
   public void goEW();
}

class DayTimeSynchronizer implements AbstractSynchronizer {
   boolean NS=false;
   boolean EW=false;
   public DayTimeSynchronizer() {
      NS=true;
   }

   public synchronized void goNS() {
      while (!NS) {
           try{
                wait();
           } catch(InterruptedException ex) {/*report error*/}
      }
      System.out.println("Car Goes NS Daytime");
      //some delay
      NS=false;
      EW=true;
      notify();
   }

   public synchronized void goEW()  {
      while(!EW) {
           try{
                wait();
           } catch(InterruptedException ex) {/*report error*/}
      }
      System.out.println("Car Goes EW Daytime");
      //some delay
      EW=false;
      NS=true;
      notify();
   }
}
```

Figure 62. Java Strategized Locking Implementation Code

```
class TrafficLights {
   void t1(final AbstractSynchronizer sync) {
      (new Thread() {
           public void run() {
                System.out.println("Starting EW Cars");
                for (int i=0;i<4;i++) {
                     sync.goEW();
                }
                System.out.println("Ending EW Cars");
           }
      }).start();
   }
   void t2(final AbstractSynchronizer sync) {
      (new Thread() {
           public void run() {
                System.out.println("Starting NS Cars");
                for (int i=0;i<4;i++) {
                     sync.goNS();
                }
                System.out.println("Ending NS Cars");
           }
      }).start();
   }
   public static void main(String[] argv) {
      System.out.println("Test");

      TrafficLights lights= new TrafficLights();

      AbstractSynchronizer day = new DayTimeSynchronizer();
      lights.t1(day);
      lights.t2(day);

      AbstractSynchronizer night = new NightTimeSynchronizer();
      lights.t1(night);
      lights.t2(night);

   }
}
```

Figure 63. Java Strategized Locking Use Code

5.2.3 Thread Safe Interfaces

Thread safe interfaces try to minimize locking overhead and intra-class method call dead locking by providing boundary accessor methods that contain locking. No internal methods check locks because they trust that the boundary methods have ensured locking is carried out. Figure 64 shows an example of a class that implements thread safe interfaces. The class has two methods, the first method *accessData1Unsafe* returns the sum of the two object variables *someData* and *someOtherData*. The second method *accessData2Unsafe* simple returns the value *someOtherData*. These two methods possess no locking semantics but are made private. Two additional methods (boundary methods) are provided, *accessData1* and *accessData2* these methods acquire locks (block if not able to acquire) call the unprotected private methods and then release the lock. The actual locking mechanism is similar to the ones provided earlier.

```
class ThreadSafeInterface {
   private int someData=3;
   private int someOtherData=9;
   public ThreadSafeInterface() {
      unlock();
   }
   public int accessData1() {
      lock();
      int result = accessData1Unsafe();
      unlock();
      return result;
   }
   public int accessData2() {
      lock();
      int result = accessData2Unsafe();
      unlock();
      return result;
   }
   private void lock() & unlock() {}

   //Note that this method accesses another
   //method without having to check for
   //locking.
   private int accessData1Unsafe() {
      return accessData2Unsafe() + someData;
   }
   private int accessData2Unsafe() {
      return someOtherData;
   }
   //... other methods that may change the instance variables
}
```

Figure 64. Join Java Thread Safe Interface Code

The interesting method to examine is *accessData1Unsafe*, this method uses the other method *accessData2Unsafe* however, this call does not have the overhead of having to test and/or reacquire the lock.

Figure 65 presents the equivalent Java code for the thread safe interface. In the example the synchronized modifier is used to protect the inside methods of the object. As Java uses re-entrant monitors this pattern is less of an issue than in other languages that use less flexible synchronization mechanisms. However, there are still time penalties in Java as the JVM still has to reacquire the lock on re-entry. This means the pattern may still be useful in some circumstances. Figure 65 below gives an example of a thread safe interface implement using the synchronized keyword. Note that the call to *accessData1Unsafe* and the call to *accessData2Unsafe* are re-entrant on the monitor. This means you will still suffer the overhead of reacquiring the lock. A reasonable optimization of this code would be to remove the synchronized modifier from all the private methods as they are uncallable from outside anyway. The only requirement for this modification is that all public and protected methods must be synchronized so that they have a thread safe interface to the outside. Consequently, in these two

```
class ThreadSafeInterface {
   private int someData=3;
   private int someOtherData=9;
   public ThreadSafeInterface() {
   }
   public synchronized int accessData1() {
      int result = accessData1Unsafe();
      return result;
   }
   public synchronized int accessData2() {
      int result = accessData2Unsafe();
      return result;
   }
   private synchronized int accessData1Unsafe() {
      return accessData2Unsafe() + someData;
   }
   private synchronized int accessData2Unsafe() {
      return someOtherData;
   }
}
```

Figure 65. Java Thread Safe Interface Code

examples it can be seen that generally the Java solution is the more appropriate as the synchronized keywords do naturally what the Join Java solution is trying to emulate.

5.2.4 Double Check Locking Optimization

The double-checked locking optimization tries to avoid synchronization overhead wherever it is not necessary by doing an initial check before trying to acquire the more expensive lock. The example presented in Figure 66 simply checks a flag first (thus avoiding repeated checks of the lock). Initially when the class is instantiated the constructor makes a call to *unlock*. This call allows the next call to *lock* to execute immediately. There is also a method *initValue* that is supposed to initialize the variable *value* when first called. When an outside thread calls the *initValue* method a flag (*value*) is checked to see if initialization has been completed. If the flag has not been set, try to acquire the lock and then recheck the flag to make sure that another

```
ordered class DoubleLockingOpt {
   String value;
   public DoubleLockingOpt() {unlock();}
   public void initValue() {
      if (value==null) {
           lock();
                if (value==null) {
                     value = "value initialized";
                }
      }
   }
   void lock() & unlock() {}
}
```

Figure 66. Join Java Double Check Locking Optimization Code

thread has not pre-empted this thread. Finally, if the flag still has not been set, initialize the value. In the entire life of the object the synchronization check should only occur at most a few times and only then when a race condition happens between the first test and the lock call.

Unfortunately, even though this pattern initially looks like a good optimization, it cannot work in Java due to the design of the JVM. For optimization, the JVM can reorder instructions in the absence of a synchronized keyword. Even without this optimization, using multiple processors can lead to subtle problem in evaluation. This pattern in the presence of either an optimizing compiler or multiprocessor machine can bypass the lock making the pattern unreliable at best. For further information please consult (Bacon, Bloch et al. 2002). For completeness in Figure 67 an example of the pattern implemented in Java is given. This example (if it worked) is simpler in Java due to the synchronized block being effectively used. It should be noted that on

```
class DoubleLockingOpt {
   String value;
   public DoubleLockingOpt() {}
   public void initValue() {
      if (value==null) {
           synchronized(this) {
                if (value==null) {
                     value = "value initialized";
                }
           }
      }
   }
}
```

Figure 67. Java Double Check Locking Optimization Code

later versions of the Java virtual machine the use of synchronized block is extremely slow (Sun 2002) (see section 7.2) in comparison to the use of synchronized modified methods. Consequently, to improve performance the synchronized block should be removed and made a separate method. However, by doing this, one ends up with more complexity in the final program.

5.3 Patterns for Concurrency

In this section design patterns for concurrency (Schmidt 2000) are presented. The first two patterns, active objects and monitor objects are designed to facilitate sharing of resources. Active objects decouple method execution from method invocation. The monitor object pattern synchronizes method execution in order to restrict threads from running more than one synchronized method at one time. The second set of patterns is designed to facilitate higher-level concurrency synchronization mechanisms. Half sync/half Async pattern separates synchronous operations from asynchronous operations via a queuing mechanism. The leaders/followers provides an architecture which supports the idea of thread pools in which each thread has turns at dealing with incoming requests. The final pattern thread-specific storage seeks to minimize the complexity of synchronization by minimizing the sharing of data between threads.

5.3.1 Active Object

The idea of the Active Object pattern is to decouple method execution from method invocation. In Join Java, this is extremely simple; Figure 68 shows how by using a **signal** return type the effect is attained.

Figure 69 gives an example of the equivalent Java code. It can be noticed that the Java version of the above code is longer and more complicated. To implement the Active object pattern in Join Java you simply need to make a method with **signal** return type and call that from the constructor of the Object. In Java you are required to create an anonymous thread, set all the parameters (make them final) and then pass them into the anonymous inner class. Then call that from the constructor.

```
class ActiveObjectExample {
  public ActiveObjectExample(int x, int y, int z) {
     decoupledMethod(x,y,z);
  }
  private signal decoupledMethod(int x, int y, int z) {
     //do something
     //this method decouples the invocation from the execution
  }
}
```

Figure 68. Join Java Active Object Code

```
class ActiveObjectExample {
  public ActiveObjectExample(int x, int y, int z) {
    decoupledMethod(x,y,z);
  }
  private void decoupledMethod(final int x,final int y,final int z) {
    //this method decouples the invocation from the execution
    (new Thread() {
        public void run() {
            //do something
        }
    }).start();
  }
}
```

Figure 69. Java Active Object Code

5.3.2 Futures

A more rewarding examination of the Active Object pattern is the idea of futures (Chatterjee 1989) presented in Figure 70. When an asynchronous method is used, there must be some way of retrieving values from the asynchronous method. The mechanism for returning these values to the caller is a future. In Join Java, there is no specific futures mechanism however; the extension supports the idea of futures in a straightforward manner. Figure 70 shows an asynchronous method (emulating the idea of Active Objects) *doSomething* which is called with a String parameter. This method does some processing then calls the method *doneSomething* passing the completed data to a Join method *acquireFuture() & doneSomething.* This Join method will return the result to the caller of *acquireFuture* when that method is called. Consequently to use this future a method would call *doSomething* then return immediately do some other processing then to pickup the result for the processing call *acquireFuture* and read the returned value. This process is illustrated in the *main* method.

Figure 71 shows the equivalent Futures code written in Java. Even a cursory comparison of the code shows that the Join Java code is considerably shorter and more straightforward. The simplicity is due to a number of features in the language extension. Firstly thread creation is done by an asynchronous return type **signal** where as in Java one must create an anonymous inner class that extends thread. Secondly, the dynamic channel creation can act as the futures mechanism. In Java, the lack of these two features means that a low-level mechanism must be used to express the structure.

```
class FuturesExample {
   signal doSomething(String param) {
      //do something
      //then signal that result is available
      String result = "meaningful data";
      doneSomething(result);
   }
   String acquireFuture() & doneSomething(String res) {
      return res;
   }
   public static void main(String[] argv) {
      FuturesExample x = new FuturesExample();
      x.doSomething("some data");
      //do something else while waiting for processing
      //to complete
      String result = x.acquireFuture();
      System.out.println(result);
   }
}
```

Figure 70. Join Java Futures Code

```
class FuturesExample {
   private boolean finished=false;
   private String futureResult=null;
   final FuturesExample monitorHandle=this;

   public void doSomething(String param) {
      (new Thread() {
           public void run() {
                //do something
                //then signal that result is available
                monitorHandle.futureResult = "meaningful data";
                monitorHandle.finished=true;
                synchronized(monitorHandle) {
                     monitorHandle.notify();
                }
                // pass information doneSomething(result);
           }
      }).start();
   }

   public synchronized String acquireFuture() {
      while(!finished){
           try{
                wait();
           } catch(InterruptedException ex) {/*report error*/}
      }
      return futureResult;
   }

   public static void main(String[] argv) {
      FuturesExample x = new FuturesExample();
      x.doSomething("some data");
      //do something else while waiting for processing
      //to complete
      String result = x.acquireFuture();
      System.out.println(result);
   }
}
```

Figure 71. Java Futures Code

5.3.3 Monitor Object

The monitor object design pattern synchronizes concurrent method execution to ensure that only one method at a time can run. Figure 72 shows a Join Java example implementation of the monitor object design pattern. In the example, every method is a Join method that will only execute in the presence of a call to *unlocked*. When the class is instantiated, a call to *unlocked* is made by the constructor which will allow the next call to any of the methods possible. When a call to any of the methods occur the method will be allowed to proceed but no others. When the method is complete it recalls *unlocked* making it possible for another waiting method to be executed. This strategy ensures that only one method at a time is executable. As a last step each method must call the *unlocked* Join fragment to allow another waiting method to continue. This is almost the equivalent of the synchronized keyword in Java with one major difference. In Java a synchronized block of code may hold the lock more than once (recursive locking) whilst Join Java you must contend with the other callers for a lock. Due to the flexible of Join Java you could emulate the recursive locking of normal Java by removing the synchronized code from the locking method into its own method and allowing the new method to be called directly from inside a locked method.

```
class MonitorObject {
        public MonitorObject() {
                unlocked();
        }

        void method1(String param) & unlocked() {
                //synchronized code
                System.out.println(param);
                unlocked();
        }

        void method2(String param) & unlocked() {
                //synchronized code
                System.out.println(param);
                unlocked();
        }

        void method3(String param) & unlocked() {
                //synchronized code
                System.out.println(param);
                unlocked();
        }

        void method4(String param) & unlocked() {
                //synchronized code
                System.out.println(param);
                unlocked();
        }
}
```

Figure 72. Join Java Monitor Object Code

A final important difference between is that Java monitors are not parameterised whereas Join Java monitor pattern can be parameterised. As in the scoped locking pattern, the simplicity of the Java solution is open to Join Java programmers to use if they wish.

Presented below in Figure 73 is the equivalent Java code with one exception. The Java monitor implementation is re-entrant. That means that a locked method can call another locked method and will reacquire the lock immediately. To implement this in the Join Java extension the combination of the thread safe interface pattern and the monitor pattern would emulate the re-entrant nature of the Java code. The Java code can be seen to be more straightforward as the pattern is natively supported by the language. The advantages of the Join Java extension method of handling monitor style locks is that multiple locks can be used rather than just using the single one implied by the Java implementation. The monitor style locks can also carry parameters that can allow extra information to be carried between lock and unlock operations.

```
class MonitorObject {
        public MonitorObject() {
        }
        synchronized void method1(String param)  {
                //synchronized code
                System.out.println(param);
        }
        synchronized void method2(String param)  {
                //synchronized code
                System.out.println(param);
        }
        synchronized void method3(String param)  {
                //synchronized code
                System.out.println(param);
        }
        synchronized void method4(String param)  {
                //synchronized code
                System.out.println(param);
        }
}
```

Figure 73. Java Monitor Object Code

5.3.4 Half-Sync/Half-Async

The half-sync/half-async pattern decouples asynchronous and synchronous processing in concurrent programs. It provides an asynchronous layer and a synchronous layer and then couples them together with a queue layer to handle the communication between them. Figure 74, Figure 75 and Figure 76 provides a Join Java implementation of this pattern. The first class *HalfSyncHalfASync* is a test class showing how to start up the system. The class

ExternalSource simulates a external source that is asynchronously sending messages to the asynchronous layer *ASyncService*. This layer accepts messages then enqueues them via the *QueueLayer* class for retrieval by the *SyncService* that reads message synchronously. The point to note in this application is the use of non-**signal** return types in the queue layer to block calls from the synchronous layer as compared to the use of **signal** return types in the queue layer to avoid blocking calls from the asynchronous layer.

```
class HalfSyncHalfASync {

   public static void main(String[] argv) {
      //setup
      QueueLayer ql = new QueueLayer();
      SyncService ss = new SyncService(ql);
      ASyncService as = new ASyncService(ql);
      ExternalSource es = new ExternalSource(as);

      //startup sync service
      ss.runningService();

      //cause an external event
      es.someExternalEvent();
   }
}
```

Figure 74. Join Java Half-Sync/Half-ASync Test Code

```
class SyncService {
   QueueLayer queue;
   SyncService(QueueLayer queue) {
      this.queue = queue;
   }
   signal runningService() {
      //this mimics some synch process
      //it will be blocked at the call to
      //read
      System.out.println(queue.read());
   }
}

class ASyncService {
   QueueLayer queue;
   ASyncService(QueueLayer queue) {
      this.queue = queue;
   }
   //this method does not block hence
   //acts asynch
   signal interrupt(Object message) {
      queue.enqueue(message);
   }
}
```

Figure 75. Join Java Half-Sync/Half-ASync Services Code

In Figure 77, Figure 78, Figure 79 and Figure 80 the equivalent Java code is shown. Note that Figure 77 is identical to the Join Java test code but is given again for completeness. Figure 78 shows that the Synch and Async service is a little more complicated with the programmer having to create a thread for the asynchronous service. The thread in the synchronous service is to allow the test code to continue running. Figure 79 shows how much more complicated this pattern is to implement in Java as opposed to Join Java. There are greater than twenty lines of locking and queuing code in Java vs. two lines in Join Java. This is due to the user having to not only create anonymous inner threads but also provide correct locking in order to protect the message counter and the Boolean flag. A subtlety in the implementation is the requirement that the inner thread in the enqueues method must have a reference to the outer class in order for it to use the outer class's monitor. If the method was synchronized one would only synchronize the creation of the thread, the notify method call would in most probability cause a *IllegalMonitorException* to be thrown as the thread state is being modified in an unsafe way. One could argue that the thread is not required in this case as the addition to the data structure is immediate. However, the literal implementation of the pattern requires the method to be asynchronous. This also does not stop this common situation arising in other applications. The Join Java implementation stops these problems arising by encapsulating the lock in the synchronization mechanism. Consequently, the programmer does not need to worry about where to place critical sections of code, as the threads cannot enter the method until they satisfy the locking criteria.

```
class QueueLayer {
   //the read method blocks the caller because
   //its return type is Object
   //enqueue does not block its caller because
   //its return type is implicitly signal
   Object read() & enqueue(Object message) {
      return message;
   }
}
class ExternalSource {
   ASyncService collaborator;
   ExternalSource(ASyncService collab) {
      collaborator = collab;
   }
   signal someExternalEvent() {
      //this just random emits an event
      collaborator.interrupt(new String("Some message"));
   }
}
```

Figure 76. Join Java Half-Sync/Half-ASync Queue and External Source Code

```
class HalfSyncHalfASync {

   public static void main(String[] argv) {
      //setup
      QueueLayer ql = new QueueLayer();
      SyncService ss = new SyncService(ql);
      ASyncService as = new ASyncService(ql);
      ExternalSource es = new ExternalSource(as);

      //startup sync service
      ss.runningService();

      //cause an external event
      es.someExternalEvent();
   }
}
```

Figure 77. Java Half-Sync/Half-Async Test Code

```
class SyncService {
   QueueLayer queue;
   SyncService(QueueLayer queue) {
      this.queue = queue;
   }
   void runningService() {
      (new Thread() {
           public void run() {
                //this mimics some synch process
                //it will be blocked at the call to
                //read
                System.out.println(queue.read());
           }
      }).start();
   }
}
class ASyncService {
   QueueLayer queue;
   ASyncService(QueueLayer queue) {
      this.queue = queue;
   }
   //this method does not block hence
   //acts asynch
   void interrupt(final Object message) {
      (new Thread() {
           public void run() {
                queue.enqueue(message);
           }
      }).start();
   }
}
```

Figure 78. Java Half-Sync/Half-Async Service Code

```
class QueueLayer {
   //the read method blocks the caller because
   //its return type is Object
   //enqueue does not block its caller because
   //its return type is implicitly signal

   private final java.util.Vector messages = new java.util.Vector();
   private int count=0;
   final QueueLayer monitor = this;

   synchronized Object read() {
      while(count < 1) {
           try {
                wait();
           } catch (Exception ex) {/*report error*/}
      }
      return messages.remove(0);
   }

   void enqueue(final Object message) {
      (new Thread() {
           public void run() {
                synchronized (monitor) {
                     messages.add(message);
                     count++;
                     monitor.notify();
                }
           }
      }).start();
   }
}
```

Figure 79. Java Half-Sync/Half-Async Queue Source Code

```
class ExternalSource {
   ASyncService collaborator;

   ExternalSource(ASyncService collab) {
      collaborator = collab;
   }

   void someExternalEvent() {
      //this just randomly emits an event
      (new Thread() {
           public void run() {
                collaborator.interrupt(new String("Some message"));
                //do something
           }
      }).start();
   }
}
```

Figure 80. Java Half-Sync/Half-Async External Source Code

5.3.5 Leader/Follower

The leader/followers pattern is a paradigm that allows multiple threads to take turns handling incoming service oriented requests. That is the pattern supports the idea of a thread pool in which threads are reused so that the overhead of destroying and rebuilding threads is removed. The leader/followers pattern is the mechanism in which these threads are stored and forwarded to the jobs they need to do. Figure 81 shows a simplified example of a leader/followers pattern. In this example, a class called *LeaderFollowers* is created with an asynchronous method (*athread*) which loops on some condition. On each loop, it calls *waitForLeadershipToken* which is blocked until a corresponding call to *leadershipToken* is received. When the *leadershipToken* is received, the method assumes the leadership role does the work required and later relinquishes the leadership role by calling *leadershipToken*. Another thread that has called the method *waitForLeadershipToken* will then be allowed to proceed.

