Department of Computing
Imperial College London

JCThorn

Extending Thorn with joins and chords

Ignacio Solla Paula

is206@doc.ic.ac.uk

September 30, 2010

Supervisor: Professor Susan Eisenbech

Second Marker: Professor Sophia Drossopoulou

Abstract

Concurrency and distribution are growing areas of concern for programmers due to the widespread presence of multi-core processors, distributed applications and cloud computing. However, writing concurrent applications can still be unnecessarily hard and error-prone, partly because the most popular programming languages were designed for a sequential world.

Many languages rely on shared memory and locking based mechanisms for process communication, which lead to problems such as race conditions, deadlocks or lost of encapsulation. On the other hand, Thorn follows the message passing paradigm, an approach which is undeniably more natural for distributed applications and avoids many of the intricacies of shared memory for concurrent problems. However, solutions to synchronisation problems still require a careful design.

We have created JCThorn, an extension of the Thorn language, that incorporates constructs based on the Join-Calculus in order to simplify the expression of synchronisation points. We added two variations of the same construct, ie. *joins*, for the low-level communication mode of Thorn, and *chords*, for the high-level mode. Our work demonstrates that both constructs can be implemented efficiently with no major performance costs.

Acknowledgements

I would like to thank my supervisor, Dr. Susan Eisenbach, for all the support, guidance and help she has rendered throughout the entire duration of this project. I am also grateful to Professor Sophia Drossopoulou for agreeing to be my second marker.

A huge amount of thanks goes to Bard Bloom, John Field and Jan Vitek for all the help they have rendered throughout this project, especially with questions and problems encountered with the THORN interpreter.

I am also extremely thankful for all the help my family has provided. Muchas gracias por estar siempre ahi cuando lo necesito y apoyarme en todas las decisiones que tomo, especialmente a mis padres.

I would also like to thank Chong for proof reading this report and generally for being such a good friend. I thank all other friends who have repeatedly rescued me over the past months, and made me enjoy the world outside THORN.

Contents

1 Introduction **7**
- 1.1 Contributions . . . 8
- 1.2 Report structure . . . 8

2 Background **11**
- 2.1 Concurrent programming overview . . . 11
  - 2.1.1 Join Calculus for Shared Memory . . . 12
  - 2.1.2 Join Calculus for Message Passing . . . 13
- 2.2 Join Calculus . . . 14
  - 2.2.1 Process algebras . . . 14
  - 2.2.2 Introduction to the Reflexive CHAM and the Join-Calculus . . . 15
  - 2.2.3 RCHAM formally . . . 16
- 2.3 Languages with support for Join-patterns . . . 17
  - 2.3.1 JErlang . . . 17
  - 2.3.2 Jocaml . . . 20
  - 2.3.3 Polyphonic C# . . . 23
  - 2.3.4 Join Java . . . 26
  - 2.3.5 School . . . 27
- 2.4 Thorn . . . 28
  - 2.4.1 Classes . . . 29
  - 2.4.2 Pattern Matching . . . 30
  - 2.4.3 Built-in Data Types . . . 31
  - 2.4.4 Pure data . . . 31
  - 2.4.5 Components . . . 32
  - 2.4.6 High-level Communication . . . 32
  - 2.4.7 Low-level Communication . . . 33
  - 2.4.8 Type System . . . 34
  - 2.4.9 Modules . . . 35
  - 2.4.10 De Bruijn index . . . 36
- 2.5 Summary . . . 36

3 The language **37**
- 3.1 Introduction . . . 37
- 3.2 Joins . . . 41
- 3.3 Chords . . . 43
  - 3.3.1 Algebraic patterns on arguments . . . 44

3.3.2 Inheritance . . . 45
3.4 Common features to both joins and chords . . . 46
3.4.1 Resolution and priorities . . . 46
3.4.2 Non-linear patterns and side conditions . . . 48
3.4.3 Timeouts . . . 49
3.5 Incompatibilities with Thorn . . . 50
3.5.1 Before and After . . . 50
3.5.2 Catch . . . 52
3.6 Summary . . . 53

4 The implementation **55**
4.1 Overview . . . 55
4.2 Grammar and AST . . . 57
4.3 Joins implementation . . . 60
4.3.1 Review of other languages . . . 60
4.3.2 Runtime . . . 62
4.3.3 Local matching phase . . . 63
4.3.4 Contextual matching phase . . . 65
4.4 Chords implementation . . . 66
4.4.1 Translation . . . 66
4.4.2 Catch translation problems . . . 72
4.5 Summary . . . 74