```
ordered class LeaderFollowers {
  boolean someAbortCondition = false;

  public LeaderFollowers(int numThreads) {
    leadershipToken();
    for(int i=0; i<numThreads;i++) {
      aThread(i);
    }
  }

  signal aThread(int identity) {
    System.out.println("New Thread");
    while (true) {
      //wait for leadership and an event
      String ev=waitForLeaderShip();
      //then give up leadership
      leadershipToken();
      //process the event
      if(ev==null) { return; }
      System.out.println("Got Leader Ship Ident "+identity+":"+ev);
    }
  }

  //for a thread to execute it needs leadership
  String waitForLeaderShip() & leadershipToken() & event(String ex) {
    //System.out.println("Someone took leadership");
    return ex;
  }

  String waitForLeaderShip() & leadershipToken() & finished() {
    System.out.println("Thread End!!");
    return null;
  }
}
```

Figure 81. Join Java Leader/Follower Code

```
class LeaderFollowers {
  boolean someAbortCondition = false;
  private java.util.List eventInfo;

  public LeaderFollowers(int numThreads) {
    //Vital to protect data structure this is often forgotten
    eventInfo=java.util.Collections.synchronizedList(
       new java.util.ArrayList());
    for(int i=0; i<numThreads;i++) {
       aThread(i);
    }
  }

  void aThread(final int identity) {
    (new Thread() {
       public void run() {
         System.out.println("New Thread");
           while (!someAbortCondition || eventInfo.size()>0) {
             //wait for leadership and an event
             String ev=waitForLeaderShip();
             //handle the event
             System.out.println(
                "Got LeaderShip Ident "+identity+":"+ev);
           }
           System.out.println("Thread Ends");
       }
     }).start();
  }

  //for a thread to execute it needs leadership
  String waitForLeaderShip()  {
    String eventId=null;
    synchronized(this) {
      try {
        if (eventInfo.size()==0) {
          wait();
        }
        //System.out.println("Someone took leadership");
        eventId = (String) eventInfo.remove(0);
      } catch (InterruptedException ex) {/*report error*/}
      return eventId;
    }
  }

  synchronized void event(String ex) {
    eventInfo.add(ex);
    notify();
  }
}
```

Figure 82. Java Leader/Follower Code

5.4 Simple Concurrency Mechanisms

It is important to establish how Join Java can represent all the basic concurrency and synchronization mechanisms. In this section, a number of low-level synchronization and concurrency mechanisms are emulated. The standard monitor structure is omitted as that has been illustrated as a pattern. Some of these examples are more succinctly expressed in the native Java language however; these examples are given to show that anything expressible in Java concurrency semantics is also expressible in Join Java semantics.

5.4.1 Semaphores

Simply speaking a semaphore is a variable *value* on which two operations manipulate. The first of the two operations *P* (or wait) decrements the variable unless the variable *value* is < 0. If this is the case *P* is blocked until variable *value* is >= 0 and then it decrements *value*. The second operation *V* (or elevate) increments *value*. Both these operations must be atomic in order to avoid race conditions on the check and set operations. Figure 83 shows a Join Java program that exhibits the same behaviour as that of a semaphore. This code is similar to any implementation of semaphores in Java except the blocking semantics of Join Patterns are used instead of the customary *wait*/*notify* methods of standard Java. The Join method *void P() & unblockP()* is the core of the Join implementation. When this object is created a single call to *unblockP()* is made. Consequently, when the first call to *P()* occurs it is allowed to complete. Any further calls to *P()* are blocked in the fifo queue of the pattern matcher. When *V()* is called a call to *unblockP()* is made allowing the next *P()* to be unblocked. One of the main requirements of a semaphore is to create a queue of the waiting threads. In this code, it is not immediately apparent where this queue appears. In fact, the pattern matcher implicitly holds all waiting calls transparently to the programmer in a queue.

```
ordered class Semaphore {
  public Semaphore() {
    unblockP();
  }

  //Proberen (wait)
  public void P() & unblockP() {
  }

  //Verhogen (elevate)
  public signal V() {
    unblockP();
  }

  //if call to V occurs with no waiting Ps eliminate the call
  public signal unblockP() {}
}
```

Figure 83. Join Java Code Emulating Semaphores

As can be seen with emulating unbounded semaphores[13] there is a significant number of lines difference between the Java implementation in Figure 83 and the Join Java implementation in Figure 84. The Join Java implementation appears straightforward with a simple lock being formed that represents a block on calls to *P()* for each decrement of the semaphore. When the *V()* is called the *P()* is unblocked for the next waiting *P()*.

```
class Semaphore {
  private int value=0;

  //wait proberen
  synchronized public void P() {
    value--;
    if(value<0) {
        try {
            wait();
        } catch (InterruptedException ex) {/* handle error */}
    }
  }

  //elevate Verhogen
  synchronized public void V() {
    value++;
    if(value<=0) {
        notify();
    }
  }
}
```

Figure 84. Java Code Emulating Semaphores

[13] These examples could be converted to a counting semaphore by the introduction of a simple counter in the Java example and a number of calls to unBlockP() being called in the constructor of the Join Java example.

5.4.2 Timeouts

A structure that is not directly available in Java[14] is the idea of a timeout on synchronization. This is the ability for a blocked method to timeout rather than wait for the completion of an event that may never occur. Join Java provides no specific mechanism for timing out method calls. However, it is straightforward to incorporate this behaviour into Join Java programs by making use of the **ordered** modifier. Figure 85 shows an example class that has a call to a method *block* which should return if *ready* is called or 2 seconds elapses (a timeout). In this class the compiler is told to use the **ordered** (or deterministic) evaluation behaviour for Join methods. This means that in the case where there are two possible completed patterns it should accept the first one defined in the class. The first method *block* when called, calls a method *timer* which is a thread (due to the **signal** return type) and calls a second method *block2*. The Join method *block2() & ready()* defines the behaviour to exhibit if the *block* method completes before the timeout. In this case it prints a message "All Okay". The third method which is also a Join method *block2() & signalTimeOut()* defines the behaviour if the call times out. The point to note with the two Join methods is that the first Join fragments are identical (*block2)* however, the second fragment is different. Consequently, it can be reasoned that if there is a call to *block2* waiting (which is the case here) the choice of which Join method to execute is made at runtime and depends on which fragment *ready* or *signalTimeOut* occurs first. The fourth method *signalTimeOut* acts as a cleaner that removes the timer's call to *signalTimeOut* in the event that the *block2* call was completed by the arrival of *ready*. The final method is a thread that sleeps for about two seconds[15] and calls the *signalTimeOut* fragment. A true implementation would also have to handle the situation of multiple calls arriving and associating the timeouts to the specific calls. This would be achieved using simple conditionals via the arguments of the timeout fragment.

[14] Java supplies a TimerTask (since version 1.3) (Sun 1996) however, this thesis is examining language level implementations rather than library level.

[15] Due to the implementation of the Java JVM's in Java this means at least two seconds. There is no guarantee that it will happen exactly at 2 seconds.

```
ordered class Timeout {
        //set a timer and block
        public void block() {
                System.out.println("Timeout Program Starting");
                timer(2000);
                block2();
        }

        //This is the code we execute if the
        //method gets woken before the timeout
        private void block2() & ready() {
                System.out.println("All Okay");
        }

        //Message to send if we time out
        private void block2() & signalTimeOut() {
                System.out.println("TimedOut");
        }

        //this is just to clear the pool

        private signal signalTimeOut() {}

        public signal ready() {}

        //Timer to generate an abort
        public signal timer(long time) {
                //delay
                try {
                        System.out.println("thread sleeping");
                        Thread.sleep(time);
                        System.out.println("thread waking");
                } catch (Exception ex) {
                        System.err.println("sleep Exception");
                }
                signalTimeOut();
        }
}
```

Figure 85. Join Java Timeout

In Figure 86, the equivalent standard Java code is shown. The shorter Java solution using the *wait(time)* to block and then use a *notify* to prematurely abort the timeout does not however allow the programmer to distinguish what caused the wait to reawaken. You would then have to use flags in that example which would complicate the code. Consequently using the Thread method interrupt is the most concise solution available that still allows the programmer to distinguish between a timeout and an abort.

```
class Timeout {
        //time to timeout at
        final long time = 2000;
        Thread waiter=null;

        //method to prematurely abort the blocked call
        void abort() {
                waiter.interrupt();
        }

        //code to execute when timeout occurs
        void ready() {
                System.out.println("Method Timed Out");
        }

        void block() {
                try {
                        waiter = Thread.currentThread();
                        synchronized(this) {wait(time);}
                        ready();
                } catch (InterruptedException ex) {
                        System.err.println("Did not timeout");
                } catch (Exception ex) {
                        System.err.println("block prob"+ex);
                }
        }
}
```

Figure 86. Java Timeout

5.4.3 Channels

A channel is an abstraction in which a mechanism is provided to allow messages to be transferred between concurrent code segments. This differs from shared memory systems where a common memory segment is shared and each concurrent code segment accesses it through some locking mechanism. The shared memory tends to be non-scaleable as there is no explicit definition of control. That is the shared memory access tends to be arbitrary and global. This problem is somewhat reminiscent of the problems goto (Dijkstra 1968) introduced into programming of imperative languages. If there is a lot of communication between threads, it is often advisable to use a higher-level abstraction to that of shared memory. One of these solutions is the channel abstraction. In Java whilst there is a pipe class (Sun 1996) there is no specific syntactic support for message passing between concurrent threads. Consequently, in Java to emulate the idea of a message passing construct between threads you need to write library code. This can be seen in Figure 88 where synchronized methods are used by the data structure to store waiting messages. When the input command is called, the parameter is put onto a queue. When a recipient wants a message they call the output method and the message is

```
class UniChannel {
        int output() & input(int value) {
                return value;
        }
}
```

Figure 87. Join Java Uni-Directional Channel

received or blocked until one is available. Comparing this with the Join Java program in Figure 87 it can be seen that Join Java is a lot simpler.

In fact, the Join Java implementation is much more flexible as it allows for pattern matching. If one were to add a second Join method, one could make runtime decisions on which code to execute[16]. In the Java solution, this would be a lot more complicated as one would need to generate all the appropriate code for determining which method body to execute based on the methods called by the user.

5.4.4 Producer Consumer

The producer consumer problem is a standard concurrency problem where two sets of concurrent objects communicate via a shared storage resource. Producers send messages to the shared resource that buffers the messages until a corresponding consumer arrives. Consumers arrive and try to remove messages from the storage resource. They either receive one of the messages left by a producer or are blocked if there are no messages available. In this section,

```
class UniChannel {
        int inputwaiting=0;
        int outputwaiting=0;
        java.util.Vector values=new java.util.Vector();
        synchronized int output() {
                if (inputwaiting==0) {
                        try {
                                wait();
                        } catch (InterruptedException ex) {}
                }
                inputwaiting--;
                return ((Integer)values.remove(0)).intValue();
        }
        synchronized void input(int newvalue) {
                values.add(new Integer(newvalue));
                inputwaiting++;
                synchronized(this) {
                        notify();
                }
        }
}
```

Figure 88. Java Uni-Directional Channel

[16] See section 3.4.1.3 for further information.

```
class Producer {
    private CubbyHole cubbyhole;
    private int number;

    public Producer(CubbyHole c, int number) {
        cubbyhole = c;
        this.number = number;
    }

    signal start() {
        for (int i = 0; i < 10; i++) {
            cubbyhole.put(i);
            System.out.println("Producer #" + this.number
                               + " put: " + i);
            try {
                Thread.sleep((int)(Math.random() * 100));
            } catch (InterruptedException e) { }
        }
    }
}

class Consumer {
    private CubbyHole cubbyhole;
    private int number;

    public Consumer(CubbyHole c, int number) {
        cubbyhole = c;
        this.number = number;
    }

    signal start() {
        int value = 0;
        for (int i = 0; i < 10; i++) {
            //System.out.println("Consuming");
            value = cubbyhole.get();
            System.out.println("Consumer #" + this.number
                               + " got: " + value);
        }
    }
}
(Source Modified From: (Campione, Walrath et al. 2000))
```

Figure 89. Join Java Producer/Consumer Code

the producer consumer code example provided by Sun (Campione, Walrath et al. 2000) is used. This code is presented in Figure 91 and Figure 92. In this example, they have supplied three classes a producer class, a consumer class and a cubbyhole class. The cubbyhole class acts as the communications mechanism between the producers and consumers.

When a producer sends a message, it calls the *put* method in the cubbyhole class leaving the message. This message is buffered waiting on the next consumer to arrive. The Join Java equivalent code is presented in Figure 89 and Figure 90. In the example the code has been designed to as closely as possible reflect the Java example in order to see the improvement in code readability. There are likely code optimizations that could be made in the Join Java

```
public class CubbyHole {
    public CubbyHole() {
        empty();
    }

    public void put(int value) & empty() {
        //System.out.println("Buffer filled");
        filled(value);
    }

    public int get() & filled(int value) {
        //System.out.println("Buffer Emptied");
        empty();
        return value;
    }
}

class Test {
    public static void main(String[] args) {
        CubbyHole c = new CubbyHole();
        Producer p1 = new Producer(c, 1);
        Consumer c1 = new Consumer(c, 1);
        p1.start();
        c1.start();
    }
}
(Source Modified From:(Campione, Walrath et al. 2000))
```

Figure 90. Join Java Producer/Consumer Support Code

version to reduce further the code size. The producer and consumer classes are similar to the Java example except for the use of the asynchronous methods, which allows us to omit some of the overhead code that you need to write in the standard Java example. The Join Java cubby hole class is a lot simpler as it uses two Join fragment, calls *empty* and *filled* to regulate access to the buffer. If the cubbyhole is empty that is an *empty* Join fragment has been called, a *put* method is allowed to complete and return. If it is not empty then the producers *put* method blocks the producer until space is made available. When the consumers *get* command is called and the buffer is not filled then it is blocked until a producer calls the corresponding *put* command. The Join Java cubbyhole class operates in exactly the same way as the Sun example in Figure 92.

```
public class Producer extends Thread {
    private CubbyHole cubbyhole;
    private int number;

    public Producer(CubbyHole c, int number) {
        cubbyhole = c;
        this.number = number;
    }

    public void run() {
        for (int i = 0; i < 10; i++) {
            cubbyhole.put(i);
            System.out.println("Producer #" + this.number
                               + " put: " + i);
            try {
                sleep((int)(Math.random() * 100));
            } catch (InterruptedException e) { }
        }
    }
}
public class Consumer extends Thread {
    private CubbyHole cubbyhole;
    private int number;

    public Consumer(CubbyHole c, int number) {
        cubbyhole = c;
        this.number = number;
    }

    public void run() {
        int value = 0;
        for (int i = 0; i < 10; i++) {
            value = cubbyhole.get();
            System.out.println("Consumer #" + this.number
                               + " got: " + value);
        }
    }
}
(Source: (Campione, Walrath et al. 2000))
```

Figure 91. Java Producer/Consumer Code

```
public class CubbyHole {
    private int contents;
    private boolean available = false;
    public synchronized int get() {
        while (available == false) {
            try {
                wait();
            } catch (InterruptedException e) { }
        }
        available = false;
        notifyAll();
        return contents;
    }
    public synchronized void put(int value) {
        while (available == true) {
            try {
                wait();
            } catch (InterruptedException e) { }
        }
        contents = value;
        available = true;
        notifyAll();
    }
}

public class ProducerConsumerTest {
    public static void main(String[] args) {
        CubbyHole c = new CubbyHole();
        Producer p1 = new Producer(c, 1);
        Consumer c1 = new Consumer(c, 1);
        p1.start();
        c1.start();
    }
}
(Source: (Campione, Walrath et al. 2000))
```

Figure 92. Java Producer/Consumer Support Code

```
class Buffer {
  private int size = 5 ;
  public Buffer() {
    for (int i=0;i<size;i++) {
        System.out.println(".");
        emptyBuffer();
    }
  }
  public void deposit(Object value) & emptyBuffer() {
    filledBuffer(value);
  }
  public Object fetch() & filledBuffer(Object value) {
    emptyBuffer();
    return value;
  }
}
```

Figure 93. Join Java Bounded Buffer

5.4.5 Bounded Buffer

The bounded buffer is a variant on the producer consumer in which the shared storage resource for exchanging messages has a limited number of cells that can be utilized. It is more complicated than the producer consumer problem in the previous section as it has multiple buffers. An example Join Java implementation of the bounded buffer class is given in Figure 93. The buffer class on start up will create a number of calls to the *emptyBuffer* Join fragment. An *emptyBuffer* is called for each buffer cell that is to be made available to the user of the class. These act as the markers for empty cells in the bounded buffer. When a call to *deposit* arrives, if there is an empty buffer available the call will complete by calling a *filledBuffer* method with the message as the parameters. If a *fetch* Join fragment method call occurs and a *filledBuffer* fragment call is available, the parameter of the *filledBuffer* is returned to the caller and a buffer is made available via a call to *emptyBuffer*. In this way, the system keeps a constant number of Join fragments in the pool. These Join fragments are either *emptyBuffer* calls indicating an available buffer cell or a *filledBuffer(<param>)* indicating a buffer cell is full and contains the parameter. Join methods are implicitly synchronized consequently there is no point in which the transitions become non-deterministic in this example.

The standard Java version can be seen in Figure 94. One can see that the code presented here is more complicated as it needs to not only use a data-structure to store the buffer contents it also needs some mechanism to record the status of the buffers. In the example presented above the storage is achieved via an array of *Objects* and the status recording is done via the use of semaphores[17]. This means that the Java version relies on additional code (see Figure 84) in order to get the same result as the Join Java code in Figure 93.

[17] The Java solution could be implemented using monitors.

```
class Buffer {
  private int size = 5 ;
  private Object store[] = new Object[size] ;
  private int inptr = 0 ;
  private int outptr = 0 ;

  Semaphore spaces  = new Semaphore(size) ;
  Semaphore elements = new Semaphore(0) ;

  public void deposit(Object value) {
    spaces.P() ;
    store[inptr] = value ;
    inptr = (inptr+1) % size ;
    elements.V() ;
  }

  public Object fetch() {
    Object value ;
    elements.P() ;
    value = store[outptr] ;
    outptr = (outptr+1) % size ;
    spaces.V() ;
    return value ;
  }
}
```

(Source: (Matthews 2002))

Figure 94. Java Bounded Buffer

5.4.6 Readers Writers

In the reader-writer concurrency problem any number of readers can read a shared resource providing no writer is active. Only one writer can be active at a time and only if there are no readers. Consequently, the problem is giving writers exclusive access to the shared resource without starving the readers. The Join Java program presented in Figure 95 allows writers to have exclusive access to the resource. In this program, the resource is the *eventInfo* array. The readers and writers are asynchronous methods that appear in Figure 96. Each thread takes turns calling either the *read* method or the *write* method. When a *write* method occurs a *startWriting* fragment is called. This call will be blocked until calls to *noReading* and *noWriting* are called. This effectively provides the mutual exclusion that the writer thread needs. When the two methods are called and the *startReading* method continues a counter for the *numWriters's* is incremented. The resource is then modified and the *endWriting* method is called that will then decrement the *numWriters* counter. If the *numWriters* is zero then the *noWriter* method is called allowing either a reader or writer to proceed. The process for readers is similar except it omits the requirement for no readers.

```
class ReaderWriter {
        int numberReads=1000; int numberWrites=1000;
        private java.util.List eventInfo =
             new java.util.ArrayList(100);
        int numReaders=0; int numWriters=0;
        public ReaderWriter() {
                noReading(); noWriting();
                for(int i=0;i<numberReads;i++) {
                        eventInfo.add(i,"INIT"+i+"]");
                }
        }
        private void write(String value,int location) {
                startWriting();
                eventInfo.set(location,value);
                endWriting();
        }
        private String read(int location) {
                startReading();
                String value = (String) eventInfo.get(location);
                endReading();
                return value;
        }
        private void startWriting()&noReading()&noWriting() {
                numWriters++;
        }
        private void startReading() & noWriting() {
                numReaders++;
        }
        private void endWriting() {
                numWriters--;
                noReading();
                if(noWriters==0) noWriting();
        }
        private void endReading() {
                numReaders--;
                noWriting();
                if(noReaders==0) noReading();
        }
        …. Reader method see next figure
        …. Writer method see next figure
}
```

Figure 95. Join Java Reader Writers Source part 1

```
        public signal reader() {
                for(int i=0;i<numberReads;i++) {
                        System.out.println("Reading "+read(i));
                        Thread.yield();
                }
                System.out.println("Reading Ending");
        }
        public signal writer() {
                for(int i=0;i<numberWrites;i++) {
                        System.out.println("Writing ");
                        write("SomeData:"+i,i);
                        Thread.yield();
                }
        }
```

Figure 96. Join Java Reader Writers Source part 2

The Java solution presented in Figure 97 and Figure 98 is similar in complexity to the Join Java version. Like the last pattern, the Java solution makes use of semaphores[18] to reduce the complexity of the code[19]. Whilst in the Join Java code the programmer uses the Join fragment calls to represent the state of the readers and writers in the Java code the programmer represents the state via semaphores. The Java version encodes all the logic in the read and write methods in the class. A subjective observation of this code would indicate that the logic of both programs could be critiqued for different problems. The Join Java code shows somewhat more complexity at the method level with additional methods being created to handle the logic of the interaction. The Java solution shows more complexity at the code level hiding the logic in the code segments. Arguable the Join Java solution is better because it gives the definition of the interaction between the readers and writers at the message definition level of the class specification.

```
class ReaderWriter {
        int numberReads=1000;
        int numberWrites=1000;
        Semaphore countProtect = new Semaphore(1);
        Semaphore dataProtect = new Semaphore(1);
        int noReaders=0;
        private java.util.List eventInfo =
              new java.util.ArrayList(100);

        private void write(String value,int location) {
                dataProtect.P();
                eventInfo.set(location,value);
                dataProtect.V();
        }

        private String read(int location) {
                countProtect.P();
                noReaders++;
                if (noReaders==1)
                        dataProtect.P();
                countProtect.V();
                String eventId = (String) eventInfo.get(location);
                countProtect.P();
                noReaders--;
                if (noReaders==0)
                        dataProtect.V();
                countProtect.V();
                return eventId;
        }
        …. Reader method see next figure
        …. Writer method see next figure
}
```

Figure 97. Java Reader Writers Source part 1

[18] This solution could also be implemented as a monitor implementation in Java.

[19] The semaphore library is not shown but can be see in section 5.4.1

```
public void reader() {
  (new Thread() {
    public void run() {
      for(int i=0;i<numberReads;i++) {
        System.out.println("Reading "+read(i));
          Thread.yield();
      }
      System.out.println("Reading Ending");
     }
   }).start();
}

public void writer() {
  (new Thread() {
    public void run() {
      for(int i=0;i<numberWrites;i++) {
        System.out.println("Writing ");
        write("SomeData:"+i,i);
        Thread.yield();
      }
    }
  }).start();
}
```

Figure 98. Java Reader Writers Source part 2

5.4.7 Thread Pool

The final concurrency pattern examined in this chapter is that of the thread pool[20]. One of the more expensive portions of thread operations in most object-oriented languages is thread creation and deletion. In a naively written application, most threads are created, complete a set task and then are destroyed. Unfortunately, if these tasks are small and frequent the overhead is considerable leading to a large portion of time being dedicated to thread creation and deletion. A solution to this problem is to reuse threads. In this way the creation/deletion overheads are avoided. In the Join Java solution, (Figure 99 below) threads are created via the asynchronous method *workthread*. When the thread pool is instantiated, a number of threads are created and they all start looping waiting for jobs to arrive. Each thread then waits in the loop calling a Join fragment *getJob()*. This synchronous fragment will block while there are no jobs waiting. When a job arrives (via a call to *assigntask()*) the *getJob()* Join fragment matches with the *assigntask()* fragment to complete the Join method. The *assigntask* method passes in a job to complete and this is returned via the *getJob* method to the thread. The thread then goes on to complete the task. When the task is complete, the thread loops back to call *getJob* again. The rest of the code is consistent with the Java version in Figure 100. The Java version uses a collection class List to store its waiting jobs. The collection classes are not thread safe

[20] This pattern is also known as the Workers pattern

consequently, access to the list needs to be wrapped in a thread safe interface. This is done by a factory method in the Collections class. This is frequently overlooked by inexperienced programmers, as it is not immediately obvious that it is needed. The thread mechanism in the Java version is via an anonymous inner class that loops in a similar fashion to that of the Join Java version. However, the *getJob* method is more complicated as there is no mechanism available to Java to signal and pass objects in the same step. Consequently, the *getJob* method uses the standard wait/notify mechanism to signal when a job is ready. The *assignTask* method uses the list collection to store the waiting job then calls notify to wake up a thread. The Join Java code is much more straightforward and concise than the Java code as it does not require the programmer to handle storage of the jobs and organize the signalling of the threads to retrieve the jobs. This shortens the code considerably allowing the programmer to concentrate on other parts of the implementation.

```
interface Job {
        public void work(int id);
}
class MyJob implements Job {
        String ident;
        public MyJob(String identity) {
                ident=identity;
        }
        public void work(int id) {
          System.out.println("Doing Work Assigned by "+id+":"+ident);
          try { Thread.sleep((int)(Math.random() * 10));
          } catch (InterruptedException e) { }
          System.out.println(
                 "Finished Work Assigned by "+id+":"+ident);
        }
}

class ThreadPool {
        public ThreadPool(int numThreads) {
                for (int i=0;i<numThreads;i++) {
                        workerThread(i);
                }
        }
        signal workerThread(int id) {
                Job currentJob;
                while((currentJob=getJob())!=null) {
                        currentJob.work(id);
                }
                //System.out.println("Aborting");
        }
        Job getJob() & assignTask(Job newWork) {
                return newWork;
        }
}
```

Figure 99. Join Java Thread Pool Source

```
interface Job {
        public void work(int id);
}

class MyJob implements Job {
        String ident;
        public MyJob(String identity) {
                ident=identity;
        }
        public void work(int id) {
          System.out.println("Doing Work Assigned by "+id+":"+ident);
          try { Thread.sleep((int)(Math.random() * 10));
          } catch (InterruptedException e) { }
          System.out.println(
             "Finished Work Assigned by "+id+":"+ident);
        }
}

class ThreadPool {
        private java.util.List waitingJobs;
        public ThreadPool(int numThreads) {
                waitingJobs =
                     java.util.Collections.synchronizedList(
                         new java.util.ArrayList());
                for (int i=0;i<numThreads;i++) {
                        workerThread(i);
                }
        }
        void workerThread(final int id) {
                (new Thread() {
                        public void run() {
                                Job currentJob;
                                while((currentJob=getJob())!=null) {
                                        currentJob.work(id);
                                }
                        }
                }).start();
        }
        synchronized Job getJob() {
                while (waitingJobs.size()==0) {
                        try {
                                wait();
                        } catch (InterruptedException ex)
                             {/*report error*/}
                }
                return (Job) waitingJobs.remove(0);
        }
        synchronized void assignTask(Job newWork) {
                waitingJobs.add(newWork);
                notify();
        }
}
```

Figure 100. Java Thread Pool Source

5.4.8 Other Patterns

The content of this chapter outline only the most common design patterns in use today. There are still quite a number of other patterns that this thesis has not attempted to cover. In this section, we will briefly mention these and how they relate to the Join Java extension.

The decision concurrent programmers are frequently presented with is the safety vs. performance issue. That is a programmer can make a program perfectly safe by synchronizing all methods. However, the program becomes slow as there is constant locking operations. Alternatively, the programmer can make the program fast by not locking anything and hoping for the best. This makes the program quick but dangerous. This decision is not only faced by application programmers but also library programmers as well. That is library writers are faced with a choice of speed vs. safety. The Java AWT and Swing libraries are designed to give programmers a quick and convenient mechanism for creating graphical user interfaces. However, an issue that arises is the interaction between the GUI library and concurrency. If the library is made thread safe the library will run extremely slow. If it is not made thread safe it is fast but components might be corrupted at runtime if concurrent operations are performed on the graphical components. The library writers of the AWT/Swing chose the fast non-thread safe approach. This is the approach taken in most GUI libraries. GUI's are usually based upon the a central loop which handles events and redraws of the environment. They implicitly do not handle jobs that take a large amount of time. If the loop pauses to complete a job that takes a long time (for example a network connection) the entire GUI stops responding. The natural response to this situation is to create a separate thread that handles the long job. However, this causes a problem, as the GUI libraries are not thread safe and when the job tries to re-enter the event loop it may corrupt other components as they may be in the middle of an update initiated by the event loop. This leaves a problem in the event loop model. How do you handle events that take too long to be handled in a single iteration of the event loop? The approach in Java is to use the *event-loop concurrency* paradigm. This is most visible in the SwingWorker mechanism that is available to Swing programmers. This mechanism basically includes a hook in the event loop that reads events from a queue. When a long job completes it loads itself into the queue. The event loop then reads the completed jobs off the queue in a sequential fashion processing them. This avoids the issue of delaying the event loop for long duration jobs. Join Java interacts nicely with this pattern via the futures pattern presented earlier. For example, Figure 101 shows an example program that illustrates how one would implement a simplified version of a SwingWorker style class in Join Java. In the example, you can see the *construct*

```
ordered class JoinSwingWorker {
   public Object construct() {
        //long duration code
        //then call fragment completed()
        completed();
   }

   public Object finished() & completed() {
        //completion code is placed here
   }

   //this method handles the situation when the event loop
   //tries to get a result before the thread is complete
   public Object finished() {
        return null;
   }
}
```

Figure 101. Event Loop Concurrency

method that handles the time consuming task (the threading is omitted in this example for simplicity). When that method completes it calls the *completed()* fragment. The event loop periodically calls *finished* seeking a result. If the *completed* fragment is not available, the Join class returns a null otherwise; if the *completed* fragment is available it returns an object.

A related feature of the event loop concurrency above is that of cancellation or interruption of concurrent method invocations. At present, the Join Java architecture hides access to thread mechanisms from the user. That is interrupt calls are not directly available. The SwingWorker implementation uses interrupt calls to achieve the cancellable future. This was a conscious decision as the purpose of this thesis was to illustrate the core features of the extension. Implementation of interruption could be done trivially as a further extension of the syntax. However, it is still possible to encode a mechanism for terminating active threads in a Join Java class. As in Java, this would need to be done as a loop check within the thread. Otherwise this is the same pattern as the futures pattern presented earlier.

5.5 Conclusion

In this chapter, a number of different common patterns in concurrency were examined. The first section described design patterns relating to synchronization such as *scoped locking*, *strategized locking*, *thread safe interfaces* and *double check locking optimization*. The second section examined patterns that deal more with representation of concurrency such as, *active objects*, *monitor objects*, *half-sync/half-async*, *leader/follower* and *thread specific storage*. The final section examined a selection of simple concurrency mechanisms that are commonly seen in concurrent applications, for example *semaphores*, *timeouts*, *channels*, *producer/consumer*, *bounded buffer*, *reader writers* and *thread pools*. Each of these patterns was implemented in both Join Java and Java. Whilst the coding of these problems is subjective (how many ways are there to code a thread pool?) in general the Join Java solutions were more succinct the higher the level of the problem. That is if you implement low-level mechanisms such as *monitors* or *shared variables* in Join Java the extension is less effective. However, if you implement higher-level mechanisms such as thread pools the solutions in Join Java are a lot shorter and easier to understand than that of the Java solutions.

It can be seen from the examples in this chapter that the structure of Join Java allows message passing to be easily implemented between different processes without having to concern oneself with the low-level details of the implementation. The simple addition of Join methods and the asynchronous return type allowed us to represent better the standard set of design patterns for concurrency and synchronization. With the higher-level abstraction of communications and synchronization, that Join Java provides it can be supposed that a number of the low-level patterns will be required less. This would be due to people making direct conversions of higher-level problems to higher-level concurrency semantics. Presently programmers who wish to allow threads to communicate must create a shared variable then organize the locking of the shared variable then write the code to modify the shared variable. In the case of Join Java, it is just a matter of creating the Join method to represent the communication path that the programmer wishes to establish. The Join Java language better represents higher-level concurrency problems by giving the program additional tools for expressing synchronization. It was shown that Join Java could express virtually all the concurrency patterns presented by Schmidt (2000). The strength of the Join Java pattern implementations is the merging of the channel structure and the synchronization mechanism.

It was also shown that the Join Java extension is especially good at patterns that involve communication between threads. This chapter forms a basis of comparison of both execution time and code length for the benchmarking chapter.

6

Concurrency Semantics

Smalltalk is object-oriented, but it should have been message oriented.

(Alan Kay Creator of Smalltalk)

Table of Contents

6.1 Introduction

In this chapter, two formalisms are examined and how they can be translated directly into Join Java code. By implementing these examples, Join Java is shown to be fully capable of expressing these more formal process semantics. State diagrams (including start charts) are firstly examined showing how they map directly into the Join Java semantics. Petri nets are examined second, again showing how they map into the Join Java. Finally, some conclusions are made regarding Join Java's ability to express concurrency.

6.2 State Charts and Diagrams

Many systems exhibit dynamic behaviour where their reaction is based upon previous events. This ordering of states and transitions can be represented by state charts and state diagrams. In the next two sub-sections, it will be shown how these formalisms can be translated to Join Java.

6.2.1 State Diagrams

State diagrams represent a system as a series of states (circles) connected by transitions (arcs connecting states). A transition usually indicates the condition in which the system will change from a given state to another state connected by the transition. Being graphs, a common representation is an adjacency matrix. These adjacency matrices are interrogated when an event occurs and a shared variable indicating the current state is updated. Applications that use state diagrams represent the transition as introducing data into the system. For example, in a state machine representing an automatic teller the transition *deposit-money* would introduce the amount into the depositing state. This leads us to a disadvantage of the normal adjacency matrix representation. The transitions within an adjacency matrix are represented by changes in position within the structure. There is no mechanism to introduce data directly into the destination state when making the change. To achieve this, additional data has to be provided or control structures to support the input of "parameters" need to be written. In Join Java, this mechanism is supported via parameters of a Join fragment.

Join Java makes the conversion of state diagrams to an object-oriented syntax representation straightforward. The Join Java example here uses Mealy machine (Mealy 1954) interpretation where executing code in the body is associated with a state transitions rather than the state itself. When coding a Join method to represent a transition the Join fragments encode the transitions. Within the Join method, a call to the destination state is made to indicate the new state. To illustrate, Figure 102 shows a graph representing a basic transition from state **A** to state **C** the code representing this state change is shown in Figure 103.

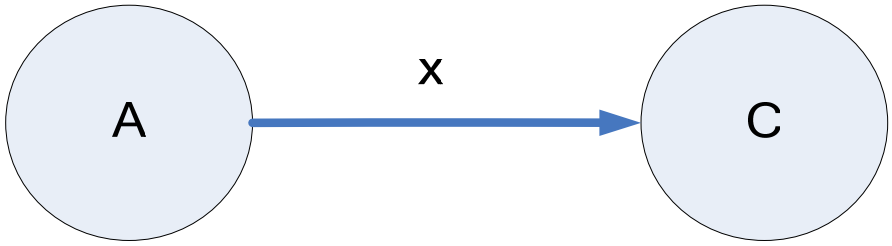

Figure 102. Simple State Transition Diagram

This could be read as transition **x** can be completed when and only when the machine is in state **A**. When the method is executed the Join fragment (Which could also be called a token) **A** representing the state is consumed and a new Join fragment **C** is created, that is the machine is now in state **C**.

Branching is carried out by introducing a rule for each transition out of a state. Using Figure 104 as an example an additional transition (**A** to **B**) is added to the transition that was previously mentioned. The appropriate rule *b() & A() { B() }* is formed. Now if the machine is in state **A**. The decision which Join method to call (and hence what state to transition to) is made at run time by the occurrence of the call (transition) to either *x()* or *b()*. In the event of two calls being made at the same time the system selects the first call that is given priority by the pattern matcher. The system only checks for matches when the pattern matcher is notified of a change in the waiting fragments. In other words the runtime system is reactively checking for matches only when the state of the pool changes.