5 State explosion problem and optimisations **75**
5.1 State explosion problem . . . 75
5.2 Successful optimisations . . . 77
5.2.1 Fail Fast (FF) . . . 78
5.2.2 Combinations with Current Message Only (CCMO) . . . 78
5.2.3 Repeating Receive and Context Independence (RRCI) . . . 78
5.2.4 Just-in-time Update of State . . . 81
5.2.5 Skip Contextual Phase (SC) . . . 82
5.2.6 Rank Reordering of join Patterns (RR) . . . 83
5.2.7 Uniquely Satisfied Patterns (USP) . . . 84
5.2.8 Single pattern joins . . . 85
5.3 Failed optimisations . . . 85
5.3.1 Letter Contents Cache (CC) . . . 85
5.3.2 Runtime reordering . . . 86
5.4 Summary . . . 87

6 Evaluation **89**
6.1 Expressiveness . . . 89
6.1.1 Solutions enhanced by joins . . . 90
6.1.2 Solutions worsened by joins . . . 97
6.1.3 When to use joins . . . 100
6.1.4 Numeric priorities . . . 102
6.2 Correctness . . . 103

6.3 Microbenchmarks to test optimisations . . . 105
6.4 Performance . . . 108
6.4.1 Effect of optimisations in typical concurrent problems . . . 108
6.4.2 Thorn versus JCThorn . . . 111
6.4.3 Large mailboxes . . . 113
6.5 Scalability . . . 119
6.6 Integration . . . 120
6.7 Summary . . . 121

7 Conclusions **123**
7.1 Future Work . . . 124

A Alternative solutions to typical concurrent problems **125**
A.1 Dining Philosophers - Thorn's solution . . . 125
A.2 Dining Philosophers - Chord solution . . . 127
A.3 Santa Claus - Thorn's solution . . . 127
A.4 Single-Lane Bridge - Chord solution . . . 128

B Microbenchmarks **130**

C Benchmark **133**
C.1 More results . . . 133
C.2 Code . . . 136

D Armstrong Challege **138**
D.1 With joins . . . 138
D.2 Without joins . . . 140

E Log of changes **141**

Bibliography **146**

Chapter 1

Introduction

The concurrency and distribution mechanisms of many popular programming languages like C# and Java are based on the 1970s model of concurrency, which was primarily concerned with programming for a single machine. Concurrency was achieved through the use of shared memory and threads, and synchronisation based on mutual exclusion. Programs in this model are hard to get right because developers need to be aware of race conditions, identify critical sections and avoid deadlocks. Moreover, making a program dependent on certain shared variables and resources can limit the extensibility and encapsulation of a system, because extending it may require knowledge of these variables.

The message-passing paradigm avoids many of the problems related to shared memory systems, such as race conditions. In addition, it is particularly well-suited to deal with the increasingly important web-based and distributed applications. By nature, these are message-based applications, and rely heavily on asynchronous communication. This fact motivated IBM Research and Purdue University to jointly develop Thorn, a language in which lightweight isolated processes communicate exclusively by message-passing.

Thorn is also well-suited for concurrent applications, and we believe that it can increase the degree of concurrency of a system. By encouraging asynchronous message passing of immutable data, we eliminate the overheads of ensuring that multiple copies are kept consistent. Thorn also eliminates the need for mutual exclusion, since each component, or thread, has exclusive access to its own data. These properties are particularly valuable nowadays, when quad-core processors have already become common for home workstations, and the trend is for the number of cores to continue to increase.

However, synchronisation between threads is still important in many applications. Considering that threads can only communicate by message-passing in Thorn, synchronisation is achieved by exchanging, possibly multiple messages, between a number of components. Nonetheless, the `receive` construct in Thorn can only retrieve a single message at a time from the mailbox of each of the components. A consequence of this limitation is that solutions to certain concurrent problems are unnecessarily hard to program, with some behaviours being impossible to express in Thorn.

1.1 Contributions

To simplify the way synchronisation is achieved in THORN, we propose an extension, that we named JCTHORN, which introduces constructs that allow easy synchronisation on the receipt of multiple messages. These constructs are inspired by the *Join-Calculus*, which is a process algebra developed at INRIA, the French institute for Computer Science, and is suited for implementation. Thus, numerous other languages, such as JOCAML and JERLANG, have emerged at INRIA and elsewhere that were inspired by the *Join-Calculus*.

We incorporate two new type of constructs, *joins* and *chords*, which are to some extent equivalent. In JCTHORN, the difference between the two lies in the fact that *joins* belong to the low-level communication mode of THORN, which sends isolated data items between components, and chords belong to the high-level mode, which sends data through channels declared in each of the components.