```
ordered class StateDiagram {
   void x() & A() {
      C();
   }
}
```

Figure 103. State Transition

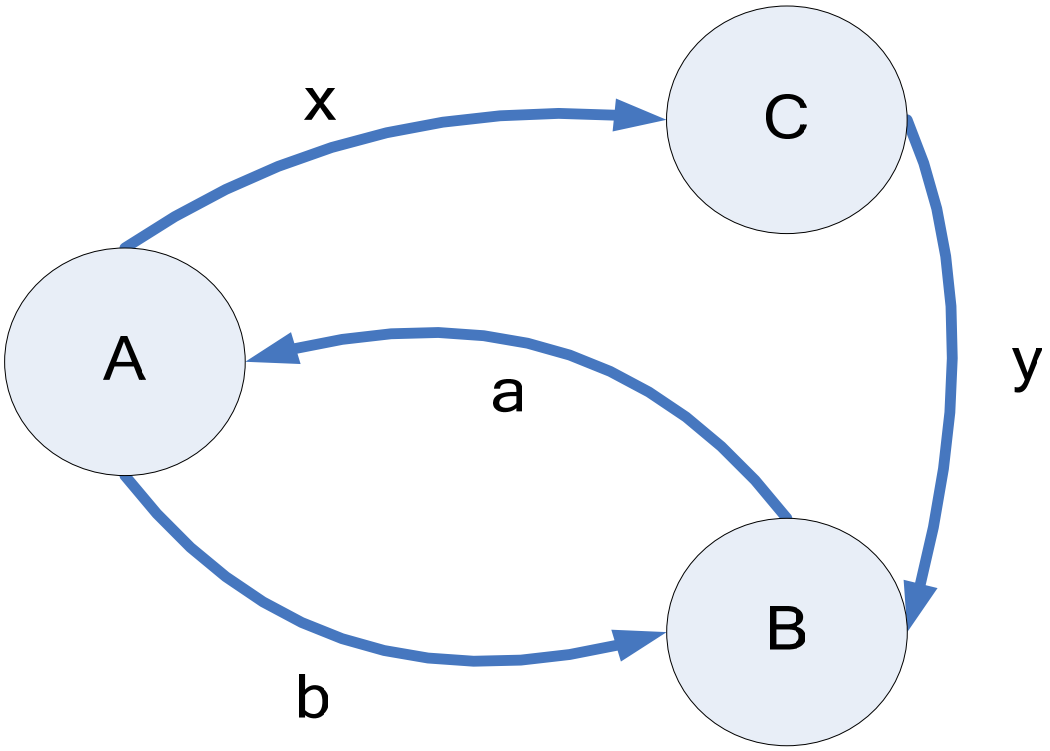

Figure 104. State Diagram

Figure 104 shows a state diagram with states **A**, **B** and **C** and transitions **a**, **b**, **x** and **y**. One Join method per transition (or edge) and an error method to absorb any transition calls made when the machine is not in the correct state only need to be written. The completed code that represents the full state diagram in Figure 104 is given in Figure 105. It can be seen that in the code below that there is a direct mapping between transitions and methods. It also needs to set the initial state (In this case state **A**) and hence the starting state of the machine. This is done by calling *A()* in the constructor. Also note the usage of the **ordered** class modifier; this is used to make sure the transition methods are fired first if possible in preference to the error methods at the end of the class.

6.2.2 State Charts

Similarly, state charts (Harell 1987; Harell, Pnueli et al. 1987) are an extension of the state diagram formalism. It adds three basic elements, hierarchy, concurrency and communication to that of the basic state diagram mechanism. Figure 106 shows a state chart with a number of charts containing state diagrams. Each state chart can have an independent state diagram. Arcs from one state chart to another state chart imply that you may transition from any state within the state chart to the initial state of a destination state chart. The charts act as higher level of a state hierarchy. A line segmenting a chart implies an *and* relationship between the two sub-states. For example, for the **C** transition to occur you must be in states **D** or **E** and **A** or **B**. State charts can be converted to Join Java by adding the inter-chart transformations as additional Join fragments in each transition. Figure 106 gives a state chart in which is described via the

```
\\note the ordered modifier. this means
\\match the first pattern that is defined
\\in preference
ordered class StateDiagram {
   //constructor
   StateDiagram() {
      //constructor sets initial state
      A();
   }
   void x() & A() {
      //code to execute on transition x
      C();  //call to destination state
   }
   void b() & A() {
      //code to execute on transition b
      B();  //call to destination state
   }
   void a() & B() {
      //code to execute on transition a
      A();  //call to destination state
   }
   void y() & C() {
      //code to execute on transition y
      B();  //call to destination state
   }
   //if one of the patterns above does not
   //match these will match.
   //this removes calls that get called
   //when the machine is not in the correct
   //state
   //These could be modified to throw an appropriate
   //exception if you wish to trap incorrect transitions
   void x() {}
   void b() {}
   void a() {}
   void y() {}
}
```

Figure 105. State Diagram Join Java Code

Join Java code in Figure 107.

The initial state is represented by calls in the constructor. In state charts, as there is an extra level of abstraction, additional Join fragments needs to be added to show which chart contains active states. Using the example in Figure 106 it can be seen that there is a transition between **D** and **E** called **y**. This transition resides within chart **inD**. Therefore the condition for transition **y** to be called is that the state chart is in state **D** and in chart **inD**. As a Join method it is *y() & D() & inD()*. The important point in this example is that when a transition occurs from one chart to another one needs to use up any state information left in the state space. In the case of transition **c** (**inD** to **inB**) one must have the **inD** token as well as the possibility of a **D** or an **E** state. The pool must be cleaned of any unused state information. This is done by creating Join patterns that will consume anything that is left in the event that state machine departs from specific state chart. In the **inD** chart the machine needs to consume both the **D** and **E** states. This is

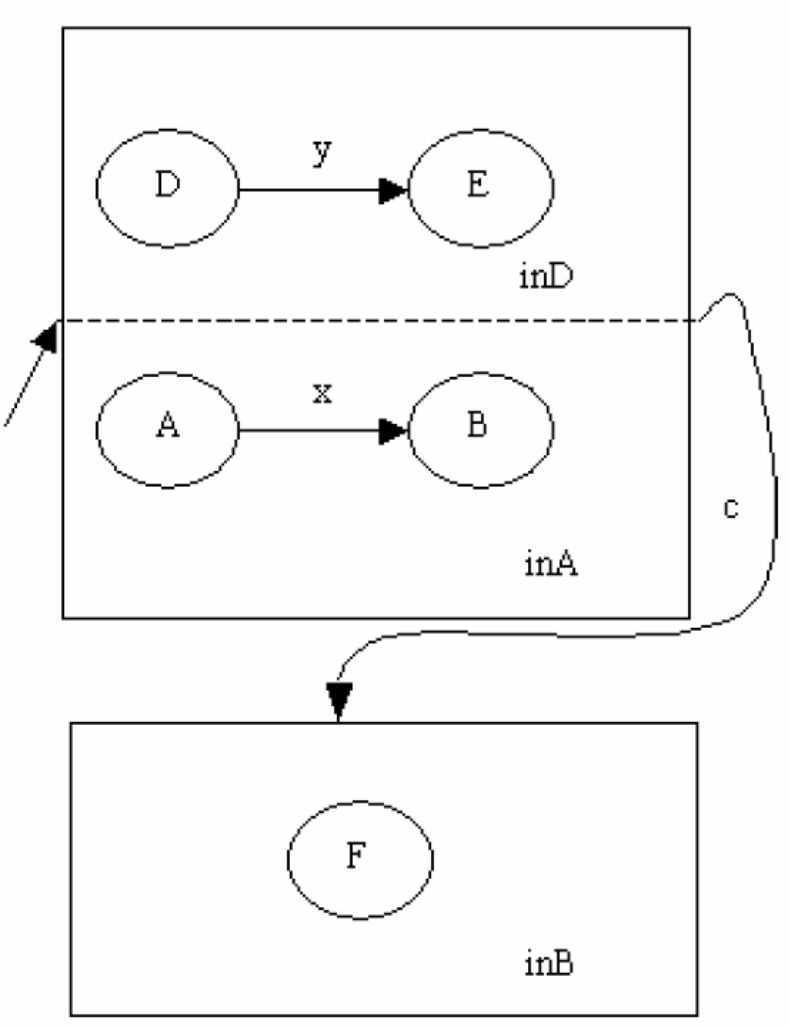

Figure 106. State Chart

illustrated in Figure 107 below where *leaveD() & D()* and *leaveD() & E()* Join methods dispose of any state information that is left. Figure 107 shows the complete code for the state chart in Figure 106. A more object-oriented approach of abstracting the charts is to build a new class for each individual chart. the object can be thrown away when the machine moves to another chart. This removes the necessity of cleaning the pool as the machine leaves a specific chart.

```
ordered class StateChart {
   //constructor
   StateDiagram() {
      //constructor sets initial state
      D();
      A();
      inD();
      inA();
   }

   void y() & D() & inD(){
      //code to execute on transition y
      E();     //call to destination state
      inD();   //make sure it is still in
               //sub-state inD
   }

   void x() & A() & inA(){
      //code to execute on transition x
      B();     //call to destination state
      inA();   //make sure it is still in
               //sub-state inA
   }

   void c() & inA() & inB() {
      //create a new state chart
      inBClass ds = new inBClass();
      //code to execute on transition a
      ds.F();  //call to destination state
      inB();   //make sure it is still in
               //sub-state inA
      LeaveA();
      LeaveB();
   }

   //clean up/take unused state
   //info out of pool
   void LeaveA() & A() {}
   void LeaveA() & B() {}
   void LeaveB() & D() {}
   void LeaveB() & E() {}
   void x() {}
   void y() {}
}
```

Figure 107. State Transition Join Java Code

6.3 Petri-Nets

Petri nets (Petri 1962) are similar to state charts in that they allow the dynamics of a system to be modelled graphically. In addition, the Petri nets can be represented mathematically allowing more rigorous examination of system properties.

A Petri net is composed of four structures, a set of places, a set of transitions, a set of input functions and a set of output functions (relative to the transition). The structure and hence the behaviour of the modelled system is defined by the configuration of places, input functions, output functions and transitions. Places can contain a token that will allow a transition to occur if and only if there are tokens waiting on all other inputs to that transition. A detailed coverage of Petri net theory can be found in (Peterson 1981).

6.3.1 Structures

Diagrammatically Petri nets have circles indicating places, blocks indicating transitions and arcs between places and transitions indicating input and output functions. Figure 108 illustrates the different symbols of a Petri net.

In Figure 108 there are three places named **P1**, **P2** and **P3**. Two input functions **I1** and **I2**, A transition **T1**, and a single output function **O1**. In this case if there are tokens waiting in both **P1** and **P2** then a transition **T1** will be executed and a single token will be placed in **P3** removing a token from both **P1** and **P2**.

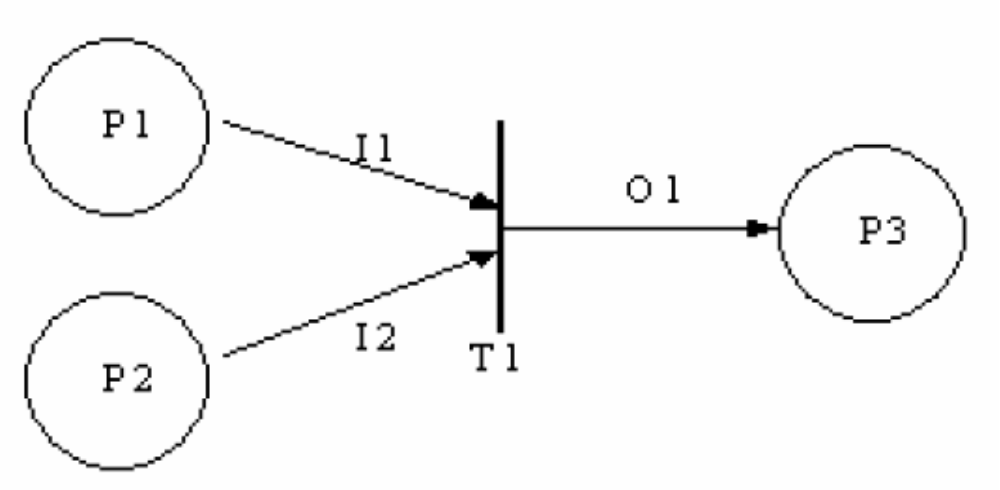

Figure 108. Simple Petri Net

As in the state chart implementations, there exists a straightforward transformation for converting a Petri net implementation into a Join Java program. As Petri nets are transition centered the translation is almost direct. In the examples presented, it can be assume that the transitions fire immediately they are enabled.

In the next two sections will cover two different types of Petri nets. Firstly, unbounded Petri nets, these nets allow multiple tokens to exist in the places at any one time. The second type covered will be one-bounded place Petri nets; these networks only allow one token per place in the network.

6.3.2 Unbounded Petri nets

Unbounded Petri nets whilst being the more complex than bounded Petri-nets are the easiest to implement in Join Java. One needs to produce a method for each transition. The Join method is composed of the places tokens must exist (preconditions). Each Join method will only be fired when at least one token exists in each of the input places. The body of the Join method contains the post conditions such as the destination place as well as any side effects that the transition may entail.

6.3.2.1 An Example in Join Java

Using Figure 109 as an example, it can be seen that there are tokens available on both places **P1** and **P2**. This allows transition **T1** to fire (calls to both **P1** and **P2** are available in the pool). The method will consume **P1** and **P2** tokens and create a **P3** token. Figure 110 shows the state of the Petri net after the transition. The code for this Petri net (Figure 111) shows how one would write the code to represent this net.

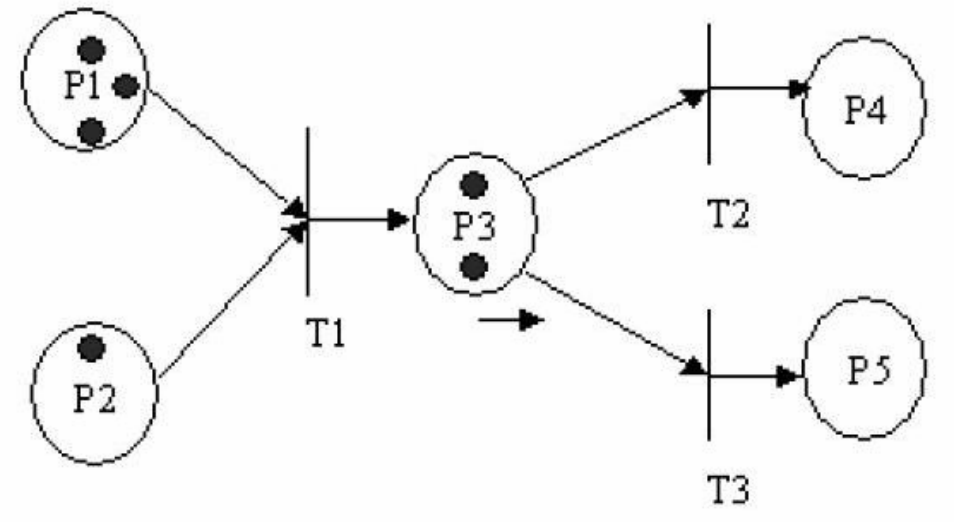

Figure 109. Non-Bounded Petri Net before Transition

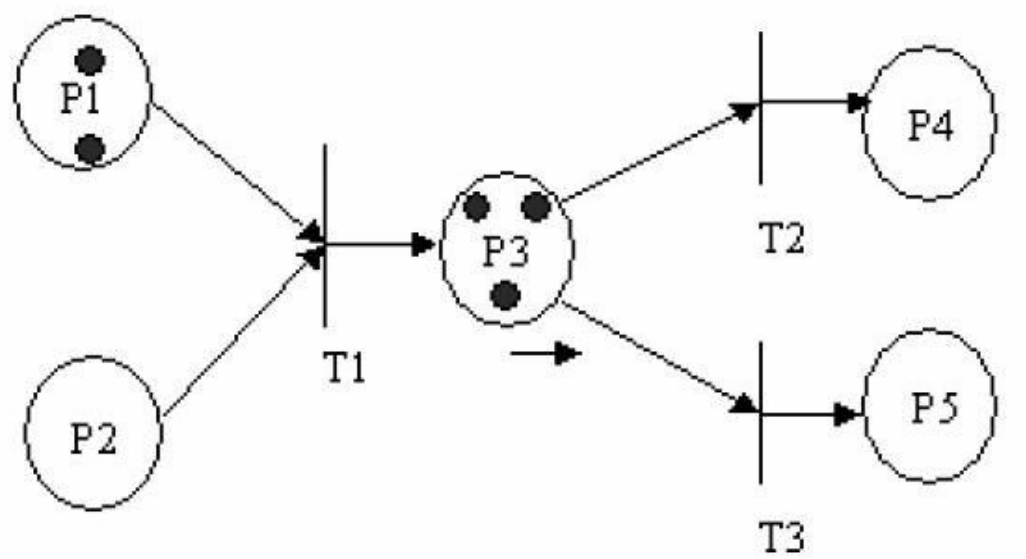

Figure 110. Non-Bounded Petri Net after Transition

6.3.3 Bounded Petri nets

One-bounded Petri nets are similar to unbounded Petri nets except that any one place can only have one token. This means in a one-bounded Petri net a transition will not fire if there is a token already waiting in the output place of the transition.

```
final class PetriNet {
   //transition 1
   signal P1() & P2() {
      P3(); //P3 has now got a token
   }
   signal P3() {
      P4(); //P4 has now got a token
      P5(); //P5 has now got a token
   }
   signal P4() {
      //No transitions out of this
   }
   signal P5() {
      //No transitions out of this
   }
}
```

Figure 111. Non-Bounded Petri Net Join Java Code

6.3.3.1 An Example in Join Java

A one bounded buffer Petri net based on the unbounded Petri net in Figure 109 is used. In a bounded Petri net there is the requirement that the destination place not have a token residing in it when the transition is triggered. In the Join Java implementation this is handled by having a "not" token indicating the lack of a token in the destination place. In other words, a place will have either a token (which shares the name of the place) or a not-token.

Translation of Figure 112 is a matter of taking each transition and place and converting them into a method. Using transition **T1** of the figure as an example places are defined that must have tokens; in this case **P1** and **P2**. **P3** is specified as it must be empty (i.e. a not-token). Therefore the condition on which **T1** will be activated is if *P1() & P2() & notP3()*. Once the condition for the execution of the **T1** transition is defined, the post condition is then specified for the transition. In this case, **P3** has a token **P1** and **P2** are empty. That is *notP1()*, *notP2()* and *P3()*. Figure 113 shows the code for the entire Petri net. Note that in addition to the transitions the network needs to be initialized. This is done via the constructor that initializes the Petri net to the starting state. The definitions of any terminal places are also required to be specified as (places with no output transitions) normal methods.

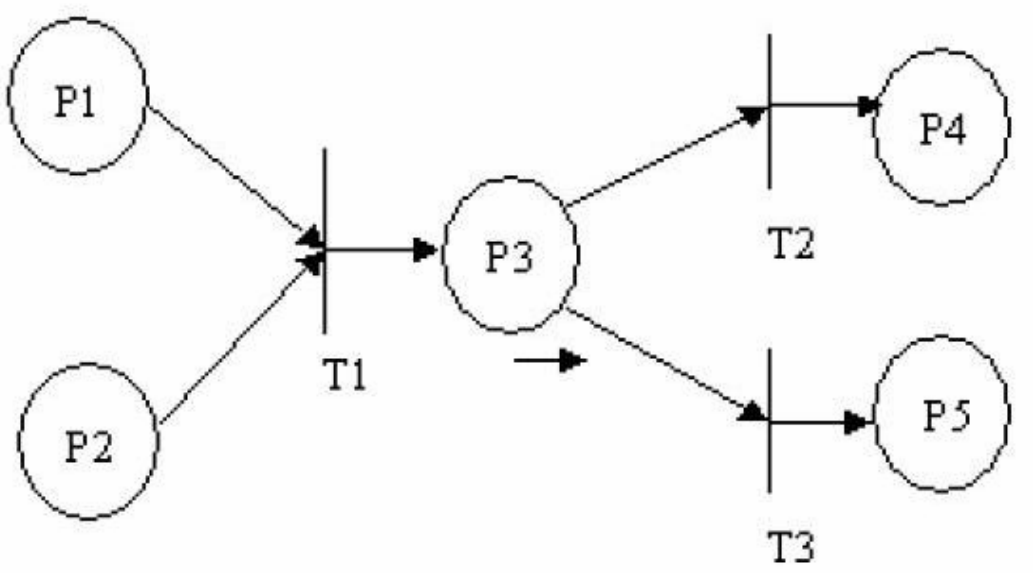

Figure 112. Bounded Petri Net Example

It should be noticed that in this example the return types of the methods are ***signal***. The caller of the method does not wait for execution of the Join method to complete successfully. The other point to note is that the methods do not pass arguments in the example provided. By adding arguments and return types to the Join methods, information can be passed between places (via the transitions). Conceivably this allows the application of the technique to more than just elementary Petri nets. There are a number of non-elementary style Petri nets such as coloured Petri nets (Jensen 1986) that this language could support via arguments in the Join fragments. However, in this thesis the examples have been restricted to covering only elementary Petri nets.

6.3.4 Partial Three Way Handshake Petri net

A more complex example of Petri nets in Join Java is provided in Figure 114. This diagram shows a Petri-net illustrating the opening stages of a TCP three way handshake from the point

```
final class PetriNet {
   //Constructor sets up initial state
   //of PetriNet
   //In this case tokens in places 1,2 and 5
   PetriNet() {
      P1();
      P2();
      notP3();
      notP4();
      P5();
   }
   //Transition 1
   signal P1() & P2() & notP3() {
      notP1();  //P1 is now empty
      notP2();  //P1 is now empty
      P3();     //P3 has now got a token
   }
   //Transition 2
   signal P3() & notP4() {
      notP3();  //P3 is now empty
      P4();     //P4 has now got a token
   }
   //Transition 3
   signal P3() & notP5() {
      notP3();  //P3 is now empty
      P5();     //P5 has now got a token
   }
   void P4() {
      //end of net
   }
   void P5() {
      //end of net
   }
}
```

Figure 113. Bounded Petri Net Join Java Code

of view of the client. In this example, the system starts in the **CLOSED** place and attempts to transition to the **ESTABLISHED** place. The line that travels clock-wise around the diagram indicates the normal path of a client making a connection to a server. The **LISTEN** place is the point that the server would normally wait. For the sake of brevity, only the client's path is covered. This diagram is more complicated as it has unnamed places indicated by light grey dots. Additional labels, **SSYN1**, **SSYN2**, **SYNACK1** and **RSYNACK1** are provided in order to make coding of the Petri net in Join Java easier. If these places were omitted, the transitions on each side would have to be combined together into a single transition possibly changing the meaning of the net. The aim of this net is to step through the process of hand shaking between a client and a server on a TCP connection. The advantage of having a complicated handshaking protocol is that it reduces the chance of confusion between the client and server in respect to which state the other machine currently resides. In the event of a problem, the client and server will return to the **CLOSED** place.

Using the first transition it can be seen that when **Connect** is enabled the token moves from **CLOSED** to **SSYN1**. To convert this into Join Java the requirements of the transition are stated as a conjunction of Join fragments. That is the state is in place **CLOSED** and transition **Connect** is enabled, in Join Java this would be *Connect(Socket) & CLOSED()*. The post conditions are then stated, in this case a token resides in place **SSYN1**, in Join Java this would

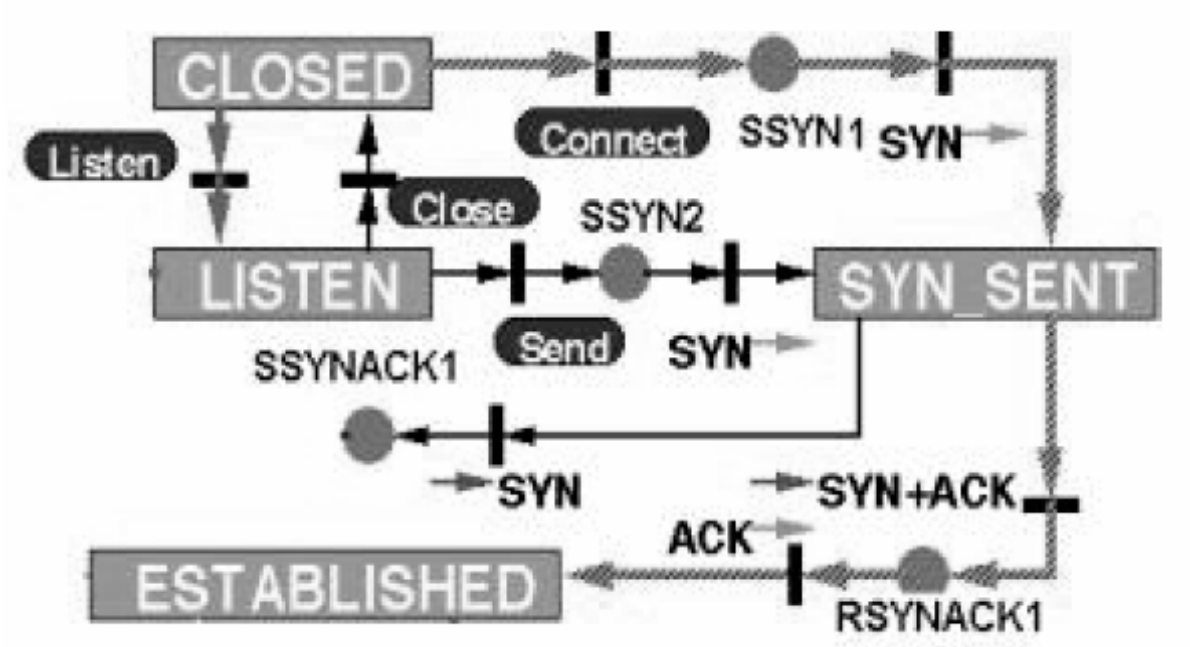

Figure 114. Partial Three Way Handshake

```
signal Connect(Socket x) & CLOSED() {
   SSYN1(x); //place SSYN1 has now got the token
}
```

Figure 115. Petri Net Join Java Method

be *SSYN1(Socket)*. As completed code, the Join Java method would look like Figure 115.

Two points should be noted in this example. Firstly, the method *Connect(Socket)* passes arguments into the Join method. These arguments allow the new place to receive information about the current connection via arguments in the call to the Join fragment *SSYN1(Socket)*. This information would be useful in a full implementation, as the system would need to record socket information in order to communicate with the server. Secondly, the "not" token is not

```
class Socket {}

ordered class TCPHandShake {
   //These methods are called from outside
   //Connect(Socket);,SYNACK();
  //Constructor
   TCPHandShake() {CLOSED();}
   //Transition (1) from CLOSED place to
   //SSYN1 place.
   signal Connect(Socket x) & CLOSED() {
      //send syn to server
      //when complete call SYN()
      //to show that the message
      //has been sent
      //send syn to server
      SYN();
      SSYN1(x); //SSYN1 place has now got
                //the token
   }
   //Transition (2) from SSYN1 place to
   //SYNSENT place.
   //The SYN() method in this transition
   //would be called by the network software
   //when the syn is received.
   signal SYN() & SSYN1(Socket x) {
      SYNSENT(x);
   }
   //Transition (3) from SYNSENT place to
   //RSYNACK1 place.
   signal SYNACK() & SYNSENT(Socket x) {
      //send ack to server
      //when complete call ACK()
      //to show that the message
      //has been sent
      //send ack to server
      ACK();
      RSYNACK1(x);
   }
}
```

Figure 116. TCP Handshake Class (part a)

required in this implementation as there will only ever be one token in the network and hence there is no need to worry about checking for tokens in the destination places.

Figure 116 and Figure 117 shows the full class containing all the transitions required for a successful negotiation between a client and server. As usual, there is need to include the constructor that would initialize the Petri net. The method calls *SYN()*, *SYNACK()*, and *ACK()* in a real implementation would be more complicated as they would need to signal the success or failure of the packet that is sent or received from the server.

This example illustrates how the conditions can be set for progress through the network by adding methods that an external agent may use. In this case the *Connect(Socket)* method above would be called by some external agency (perhaps the user clicking on the connect button in a User Interface). This would allow the network to fire and transition to the second place in the network. Each place in the network would either send a message and move to the next place or wait for the receipt of a message from outside. Again, the ***ordered*** keyword is used to handle the messages that arrive in the incorrect order by returning the token to the **CLOSED** place. In this case the token is sent back to the start position by calling *CLOSED()*. It is assumed that all transitions once enabled will fire immediately.

```
   //Transition (4) from RSYNACK1 place to
   //ESTABLISHED place.
   signal ACK() & RSYNACK1(Socket x) {
      ESTABLISHED(x);
   }
   //ESTABLISHED place
   signal ESTABLISHED(Socket x) {
      // client code to handle
      // application
      Close(x);
      // to start of close sequence
   }
   //commence close procedure
   void Close(Socket x) {}
   //Any other return to CLOSED place
   signal Connect(Socket x) {CLOSED();}
   signal SYN() {CLOSED();}
   signal SYNACK() {CLOSED();}
   signal ACK() {CLOSED();}
}
```

Figure 117. TCP Handshake Class (part b)

6.4 Conclusion

This chapter shows how the formalisms such as state charts and Petri-nets could be mapped directly into the Join Java language without violating object-oriented precepts such as data hiding. For instance, state charts are able to encapsulate each chart as an object that can call the methods in another Join Java object. This means when a state is finished it can be thrown away and the simulation continues. If the system is to re-enter a state it can create a new instance of the state. This is commensurate with the object-oriented paradigm where objects are routinely created and destroyed during runtime. In this chapter, it can be seen that the Join Java language extension is expressive enough to represent clearly both of these popular formalisms. In the next chapter, the succinct representations of concurrency patterns in the previous chapter are used to show that Join Java is not only very expressive but also sufficiently fast.

7

Evaluation

True genius resides in the capacity for evaluation of uncertain, hazardous, and conflicting information.

(Winston Churchill)

Table of Contents

7.1 Introduction

In this chapter, the Join Java language will be evaluated against a standard Java implementation. This chapter approaches the evaluation of the Join extension in two ways. Firstly, a quantitative evaluation in the form of benchmarking is carried out comparing Join Java programs to that of Java programs of similar functionality. This section also examines the compilation speed of the base and extension compilers. Finally, the section looks at factors that affect the speed of the compiler. Secondly, a qualitative evaluation based on an existing evaluation framework is undertaken. The compiler will be evaluated in light of the evaluation criteria specified by (Bloom 1979). Three qualitative criteria are adopted from Bloom. These criteria are modularity, expressive power and ease of use. These criteria are used to evaluate the Join Java extension in light of the standard Java language. This section makes mostly subjective evaluations of Join Java using the criteria. However, as part of this section an objective comparison of Join Java in terms of lines of code vs. Java lines of code is made. The chapter finishes by giving an overview of the findings.

7.2 Performance

In this section, an objective measurement of the speed of patterns implemented in the Join Java extension as compared against those implemented in standard Java is undertaken. This section uses the patterns introduced in Chapter 5 as a basis for the performance comparisons.

As with any higher-level language extension it must be expected that some form of performance penalty due to the overheads involved in doing the additional processing will be incurred. A performance penalty will also be paid on any prototype implementation as it is still targeted at the standard Java platform JVM that has no optimizations for the extension. This is discussed further in the conclusion of this chapter.

For the sake of a proof of concept implementation of the language extension the pre-calculated pattern matcher described in section 4.4.2.2 is used for the benchmarking. This extension has a greater start-up time but operates faster once the program is running. It also has a larger memory footprint than the other pattern matcher implementations.

7.2.1 Pattern Benchmarks

Table 8 shows the results of running the patterns in Join Java versus the patterns in Java. It can be seen that on average the reduction in speed is approximately 30%. Most of these patterns were tested with approximately 10,000 iterations each. The platform these benchmarks were run on was the Sun Java 1.4 JVM running on a 2 GHz Pentium4, 512 Meg memory, Windows XP and CYGWIN. Two of the basic low-level concurrency primitives, semaphores and monitors, are omitted in the benchmarking. This is because Join Java programmers would not want to implement these patterns using Join Java. If a programmer felt the need to use them, they would choose the native versions provided by the Java language.

The benchmark results in Table 8 are graphed in Figure 118 where the speed of Join Java vs. Java can be seen. The Java speed is represented as 100% and the Join Java results are represented by the bars.

Test	Pattern	Java Time (µs)	Join Java Time (µs)	Time Difference	%
1	ActiveObject	32,536,786	40,748,593	8,211,807	125%
2	BoundedBuffer	220,316	300,432	80,116	136%
3	Futures	350,504	420,605	70,101	120%
4	HalfSyncHalfASync	280,403	270,388	- 10,015	96%
5	LeaderFollower	771,109	1,442,073	670,964	187%
6	MonitorObject	981,411	1,311,887	330,476	134%
7	ProducerConsumer	650,936	1,081,556	430,620	166%
8	ReaderWriter	851,224	1,873,692	1,022,468	220%
9	ScopedLocking	3,685,299	4,075,861	390,562	111%
10	StratLocking	2,293,297	2,653,816	360,519	116%
11	ThreadPool	250,360	300,432	50,072	120%
12	ThreadSafeInterface	100,136,472	110,241,001	10,104,529	110%
13	Timeout	5,227,517	5,297,617	70,100	101%
14	UniChannels	26,518,131	28,601,126	2,082,995	108%
				Average	132%

Table 8. Join Java vs. Java Benchmarking Results

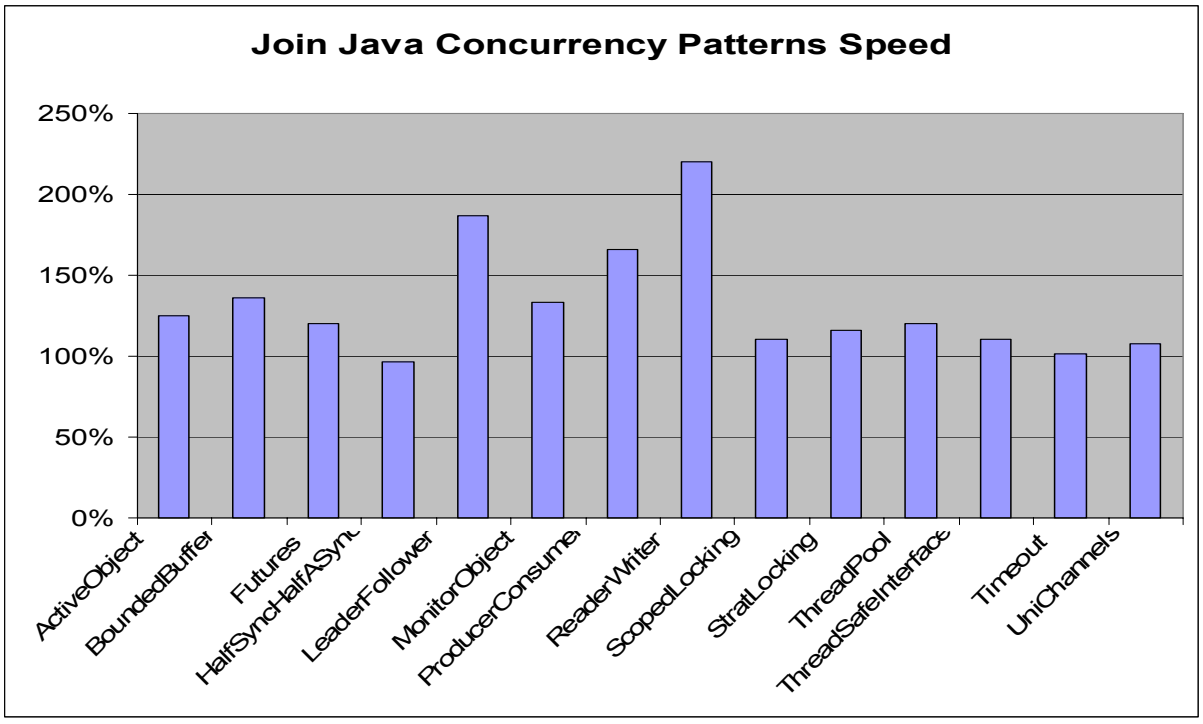

Figure 118. Join Java vs. Java Benchmark Speed (Java = 100%)

7.2.2 Low Level Benchmarks

A set of low-level benchmarks was made on the pattern matcher via a modified version of CaffeineMark (Software 2003) .

Test	Café Test Case	Time (ns)
1	Invoke Join instance void->int	110,878
2	Invoke Join instance void->object	115,207
3	Invoke Join instance object->object	117,349
4	Invoke Join instance int->int	119,061
5	Invoke Join instance void->void	114,965
6	Invoke Join instance (signal)	638,588

Table 9. Join Java Cafe Style Results

Test	Café Test Case	Time (ns)
1	sync method with notify	917
2	sync method with wait	1,240
3	sync method with wait and threading	333,972

Table 10. Java Cafe Style Results

Initial comparison of Table 9 and Table 10 indicates that Join Java is an order of magnitude slower than that of Java code. This is however, not the case as the Join Java code is doing much more than the Java code. Consequently, a fair micro-benchmark requires the Java code to contain threading and wait/notifys. This is done in test 3 of Table 10 where the speed of Java is approximately 300 μ-seconds whilst the equivalent operation in Join Java as shown in test 6 of Table 9 is approximately 600 μ-seconds. From this, it can be seen that on average the Join Java patterns are approximately twice as slow as an equivalent Java method.

7.2.3 Compilation Speed

A number of tests were conducted on the compilation speed of the Join Java compiler vs. that of the standard Sun Java compiler. It should be recognized that the Sun compiler is a production compiler optimized for speed whilst the Join Java compiler is optimized for prototyping. For the benchmarking, we only used standard Java source code, as the Sun JDK is unable to compile Join Java code. In Table 11 we can see the speed comparison of Join Java vs. Sun JDK. In general, we can see that the Join Java compiler on average takes approximately 40% longer to compile a standard Java program than that of the Sun JDK. In the tests, the only outlying value was that of Layout Tutorial. Even when running tests on a number of other

codes examples very few exceeded the 40% to 50%. The only apparent difference between the Layout tutorial code (80% longer to compile) and the other examples is that it has significantly more Java 1.4 API calls. It can be presumed that the extensible compiler (which predates Java 1.4) may not have optimizations for the code being generated for these calls.

Program	Join Java Time	Java Time	Penality
LayoutTutorial (Swing Java Tutorial)	2584	1442	179%
JavaOO (Java Tutorial)	1763	1442	122%
NutsAndBolts(Java Tutorial)	1642	1182	139%
Concepts(Java Tutorial)	1753	1382	127%
Data(Java Tutorial)	1933	1442	134%
Collections Algorithms(Java Tutorial)	1943	1432	136%
Events (Swing Java Tutorial)	3906	2594	151%
Security 1.2 (Java Tutorial)	1642	1172	140%
			141%

Table 11. Compilation Speed Join Java vs. Java

7.2.4 Performance Factors

Given the results of our raw Café style tests (Table 9 and of Table 10) and the results from the concurrency patterns (Table 8 and Figure 118) it can be seen that the Join Java extension in raw terms is approximately twice as slow for low-level artificial benchmarks but approximately 30% slower for pattern implementations. By extension, it is reasonable to expect that on any application where the threading is a minor part of the runtime this penalty will shrink further.

In the prototype language, it was identified that in general, program source code length is shorter but the runtime is slower than that of an equivalent Java program. There are a number of reasons for the reduction of performance over the standard Java program. Some of these issues will be dealt with in this section.

1. Overhead of higher-level abstractions: Join Java abstractions are inherently higher level than that of the standard Java synchronization construct. Java synchronization has no concept of pattern matching and only provides a simple check and set locking mechanism. Consequently, the overhead for this is very low. When introducing a pattern matching mechanism into the synchronization constraints there will be a fixed overhead that will always be slower than that of the basic Java mechanism.

2. Overhead of using a standard JVM: The Java JVM is optimized to support the standard Java instruction such as the test and set monitors of the standard Java language. These

are implemented at the byte code level of the JVM. The JVM can then optimize with full information. The Join Java extension translates to Java that is then executed on the standard JVM byte code interpreter. This is further complicated by the Java hot spot just in time compiler, which is optimized for standard Java. It is speculated that the Join Java language extension defeats optimizations of the hotspot environment. An example of these limitations is on-stack code replacement (Paleczny, Vick et al. 2001). In a standard micro benchmark, the JVM will fail to self-optimize due to the optimization algorithm having an analysis phase and then a optimization phase. Most micro benchmarks will only run once on the stack for each problem. When the test is finished the benchmark moves on, not allowing the optimizer to load the optimized compiled code. This leads to lower than realistic results on the benchmark. The Join Java pattern matcher works in a similar way and there is a possibility that this would lead to circumvention of the hotspot optimizations and lower than realistic results.

3. Overhead of using an extensible compiler: Whilst the extensible compiler is an ideal test platform for prototyping the language there are issues with optimizing extensions. A purpose built environment would have to yield better results as specialized phases could be constructed for analysing and optimizing the runtime pattern matcher.

4. Overhead of separating the pattern matcher from the code: At present, the pattern matcher is a generic code segment that is used for all Join Java programs. The compiler generates calls to the pattern matcher. This aids in prototyping but does not encourage optimization.

5. Runtime Optimization: The pattern matchers that were implemented on the Join Java extension have been simplistic generic mechanisms. Any production compiler would need to use a much more sophisticated algorithm to reduce the overhead of pattern matching at runtime. For example, this could be done by using runtime information from the JVM. At present, due to the pattern matcher being separated not only from the JVM but also from the actual Join Java class, a lot of this information is not available to the pattern matcher to make optimizations. A production compiler would need to more closely link at least the Join Java class to the pattern matcher, preferably also closely linking the pattern matcher to the JVM so that it can make optimizations below the byte code level.

6. Boxing/Unboxing: One problem that was noted in the pattern matcher is a limitation with Java. There is no byte code level support (or optimizations) for boxing and unboxing base classes. This means that whenever there are base type parameters and return types in Join fragments they must be wrapped in order to be stored in the pattern matcher. This creates a huge overhead for the runtime environment, as there is constant object creation and deletion occurring in the JVM. This means the garbage collector is carrying out a larger number of mark and sweep operations at shorter intervals. There are a number of fixes that could be undertaken to reduce the effect of boxing and unboxing. Firstly, boxes could be reused in much the same way as a thread pool reuses threads. When a Join method call finishes the boxes that held the arguments are placed into a pool of free boxes. When a box is required, the boxer goes to the pool and retrieves a box. If none are available, it creates a new one. This avoids a lot of the overhead encountered with object creation and deletion. However, there are more overheads added by the box manager. A second strategy involves generating the pattern matcher at compile time and avoiding the boxing problem alltogether. If this mechanism was implemented, a complete rewrite would be necessary for every change to the pattern matcher algorithm.

7.3 Extension Evaluation

In the previous section, the Join Java language extension was quantitatively benchmarked against the standard Sun Java compiler. In this section, the Join Java extension is evaluated qualitatively using Bloom's criteria. The three criteria are:

1. Modularity: Bloom considers resources to be objects of abstract types. Consequently, there is a set of operations that are associated with the resource. This assumption leads to a design that modularizes the concurrency mechanism using some form of encapsulation. Bloom says that for modularity to be achieved users using a resource should be able to assume that the resource is synchronized and they should not need to provide synchronized code to access the resource.

2. Expressive Power: A concurrency synchronization mechanism must be expressive enough to represent priority and exclusion constraints. If the synchronization mechanism is not expressive enough then it will not be able to express the entire domain of concurrency problems. A way of testing the expressive power is to implement a complete set of problems that cover access, timing, parameters, state and history information. If they can be constructed the mechanism provides sufficient expressive power.

3. Ease of Use: Is the concurrency mechanism expressive enough to express individually the problems presented above? In addition, does the mechanism separate the implementation of each of the constraints? If it does not separate these constraints, complex synchronization schemes become hard to implement due to interactions of the implementation of each of the separate constraints.

These criteria are a test of the abstraction criteria. The criteria are covered in more depth in the following sections where the Join Java language extension will be evaluated using these criteria.

7.3.1 Modularity

Bloom's criteria calls for the concurrency to be modular. That is, its implementation should be separate or separable from the rest of the implementation. This aids in making the software easy to understand and more maintainable. In object-oriented languages, it was identified that modularity (see section 2.4.1) is implicitly achieved via encapsulation. Encapsulation is in turn supported by the class structure. Consequently, it is reasonable to expect that one possible

solution to modularity would be to make use of the class structure of the language to modularize concurrency. Join Java achieves this in a number of ways.

1. Firstly, concurrency is integrated into the language at the method level with pattern matching achieved at the method level. This means that concurrency interaction is localized at the object level where each object has an implicitly separate pattern matcher from other objects.

2. Secondly, Join Java supports the idea of implementing concurrency separate from the implementation of synchronization (see section 3.4.3.3). This can be achieved using interfaces to specify the Join fragments with the implementer of the interface specifying the Join methods. Consequently, the pattern matcher is specified separately from the implementation of the fragments that are in turn separable from the implementation of the threads. This is sympathetic to Bloom's requirement that for modularity to be implemented the implementation of concurrency should be separate from the definition.

3. Thirdly, Join methods can have specific access rights. This means that the Join methods set to private can only be accessed within the class, reducing coupling to other components to the system improving modularity. This mechanism supports Bloom's other requirement that the synchronization mechanism should be separate from the resource that the synchronization mechanism is protecting. Bloom's paper is effectively implying that the synchronization mechanism is a wrapper around the unsynchronized resource.

It can be seen from this subjective evaluation of the Join Java that the extension clearly supports the idea of modularity.

7.3.2 Expressive Power

The second requirement Bloom identified was the idea of expressive power. That is, how well the concurrency mechanism expresses the standard concurrency problems. Bloom was more specific with his evaluation of expressive power by indicating two constraints that must be observed in a concurrency mechanism. These constraints are priority constraints and exclusion constraints. The evaluation of this requirement has been pursued in two ways. Firstly, the Join extension is examined in light of the Bloom criteria. Secondly, an empirical examination of the extension is conducted via a large set of concurrency patterns, specifically looking at the number of lines of code needed to express each standard problem.

7.3.2.1 Priority Constraints and Exclusion

Join Java supports a simple form of priority constraints based on the ordering of Join methods within the class definition. Patterns defined first implicitly gain priority over later defined patterns. Whilst this mechanism is simplistic possible extensions to this policy are examined in future work (section 8.3). Bloom's second constraint is that of exclusion. Join Java supports exclusion with the explicit use of Join fragments before a Join method body is executed. The non-existence of a Join fragment call excludes any Join method that contains that Join fragment from executing. Whilst this is not as clear as having an explicit exclusion operator it does demonstrate the capability of the extension to support exclusion.

7.3.2.2 Concurrent Pattern Evaluation

In addition to the subjective evaluation of the Join Java extension, concurrency patterns were implemented in both Join Java and standard Java. These have already been described in Chapter 5. In this chapter, these examples are used to examine how many lines each implementation takes.

All the concurrency and synchronization patterns covered in (Schmidt 2000) are shown in Table 12 (patterns used for synchronization) and Table 13 (patterns normally used for concurrency). In these tables, the double-checking locking optimization was omitted due to the JVM instruction reordering optimization issued described previously. Finally, a number of standard concurrency problems are covered in Table 14. Each table shows the lines of code for the concurrency pattern implemented in both Join Java and Java. The table also shows the percentage difference and the recommended approach if the sole object is to minimize lines of code.

Name	Join Java LOC	Java LOC	% Difference	Recommend
Scoped Locking	7	15	47%	Join Java
Strategized Locking	74	80	93%	Join Java
Thread Safe Interfaces	32	22	145%	Java
Double Check Locking Optimization	X	X	X	Neither

Table 12. Patterns for Synchronization Lines of Code

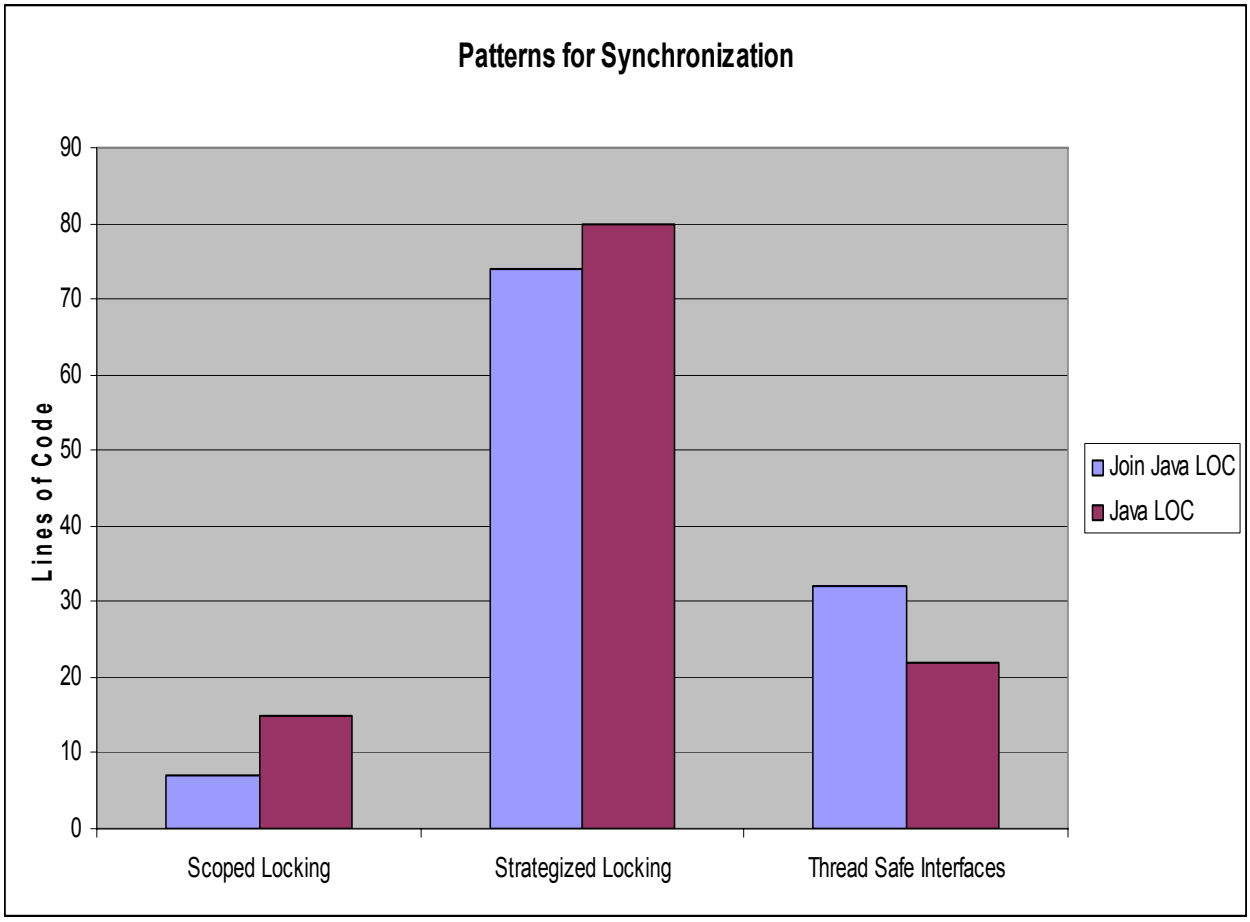

Figure 119. Patterns for Synchronization Java vs. Join Java

In Table 12 and Figure 119 the three major patterns are shown for synchronization (according to (Schmidt 2000)). The only pattern that required less lines of code in Java was that of *thread safe interfaces*. This pattern is more naturally implemented in Java using the monitors already supplied. In this case, the Join Java solution is emulating the features that Java's monitor construct already provides.

Name	Join Java LOC	Java LOC	% Difference	Recommend
Active Object	11	15	73%	Join Java
Futures	15	29	52%	Join Java
Monitor Object	26	20	130%	Java
Half-Sync/Half-Async	49	85	58%	Join Java
Leader/Follower	34	47	72%	Join Java

Table 13. Patterns for Concurrency Lines of Code

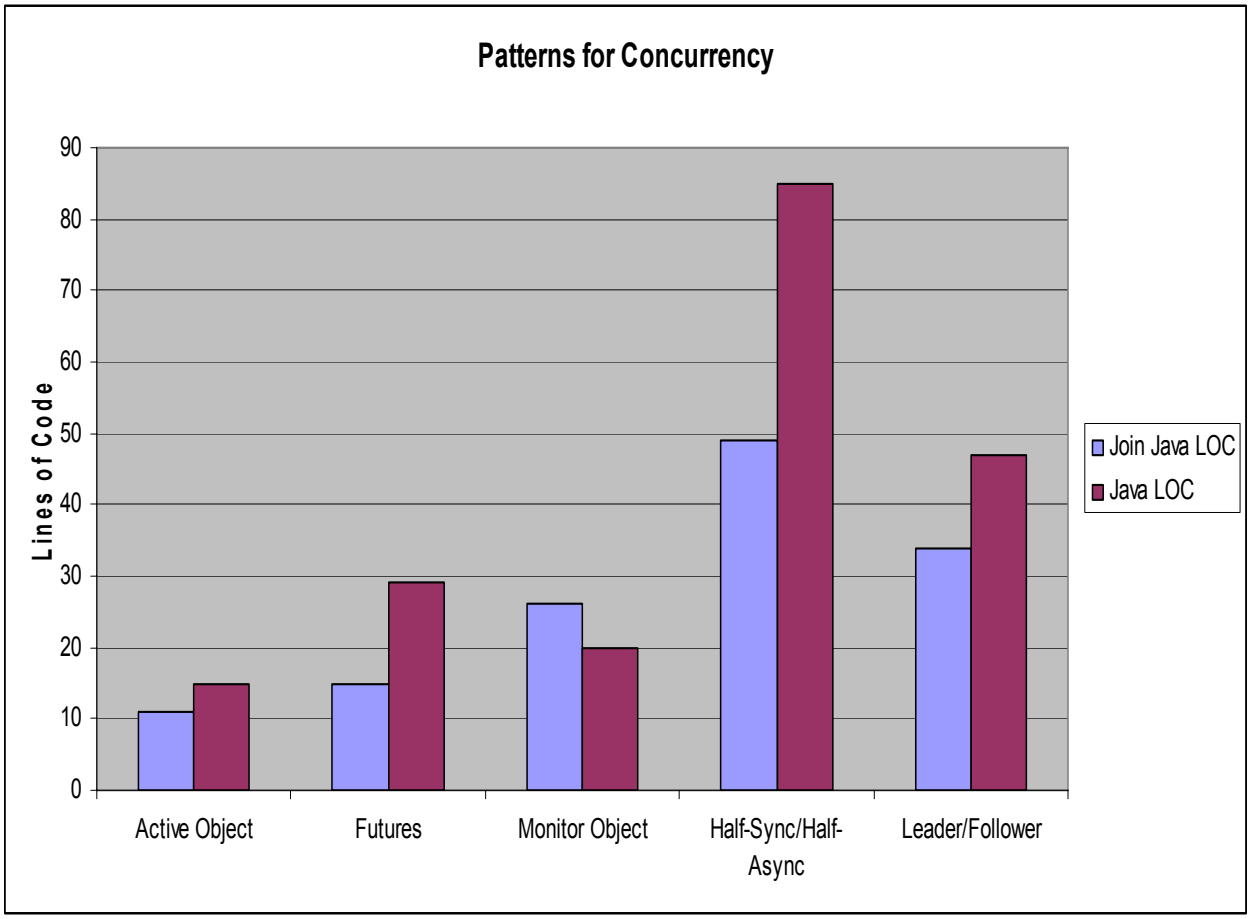

Figure 120. Patterns for Concurrency Java vs. Join Java

In Table 13 and Figure 120, it can be seen that virtually all the patterns can be expressed more succinctly in Join Java than Java. The only exception being the monitor object pattern that is already implemented in Java. Consequently, any Join Java implementation is bound to be more complicated than the explicit implementation of the Java implementation. It should be noted that the Join Java implementation very easily allows parameters to be passed consequently this higher-level abstraction of monitor objects might be in fact be preferable. Examining the remaining patterns, it can be seen that all patterns are more succinct than the Java equivalents. On average, the Join Java implementations are more than 35% shorter than the Java equivalent.

Name	Join Java LOC	Java LOC	% Difference	Recommend
Semaphores	15	24	63%	Join Java
Timeouts	37	30	123%	Either
Channels	5	22	23%	Join Java
Producer Consumer	45	64	70%	Join Java
Bounded Buffer	19	29	66%	Join Java
Readers Writers	66	85	78%	Join Java
Thread Pool (Workers)	37	55	67%	Join Java

Table 14. Simple Concurrency Mechanisms Lines of Code

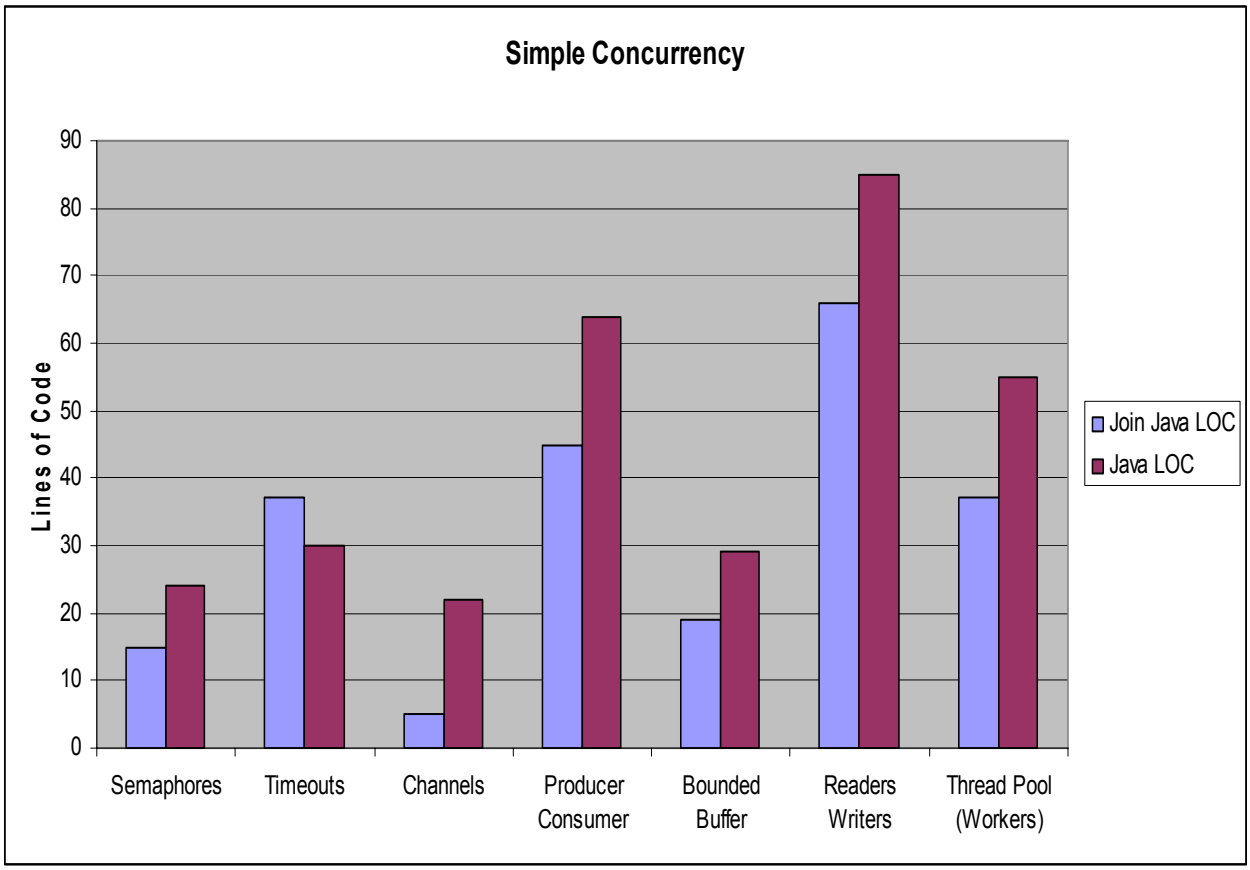

Figure 121. Simple Concurrency Java vs. Join Java

For completeness, in addition to the patterns suggested by (Schmidt 2000), a number of standard problems in concurrency are examined that were not covered previously. Table 14 and Figure 121 list these patterns. It can be seen that in all cases, except that of timeouts the Join Java code is significantly shorter than the Java equivalent program. The timeout code is longer due to the limitation of Join Java not having a mechanism for backing out of a Join fragment call. The code to get around this limitation requires a few extra Join methods to be written at the end of the class definition. Overcoming this weakness is suggested in the future work chapter (section 8.3). Overall, there is a code size reduction of on average 20%.

7.3.3 Ease of Use

The third requirement Bloom identified was that the expression should be easy to use. By this Bloom means that synchronization constraints should be separate from each other. As was identified in the previous section there are two types of synchronization constraints: priority and exclusion constraints. Bloom's criterion says the more independent these constraints are the more easy to use the synchronization construct is. This is the case in Join Java because priority is associated with the ordered modifier and exclusion with the Join methods.

7.4 Conclusion

In this chapter, the Join Java extension was qualitatively and quantitatively evaluated in relation to Java using Blooms criteria and various benchmarking tools.

In the first part of the chapter, the relative performance of Join Java programs was examined in relation to similar problems written in Java. It was shown that Join Java in the prototype implementation is slower by approximately 30%, This is considered to be an acceptable penalty. Finally, the reasons for the current performance were examined and possible improvements to increase the extension speed were suggested.

In the second part of the chapter, the Bloom criteria were used to analyse how good a synchronization mechanism the Join Java language is. It was found that overall the Join Java extension fulfils the criteria. The expressiveness of the language was also examined by analysing the relative lengths of programs written in Java and Join Java. It was found that in nearly all cases the Join Java program was more succinct. It was also noticed that in terms of lines of code the higher-level the concurrency pattern being implemented had bigger differences between the relative low-level implementation of the Java and the high-level implementation of Join Java. This matches with the design criteria of Join Java that it should make the expression of high-level concurrency patterns easier and shorter.

8

Conclusion

Once, after finishing a picture, I thought I would stop for a while, take a trip, do things -- the next time I thought of this, I found five years had gone by.

(Willem de Kooning)

Table of Contents

8.1 Introduction

In this chapter, the contributions that have been made to the field of concurrent programming languages are examined. Finally, how this research can be further extended is suggested.

In this thesis, some weakness with the existing implementations of concurrency in object-oriented languages was shown. It was seen how the use of low-level primitives in object-oriented languages are the modern equivalent of goto statements. Weaknesses in Java's implementation of monitors with its lack of protection of the lock in the synchronized block were identified. It was found that a higher-level abstraction for representing concurrency, synchronization and communication is needed in mainstream concurrent object-oriented languages. This thesis aimed at finding that higher-level abstraction. However, a number of excellent languages have already been suggested that at least partially achieve this aim. These proposals in general are not mainstream as they are academic by nature only illustrating the concepts that the research wanted to show, and not suitable for production level software engineering. Therefore, it is necessary to take an existing popular production language and minimally change it so that it supports the extension but also allows programmers to use their familiar programming tools. To this end, a popular production object-oriented language was chosen to demonstrate the requirements. This thesis identified eight requirements (see Section 3.3) that must be maintained in order to successfully modify an existing language. These requirements are faithfulness, increased robustness, performance, minimalist design, backward compatibility, message passing, true superset and hidden locks. Each of these requirements is now justified with respect to the Join Java extension.

1. Faithfulness: The Join Java extension makes use of existing semantic structures within the language. For instance, Join fragments are based upon the idea of a method signature. They work identically and are interchangeable when being called by non Join Java aware classes. When the Join calculus introduced a concept that was not sympathetic to the object-oriented paradigm of Java it was eliminated from the design. For example, the Join calculus supports the idea of a *reply to* construct which allows a Join method to send return values to several calls from the same method call. This is cognitively different from what imperative programmers are familiar with, consequently it was not included in Join Java.

2. Increased Robustness: The Join Java language prototype is stable and predictable (when using the **ordered** modifier). It has provided an integrated mechanism for

communication between threads with integrated locking that is not inadvertently modifiable by the programmer.

3. Performance: In section 7.2 it was seen that Join Java whilst being slightly slower than Java still has acceptable performance in the prototype. It was also noted that when measuring the speed of realistic applications the delay fell even further to the point of not being noticeable. It was also pointed out how to improve the performance of a production version of the language.

4. Minimalist Design: As the language extension is based on a well-known and popular language, the Join Java extension must be made minimal in order to avoid the cognitive load of learning altered language dynamics. The changes to the language were also localized to only two places within the language. That is the modifiers for a class and the method signatures. The rest of the language works exactly as it does in the base language.

5. Backward Compatibility: Join Java is a superset of Java so this means that any program written in Java is compilable and runnable in Join Java. All Java concurrency semantics are available in Join Java although in most cases are not recommended to be used. This means that a programmer can make a gradual change to the Join Java semantics.

6. Message passing mechanism that complements the method call techniques of serialized applications: The Join pattern component of Join methods gives us a channel formation mechanism that not only mirrors that of normal method calls, it also adds the extra feature of being dynamic in nature.

7. True superset: Language extension should not interfere with the base language method of communication between threads. The Join Java extension is a true superset of the Java language, and no existing feature of the language is disabled to allow the Join Java extension to be integrated. The only restriction of making Join Java classes final only affects classes written with Join Java methods.

8. Locks for communications channels will be hidden from the programmer: This requirement was partially achieved. When a programmer is using the Join Java extension to communicate between threads the lock is hidden from the user. However, if the programmer uses synchronized blocks and shared variables from the base

language the lock is still visible. This is required to meet the other requirements on this list.

In the design patterns and applications chapter it was shown how the introduction of a relatively simple language modification increases the expressiveness of the language. It was shown how the Join Java extension could be used to convey any expression in standard Java in Join Java and in the vast majority of cases have less lines of code. The prototype implementation was shown to have acceptable performance penalties and those penalties were analysed to show how further improvements could be made on a production implementation.

The Join Java language extension fills the gap between the low-level concurrency and synchronization constructs of production languages and the high-level descriptions of concurrency patterns. It does this by providing higher-level abstractions of concurrency that include both an explicit communications mechanism and a method of synchronization of code segments. Due to the localization of synchronization to the method signatures and communication channel selection to the class level it is easier to handle by the programmer. If examined in relation to classic synchronization mechanisms such as semaphores and Java's implementation of monitors, which have poor locality, it can be seen that the Join Java approach is preferable. In addition, neither monitors nor semaphores have communications semantics, as they are simple locking mechanisms. It is usually up to the programmer to provide the communications mechanism via the low-level semantics of the language. This generally leads to success of the implementation being purely a function of the skill of the programmer.

8.2 Contributions

In this section, each of research contributions to the field is identified. There have been a number of general contributions made by this research. Firstly, the provision of high-level concurrency constructs into Java has been demonstrated. An alternative thread communications mechanism to that of the standard shared memory area mechanism used by other object-oriented language was described. Thread integration in Java has been improved allowing threads to be created via a simple asynchronous method call rather than via the external library that is used in the standard Java language. Finally an investigation of how by making minimal changes to a production language higher-level concurrency can be provided for mainstream programmers. During the course of this research, a number of specific contributions were made. These contributions are discussed in more detail in the following sections.

8.2.1 Support for Higher-Level Abstractions at Language Level

This research identified that concurrency abstractions are generally low-level in nature in mainstream production languages. This deficiency leads to scalability problems as most production concurrent object-oriented languages use low-level wait/notify style mnemonics to indicate locking. There generally is no communications mechanism, instead relying on shared memory paradigm in which the locks protect access. Programmers are required to write code to protect the communications mechanism on their own. However, concurrency patterns are high level in nature and are written in terms the programmer understands. The programmer must then translate the patterns into the low-level concurrency semantics of the programming language in order to implement the patterns. In Section 2, this gap between the low-level implementations in the languages and the high-level descriptions of concurrency patterns was identified. It was proposed that the languages should provide a high level set of abstractions that make the mapping of design patterns to code more straightforward. A set of criteria for evaluating abstractions for concurrency based on Kafura's properties (that an abstraction is well named, coherent, minimal and complete) was proposed. It was found that the Join Java extension is generally consistent with these properties. The second property coherence demands that the attributes and behaviours are expected given the situation. This is supported by using the abstractions of methods and method calls. The third property minimalism of the abstraction has been supported by only making minimal modifications to the language to support the functionality that is required. The fourth property completeness requires that the abstraction have enough flexibility to model the domain in which it models. In this case, communication and synchronization is the domain that is being modelled and the criteria is

satisfied as any concurrency problem can be expressed in it. This is illustrated in Chapter 5 and Chapter 6, which shows a wide sot of concurrency expressions. Finally, it was shown how this abstraction supports higher-level abstractions of concurrency better in the design pattern (Chapter 5) where nearly all patterns were expressed in less code. It also was demonstrated in the applications chapter where it was found that two popular process semantics could be modelled with almost a mechanical style transformation.

8.2.2 Introduction of Dynamic Channel Creation Semantics into Java

In addition to standard communications between threads, a mechanism of runtime decisions on where communications will travel was provided. The combination of pattern matching and Join fragments allows dynamic communications channels to be formed at runtime. This is achieved by reusing Join fragments in several patterns. When calls to those fragments are waiting then the decision on which pattern to execute is made at the time the remaining fragments are called. In this way, a program can decide which pattern to dispatch via the selection of which Join fragments to call. This mechanism is a simple yet powerful mechanism for threads to cooperate in solving higher-level patterns.

8.2.3 Improved Thread Integration in Java

Java's thread mechanism is implemented using a specialized class (Thread). When a user wishes to create an additional asynchronous process they need to either create a subclass with a run method that overrides thread class method or pass a specially defined class containing a run method. These methods are non-parameterized consequently, the programmer must make use of global shared memory compromising encapsulation. In the language extension, the idea of asynchronous methods was introduced to Java. These methods are equivalent to standard methods with the exception that once called the caller is released to immediately continue. Consequently, the complicated mechanism of thread creation in Java is replaced by a more straightforward asynchronous return type. Whilst a minimal addition to the language, it improves the conciseness of the language and supports the creation of asynchronous patterns (eg non-blocking first fragments).

8.2.4 Implementation of Process Calculus Semantics into a Production Language

This thesis demonstrated how to implement the semantics from a process calculus (Join) into a mainstream programming language. Whilst this Join calculus has been implemented in other languages, they have been generally only experimental languages. Many of these implementations are more extensive in their adoption of the host process semantics but they tend to be either non-object-oriented or non-mainstream. The Join calculus was selected mostly due to its superior expressiveness to that of other process calculi. It also has explicit synchronization via the conjunction of guard style semantics. The Join calculus is sympathetic to the extensive scoping semantics of object-oriented languages. Explicit synchronization is sympathetic to the programming style of most mainstream languages.

8.2.5 Implementation of a Set of Patterns in a New Concurrent Language Join Java

This thesis implemented a number of concurrency patterns in the Join Java extension. These implementations investigated the capability of the Join Java extension to express a sufficiently diverse set of concurrency problems. It was found that Join Java can express them and in general, they are produced in less lines of code than standard Java. Consequently, it is clear that the thesis has succeeded in showing that Join Java is a superior semantic for expressing higher-level abstractions of concurrency. The investigation concluded with benchmarking of the runtime speeds of the patterns in comparison to that of standard Java implementations. The performance of the prototype was found to be acceptable in virtually all cases. A number of optimizations were suggested for the design of a potential production version of the language.

8.2.6 Close Integration into the Syntax and Semantics of the Base Language

Most high-level abstraction implementations are either library based or an entirely new language that demonstrates the concurrency concepts that the authors are experimenting with. Library based implementations suffer from a number of limitations. Firstly, they are implicitly right hand side operations that is they are dependant on the programmer calling a method or assigning a value to indicate some state change. They then need to call a concluding library function to indicate the closure of that state. This leads to situations where programmers forget to either open or close the state leading to flaws in the program. Compilation errors sometimes

are reported as coming from the library extension rather than the source code complicating debugging. By integrating semantics into the syntactic level of the language, the possibility of these flaws being inadvertently created is reduced. Creating a new language is another option. However, it is hard to get programmers to adopt these new languages. It is also a cognitive overhead in getting users to learn an entire new set of syntax and semantics for the sake of improving one language feature. It makes more sense to retrofit an existing language as long as the extension does not interfere with the cognitive models the programmer has for the rest of the language. Consequently, it is a good idea to make the concurrency model fit into the design criterion of the rest of the language. Thus, our extension used Java as a base language to demonstrate the new extension.

8.2.7 Reduction of Dependency on Low-Level Concurrency Primitives

If the use of low-level concurrency primitives such as *wait* and *notify* is reduced in the Java language the possibility of errors in concurrent programs is reduced. The use of *wait*/*notify* primitives in the Java language violate the encapsulation component of the object-oriented language. The violation of encapsulation also implies the use of side effect code that violates the idea of high cohesion and low coupling. By abstracting the *wait/notify* semantics out of the language and hiding these within the channel abstraction, side effect code is reduced and thus formalizing the communications mechanisms between threads.

8.2.8 Reduction of Reliance on the Low-Level Synchronization Keywords in Java

By supplying a thread communication mechanism that is implicitly synchronized, the necessity of threads requiring synchronized code blocks is removed. Whilst the low-level nature of the synchronized primitive is on occasion preferred due to the simplicity or optimization, it will rarely be required when using Join Java. There is an advantage in using a Join fragment as compared to using a synchronized modifier in a method. You can pass parameters within a Join fragment. This allows locks to pass information between locking operations. A disadvantage of Join Java is that you need to recall the fragment at the end of the method in order to release the lock. This leads to the possibility of dead lock errors if the programmer forgets to recall the Join fragment.

8.2.9 Parameterized Monitors/Parameterized Threads.

As has already been illustrated, thread creation in Join Java is simple. To create a thread in Join Java you only need to nominate the asynchronous return type and when called the method is the semantic equivalent of a normal running Java thread. What's more, the language encourages encapsulation by limiting access to thread contents via the standard access modifiers. As the semantics of the method are used to create the threads in Join Java, the threads implicitly have full access to the range of features such as parameterization and ad-hoc polymorphism. The Join Java thread mechanism is more convenient than the existing class based non-ad-hoc thread implementations.

8.2.10 Investigation of Pattern Matchers and Potential Optimizations

Finally, an initial survey of possible pattern matching algorithms was undertaken. Hybrid pattern matchers are examined and the possibility of compilation-by-compilation generation of pattern matching was explored.

8.3 Future Work

Whilst undertaking this research a number of potentially interesting extensions of the work have been identified. Many of these form extensions to the language itself and the analysis of the interactions with standard object-oriented language. In this section each of these possible future research possibilities are covered in more detail.

8.3.1 Back-Outs and Lock Checking

One feature that presented itself as being useful in Join Java is a method of checking if a Join method would finish if a method were called. This would allow the caller to check for a possible completion before committing to the call. The second option is Linda style features such as lock checking (tuple check without remove). Both these language features would reduce the lines of code needed to solve some concurrency patterns.

8.3.2 Multi-Directional Channels

A Join calculus feature that was not implemented in the Join calculus was the concept of "*reply to*" which is implemented in the original language implementations of (Maranget and Fessant 1998). As has already been stated this was omitted due to cognitive considerations of the programming environments. The advantage of implementing the **reply-to** construct is that the advantage of multi-directional channels can be used. Multi-directional channels allow the programmer to exchange information between two threads in two directions at the same time. At present the Join Java language will only allow information to flow from the callers of the Join fragments to the caller of the first Join fragment (if it is a non-asynchronous/non-void type). Implementation of this would be an interesting problem, as it would require a significant amount of reworking in the architecture of the compiler. It would however, once implemented, reduce the length of a number of concurrency patterns by a significant amount with requisite increase in cognitive overhead to the programmer.

8.3.3 Inheritance

One issue that has been avoided in this thesis is the inheritance issues involved in implementing concurrency in an object-oriented language. Once inheritance is enabled, a number of interesting topics of research would emerge. What is the interaction of Join methods with inheritance? Can pattern matchers be inherited or would they be composable in the subclass?

8.3.4 Expanding Pattern Matching

Whilst a small number of pattern matchers were examined, there is scope to explore fully which pattern matchers are the best for this task. One could also explore how to generate custom pattern matchers at compile time that are optimized for the particular set of patterns of the class being compiled. Secondly, the language could be extended with a mechanism in which the programmer can select other pattern matching policies apart from ordered.

8.3.5 Hardware Join Java

Current work being done by John Hopf is to extend the Join Java compiler to support hardware development for software programmers (Hopf, Itzstein et al. 2002). Hopf has used the high-level semantics of Join Java combined with support for reconfigurable hardware descriptions in VHDL to create a compiler that generates FPGA cores. The Hardware Join Java compiler also generates the driver for the software hardware interface.

8.4 Concluding Remark

The Join Java extension to Java shows great promise as a higher-level abstraction of concurrency for what are normally considered high-level languages. Up to this point production object-oriented languages have been limited by low-level primitives for concurrency. Programmers must learn the art of balancing safety with speed. An overly cautious programmer will usually generate a serialized program. Whilst a programmer coding for speed will not be surprised if they are confronted with their program occasionally entering unsafe states due to unprotected shared code. This thesis has shown with a minimal change to a language a much more powerful semantic for expressing high-level concurrency patterns hence helping programmers build safer code.

9

Index

Any sufficiently advanced technology is indistinguishable from magic.

(Arthur C Clarke)

10

References

I don't believe in personal immortality; the only way I expect to have some version of such a thing is through my books.

(Isaac Asimov)

Agha, G. A. (1986). ACTORS: A Model of Concurrent Computation in Distributed Systems, Cambridge Press, MA.

Aho, A., R. Sethi, et al. (1986). Compilers: Principles, Techniques and Tools, Addison Wesley.

Andrews, D. (1998). Survey Reveals Java Adoption Plans. Byte Magazine. **3:** 26,30.

Andrews, G. R. (2000). Foundations of Multithreaded, Parallel, and Distributed Programming, Addison Wesley Longman, Inc.

Attardi, G. and M. Simi (1981). Consistency and Completeness of OMEGA, a Logic for Knowledge Representation. Proceedings of the Seventh International Joint Conference on Artificial Intelligence. International Joint Conferences on Artificial Intelligence, Menlo Park California.

Bachrach, J. and K. Playford (2001). The Java syntactic extender (JSE). Proceedings of the 16th ACM SIGPLAN conference on Object oriented programming, systems, languages, and applications, Tampa Bay Florida, ACM Press New York, NY, USA.

Bacon, D., J. Bloch, et al. (2002). The double checked locking is broken, http://www.cs.umd.edu/~pugh/java/memoryModel/DoubleCheckedLocking.html. **2004**.

Baeten, J. C. M. and C. Verhoef (1995). Concrete Process Algebra. Handbook of Logic in Computer Science. S. Abramsky, D. M. Gabbay and T. S. E. Maibaum. Oxford, Oxford University Press. **4**.

Baker, J. and W. C. Hsieh (2002). Maya: multiple-dispatch syntax extension in Java. Proceedings of the ACM SIGPLAN 2002 Conference on Programming language design and implementation, Berlin, Germany, ACM Press New York, NY, USA.

Bakkers, A., G. Hilderink, et al. (1999). A Distributed Real-Time Java System Based on CSP. Architectures, Languages and Techniques. B. M. Cook. Broenink Control Laboratory, IOS Press. **citeseer.nj.nec.com/245642.html**.

Benton, N. (2004). "Modern Concurrency Abstractions for C#." ACM Transactions on Programming Languages and Systems, **26**(5): 769-804.

Bergstra, J. A. and J. W. Klop. (1985). "Algebra of communicating processes with abstraction." Theoretical Computer Science **37**(1).

Berry, G. (1993). Preemption in Concurrent Systems. FSTTCS'93 Lecture Notes in Computer Science, Springer Verlag.

Berry, G. and G. Boudol (1992). "The Chemical Abstract Machine." Theoretical Computer Science **1**(96): 217-248.

Black, A. P., M. Carlsson, et al. (2001). Timber: A Programming Language for Real-Time Embedded Systems. Beaverton Oregon, Pacific Software Research Center Computer Science and Engineering Department Oregon Health and Science University.

Bloom, T. (1979). Evaluating Synchronization Mechanisms. Proceedings of the Seventh Symposium on Operating Systems Principles.

Booch, G. (1987). Software Components with Ada: Structures, Tools, and Subsystems. Menlo Park, California, Benjamin/Cummings Publishing Company, Inc.

Briot, J.-P. (1989). Actalk: A Testbed for Classifying and Designing Actor Languages in the Smalltalk-80 Environment. ECOOP 89.

Brookes, S. D., C. A. R. Hoare, et al. (1984). "A theory of communicating sequential processes." Journal of ACM **31**(4): 560--599.

Buhr, P. A. (1992). "muC++ Annotated Reference Manual." **Version 4.8**.

Buhr, P. A., M. Fortier, et al. (1995). "Monitor Classification." ACM Computing Surveys **27**(1): 63-107.

Campione, M., K. Walrath, et al. (2000). Java Tutorial, Addison-Wesley Professional.

Carriero, N. and D. Gelernter (1989). "Linda in Context." Communications of the ACM **32**(4): 444-458.

Chatterjee, A. (1989). FUTURES: a mechanism for concurrency among objects. Proceedings of the 1989 ACM/IEEE conference on Supercomputing, Reno, Nevada, United States, ACM Press.

Chien, A. A. (1990). Concurrent Aggregates: Supporting Modularity in Massively Parallel Programs, Compiler for CM5 and workstations. San Diego, University of California San Diego.

Colby, C., L. Jagadeesan, et al. (1998). Design and Implementation of Triveni: A Process-Algebraic API for Threads + Events. International Conference on Computer Languages. 1998, IEEE Computer Press.

Colby, C., L. Jagadeesan, et al. (1998). "Objects and Concurrency in Triveni: A Telecommunication Case Study in Java."

Crnogorac, L., A. Rao, et al. (1998). Classifying Inheritance Mechanisms in Concurrent Object-orinted Programming. ECOOP'98 - European Conference on Object Oriented Programming, Springer - Lecture Notes in Computer Science 1445.

Dahl, O.-J. and E. Dijkstra, W. (1972). Structured Programming, Academic Press.

Dahl, O.-J. and K. Nygaard (1966). "SIMULA--An ALGOL-based simulation language." Communications of the ACM **9(9)**: 671--678.

Demaine, E. D. (1997). Higher-Order Concurrency in Java. Parallel Programming and Java, Proceedings of WoTUG 20. A. Bakkers. University of Twente, Netherlands, IOS Press, Netherlands. **50:** 34--47.

Dijkstra, E., W. (1968). "Go To Statement Considered Harmful." Communications of the ACM **11**(3): 147-148.

Dijkstra, E., W. (1968). "The structure of the "THE"–multiprogramming system." Communications of the ACM **11**(5): 341-346.

Dijkstra, E., W. (1972). "The Humble Programmer." Communications of the ACM **15**(10): 859-866.

Dijkstra, E. W. and C. S. Scholten (1990). Predicate Calculus and Program Semantics, Springer-Verlag.

Doi, N. and e. al (1988). An Implementation of An Operating System Kernel using Concurrent Object Oriented Language ABCL/c+. ECOOP'88.

Fessant, F. M. F. L. and S. Conchon (1998). Join Language Manual, INRIA (http://pauillac.inria.fr/join).

Fidge, C. (1994). A Comparative Introduction to CSP, CCS and LOTOS. Brisbane, University of Queensland**:** 49.

Fournet, C. and G. Gonthier (1996). The reflexive CHAM and the join-calculus. Proc. 23rd Annual ACM Symposium on Principles of Programming Languages, ACM Press. **January:** 372--385.

Fournet, C., G. Gonthier, et al. (1996). "A Calculus of Mobile Agents." Lecture Notes in Computer Science **1119**.

Fournet, C., C. Laneve, et al. (2000). Inheritance in the Join Calculus. FST TCS 2000: Foundations of Software Technology and Theoretical Computer Science. S. Kapoor and S. Prasad. New Delhi India, Springer-Verlag. **1974:** 397-408.

Fournet, C. and L. Maranget (1998). Join-Calculus Language Manual Release 1.03, INRIA (http://pauillac.inria.fr/join).

Gagnon, E. (2002). A Portable Research Framework For The Execution of Java Byte Code. School of Computer Science. Montreal, McGill University**:** 153.

Gelernter, D. (1985). "Generative Communication in Linda." ACM Transactions on Programming Languages and Systems **7**(1): 80-112.

George, L., D. MacQueen, et al. (2000). Standard ML of New Jersey. New Jersey.

Glabbeek, R. J. v. (1986). Bounded nondeterminism and the approximation induction principle in process algebra, CWI.

Goldberg, A. and D. Robson (1983). Smalltalk-80: The and language and its implementation. Reading, Massachusetts, Adison-Wesley.

Gosling, J. and H. McGilton (1996). The Java Language Environment, Sun Microsystems Computer Company.

Guerby, L. (1996). Ada 95 Rationale. Online Ada Manual. Oust France. **2002**.

Hadjadji, A. (1994). STUDIO - A Modular, Compiled, Actor-Oriented Language, Based Upon a Multitask Runtime System. Joint Modular Languages Conference, ULM.

Hansen, P. B. (1999). "Java's Insecure Parallelism." ACM SIGPLAN Notices **34 (April)**(4): 38--44.

Hardgrave, B. C. and D. E. Douglas (1998). Trends in Information Systems Curricula: Object-Oriented Topics. Association for Information Systems 1998 Americas Conference.

Harell, D. (1987). "A visual formalism for complex systems." Science of Computer Programming **8**(3): 231-274.

Harell, D., A. Pnueli, et al. (1987). On the formal semantics of state charts. Proceedings 2nd IEEE Symp. Logic in Computer Science, Ithaca, NY.

Hewitt, C. (1976). Viewing Control Structures as Patterns of Passing Messages, MIT AI Lab.

Hewitt, C. (1985). "Linguistic Support of Receptionists for Shared Resources." LNCS **197**: 330-359.

Hilderink, G., J. Broenink, et al. (1997). Communicating Java Threads. Parallel Programming and Java, Proceedings of WoTUG 20. A. Bakkers. University of Twente, Netherlands, IOS Press, Netherlands. **50:** 48--76.

Hoare, C. A. R. (1974). "Monitors: An operating system structuring concept." Communications of the ACM **17**(10): 549-557.

Hoare, C. A. R. (1980). Communicating Sequential Processes. On the Construction of Programs -- An Advanced Course. R. M. McKeag and A. M. Macnaghten. Cambridge, Cambridge University Press**:** 229--254.

Hoare, C. A. R. (1985). Communicating Sequential Processes, Prentice Hall.

Holmes, D. (1995). Synchronisation Rings, Macquarie University, Department of Computing.

Holub, A. (2000). If I were king: A proposal for fixing the Java programming language's threading problems, IBM.

Hopf, J., G. Itzstein, Stewart, et al. (2002). Hardware Join Java: A High Level Language For Reconfigurable Hardware Development. International Conference on Field Programmable Technology, Hong Kong.

Hudson, S. (1996). LALR Parser Generator for Java. Georgia, Georgia Institute of Technology**:** CUP.

IEEE (1992). Threads Extension for Portable Operating Systems.

Inmos, L. (1984). Occam Programming Manual. Englewood Cliffs, NJ, Prentice Hall Int.

Intel (2004). Hyper-Threading Technology (www.intel.com developer). **2004**.

Jarvinen, H.-M. and R. Kurki-Suonio (1991). DisCo Specification Language: Marriage of Actions and Objects. 11th International Conference on Distributed Computing Systems. Arlington Texas, IEEE CS Press**:** 142--157.

Jensen, K. (1986). "Colored Petri-nets." Lecture Notes in Computer Science **254**(Advances in Petri-nets): 248-299.

JobNet (2004). Job Trends in IT, JobNet.

Jones, S. P. (2003). Haskell 98 Language and Libraries, Cambridge University Press.

Jones, S. P., J. Hughes, et al. (1998). Report on the Programming Language Haskell 98: A Non-strict, Purely Functional Language., Yale.

Kafura, D. (1989). ACT++: Building a Concurrent C++ With Actors, VPI.

Kafura, D. (1998). Object Oriented Software Design and Construction, Prentice-Hall, Inc.

Kafura, D., M. Mukherji, et al. (1993). "ACT++: A Class Library for Concurrent Programming in C++ using Actors." Journal of Object-Oriented Programming **6**(6).

Kahn, K. and e. al (1990). "Actors as a Special Case of Concurrent Constraint Programming." SIGPLAN Notices **2510**(ECOOP/OOPSLA '90).

Kurki-Suonio, R. and H.-M. Jarvinen (1989). Action System Approach to the Specification and Design of Distributed Systems. Proceedings of the Fifth International Workshop on Software Specification and Design, ACM Press**:** 34--40.

Lea, D. (1998). Concurrent Programming in Java. Reading, MA, USA, Addison-Wesley.

Lea, D. (2002). JSR 166: Concurrency Utilities, http://www.jcp.org/en/jsr/detail?id=166. **2003**.

Levine, T. (1998). "Deadlock control with Ada95." ACM SIGAda Ada Letters **XVIII**(1094-3641): 67--80.

Lieberman, H. (1981). Concurrent Object Oriented Programming in Act1. Object Oriented Concurrent Programming. A. Yonezawa and e. al, MIT Press.

Lim, C. C. and A. Stolcke (1991). Sather language design and performance evaluation. Berkeley, International Computer Science Institute.

Lomet, D. (1977). Process structuring, synchronization, and recovery using atomic actions. ACM Conf. on Language Design for Reliable Software, Raleigh, NC, SIGPLAN Notices Springer-Verlag.

Mapping, A. and R. Team (1994). Programming Language Ada: Language and Standard Libraries, Intermetrics.

Maranget, L. and F. l. Fessant (1998). Compiling Join Patterns. HLCL '98 in Electronic Notes in Theoretical Computer Science. U. Nestmann and B. C. Pierce. Nice, France, Elsevier Science Publishers. **16**.

Maranget, L., F. L. Fessant, et al. (1998). JoCAML Manual, INRIA (http://pauillac.inria.fr/jocaml).

Matsuoka, S., K. Wakita, et al. (1990). Synchronization Constraints With Inheritance: What Is Not Possible - So What Is? Department of Information Science University of Tokyo.

Matsuoka, S. and A. Yonezawa (1993). Analysis of Inheritance Anomaly in Object-Oriented Concurrent Programming Languages. Research Directions in Concurrent Object-Oriented Programming. P. Agha, P. Wegner and A. Yonezawa, MIT Press**:** 107-150.

Matthews, S. (2002). CS237 Concurrent Programming Course Notes. http://www.dcs.warwick.ac.uk/people/academic/Steve.Matthews/cs237/.

Mealy, G. H. (1954). "A Method of Synthesizing Sequential Circuits." Bell System Technical Journal **34**(5): 1045 -- 1079.

Meyer, B. (1988). Object Oriented Software Construction. Engelwood Cliffs New Jersey, Prentice Hall.

Meyer, B. (1997). Concurrency, Distribution,Client-Server And The Internet. Object-Oriented Software Construction, Prentice Hall.

Milner, R. (1980). Calculus of Communicating Systems. New-York, Springer-Verlag.

Milner, R. (1989). Communication and Concurrency. Hertfordshire, Prentice Hall International.

Milner, R., J. Parrow, et al. (1992). "A Calculus of Mobile Processes." Information and Computation **100**(1): 1-40.

Mitchell, S. E. (1995). TAO - A Model for the Integration of Concurrency and Synchronisation in Object-Oriented Programming, University of York.

Nierstrasz, O. (1987). "Active Objects in Hybrid." SIGPLAN Notices **2212**: 243-253.

Nokia (2003). Java API for Nokia phones, Nokia.

Odersky, M. (2000). Functional Nets. Proc. European Symposium on Programming, Springer Verlag**:** 1-25.

Odersky, M. (2000). Functional Nets Slides, LAMP EPFL Lausanne Switzerland.

Odersky, M. (2000). Funnel by Example Technical Report, EPFL Lausanne Switzerland.

Odersky, M., C. Zenger, et al. (1999). A Functional View of Join, University of South Australia.

Paleczny, M., C. Vick, et al. (2001). The Java HotSpot(tm) Server Compiler. Proceedings of the Java Virtual Machine Research and Technology Symposium (JVM 01).

Parrow, J. and B. Victor (1998). The Fusion Calculus: Expressiveness and Symmetry in Mobile Processes. LICS'98. Indianapolis, Indiana, IEEE.

Peterson, J. L. (1981). Petri Net Theory and The Modeling of Systems. Englewood Cliffs N.J, Prentice Hall Inc.

Petitpierre, C. (1998). Synchronous C++: A Language for Interactive Applications. Computer, IEEE. **September 1998:** 65-72.

Petitpierre, C. (2000). Synchronous Java, A quick look at synchronous objects, EPFL Lausanne Switzerland.

Petri, C. A. (1962). Kommunikation mit automaten. Bonn, University of Bonn.

Pons, A. (2002). "Temporal Abstract Classes and Virtual Temporal Specifications for Real-Time Systems." ACM Transactions on Software Engineering and Methodology **11**(3): 291-308.

Ramnath, S. and B. Dathan (2003). "Pattern Integration: Emphasizing the De-Coupling of Software Subsystems in Conjunction with the Use of Design Patterns." Journal of Object Technology **2**(2): 7-16.

Reppy, J. H. (1992). Higher--Order Concurrency. Ithaca, NY, Cornell Univ.

Salles, P. (1984). ALOG Language, Institut de Recherche en Informaique University of Toulouse.

Salles, P. (1984). Plasma II Language, Institut de Recherche en Informaique University of Toulouse.

Salles, P. (1989). Smart Programming Language, Institut de Recherche en Informaique University of Toulouse.

Schanzer, E. (2001). Performance Tips and Tricks in.NET Applications. **2004:** A.NET Developer Platform White Paper.

Schmidt, D. S. (2000). Pattern-Oriented Software Architecture: Patterns for Concurrent and Networked Objects, Wiley and Sons.

Schneider, F. and G. R. Andrews (1983). "Concepts and Notations for Concurrent Programming." ACM Computing Surveys **15**(1): 3-44.

Schneider, F. and G. R. Andrews (1986). "Concepts for Concurrent Programming." Lecture Notes in Computer Science **224**: 669--716.

Software, P. (2003). CaffeineMark 2.5. Libertyville, IL, Pendragon Software Corporation.

Steele, G., J. Gosling, et al. (1995). The Java Language Specification, Addison-Wesley Pub Co.

Stevens, W. P., G. J. Myers, et al. (1982). Structured design' in Advanced System Development / Feasibility Techniques. J. D. Couger, M. A. Colter and R. W. Knapp. New York, John Wiley**:** pp 164-185.

Stroustrup, B. (1983). "Adding classes to the C language: An exercise in language evolution." Software Practice and Experience **13**: 139-161.

Sun (1996). Java API Documentation, Sun Microsystems Computer Corporation.

Sun (1996). The Java Language an Overview, Sun Microsystems Computer Company.

Sun (1998). JavaSpaces White Paper, Sun Microsystems.

Sun (2002). The Java Hotspot Virtual Machine, Sun Microsystems**:** 28.

Sutherland, J. (1999). A History of Object-Oriented Programming Languages and their Impact on Program Design and Software Development, http://jeffsutherland.com/papers/Rans/OOlanguages.pdf.

Theriault, D. (1983). Issues in the Design of Act2, MIT AI Lab.

Watanabe, T. and e. al (1988). "Reflection in an Object-Oriented Concurrent Language." SIGPLAN Notices **2311**: 306-315.

Welch, P. (1999). "CSP for Java (What, Why, and How Much?)." **http://www.cs.ukc.ac.uk/projects/ofa/jcsp/**.

Wellings, A. J., B. Johnson, et al. (2000). "Integrating object-oriented programming and protected objects in {Ada 95}." ACM Transactions on Programming Languages and Systems **22**(3): 506--539.

Yonezawa, A. (1990). ABCL: An Object-Oriented Concurrent System, MIT Press.

Yonezawa, A. (1992). ABCL/R2 Language. Tokyo, Tokyo Institute of Technology.

Zenger, M. and M. Odersky (1998). Extensible Compiler (Erweiterbare Übersetzer). Karlsruhe, University of Karlsruhe.

Zenger, M. and M. Odersky (2001). Implementing Extensible Compilers. ECOOP 2001 Workshop on Multiparadigm Programming with Object-Oriented Languages. Budapest.