We also propose a number of novel and already known techniques to speed up the resolution of *joins*, which suffers from the explosion in the number of states when the components' mailboxes increase in size, because of the number of messages. Basically, *join* resolution has a polynomial worst case and, in this report, we study mechanisms to reduce the amount of computation needed.

We have also performed an in-depth study of the effects of the optimisations and the expressiveness of the language. We determined that JCTHORN performs better than existing languages and that there is indeed situations in which *joins* are extremely valuable.

1.2 Report structure

Chapter 2 presents the background work. It first gives an overview of the two main paradigms in concurrent programming, namely shared memory and message-passing. Then, it introduces process algebras, and in particular the *Join Calculus*. It also examines a number of languages that support *Join Calculus* inspired constructs and, finally, it gives an overview of the THORN language.

In chapter 3 we present the main features of JCTHORN. It starts with an example of a problem that has a straightforward solution in JCTHORN, but is hard to solve in THORN. It presents the JCTHORN's main extensions in more detail, namely *joins* and *chords*, and their common features. It ends with a description of two cases where both languages, THORN and JCTHORN, are incompatible.

Chapter 4 covers the implementation of the JCTHORN language. It shows the changes that had to be made to the grammar and AST. It describes the implementation of *joins*, which required the modification of the execution engine of THORN's interpreter. It concludes with the implementation of *chords*; these are translated into *joins* at compile time.

Chapter 5 is dedicated to the state explosion problem and the optimisations that aim to minimise its effects. It starts by describing in more detail what the state explosion problem is about. It continues by presenting a number of successful optimisations that have been included in the final version of the JCTHORN interpreter; it also describes

other optimisations that have not finally been included in this final version.

An in-depth study of the JCThorn language is presented in chapter 6. Performance and expressiveness are thoroughly analysed. Moreover, we also comment on how we ensured the correctness of the language, its scalability and how well the extensions can be integrated in Thorn. Finally, chapter 7 draws some conclusions and presents future work.

Chapter 2

Background

In this chapter, we first give an overview of concurrent programming and show how the Join-Calculus can improve the way languages handle concurrency and synchronisation in both message passing and shared memory paradigms (section 2.1). Then we investigate the origins of the Join-Calculus, giving its formal definition and presenting the chemical metaphor (section 2.2). A number of languages like Join Java or Jerlang implement constructs based on the Join-Calculus, and these are presented in section 2.3. In section 2.4, the main characteristics of Thorn are covered.

2.1 Concurrent programming overview

Concurrent programs are those designed as a collection of interacting processes or threads. They are executed simultaneously, either sequentially on a single processor by interleaving their instructions or in parallel on multiple cores. Concurrent systems may be classified as shared memory or message passing based on the communication style [33]:

- In **shared memory** systems, processes interact through mutually accessible memory locations. This approach usually requires a locking mechanism (eg. mutexes, semaphores or monitors) to synchronise or coordinate between threads.
- In **message passing** systems, processes interact exclusively by sending and receiving messages and they do not have access to shared memory. Messages may be sent asynchronously, where the sender immediately continues after sending the message, or synchronously, where the sender blocks until the reply is received. Message passing is also commonly used for distributed systems, as the shared memory paradigm is not physically implementable in a distributed setting, although it can be emulated over a message passing system[13].

Many limitations of both approaches can be overcome by making use of *multi-way joins* or *chords* – a concurrency construct inspired on the Join-Calculus. For shared memory systems, *joins* provide an additional method of coordination that avoids the use of shared variables. For message passing systems, they facilitate solving synchronisation problems.

2.1.1 Join Calculus for Shared Memory

Chords, a synchronisation construct inspired by the JOIN-CALCULUS, have become a popular extension to traditional object-oriented languages which rely on mutual exclusion based mechanisms for synchronisation. In these languages, the programmer is responsible of ensuring that processes do not interfere with each other, have exclusive access to shared resources and a deadlock situation does not occur. *Chords*, together with asynchronous methods, provide an alternative to shared memory coordination. These two enhancements have been added to languages such as JAVA (see section 2.3.4) and C# (see section 2.3.3).

Asynchronous methods are those that never return a result, and any call to them is guaranteed to complete immediately. The process calling an asynchronous method neither executes its body nor blocks. Thus, invoking an asynchronous method is comparable to sending a message or posting an event.

Chords consist of a header, a conjunction of method declarations, and a body that is only executed when all the methods in the header have been called. For example, a countdown latch can be written in SCHOOL (see section 2.3.5) as follows:

```
1  class CountdownLatch{
2    void await(int permits) & async countDown(){
3      if (-- permits > 0) await(permits);
4    }
5  }
```

Listing 2.1: Countdown latch written in SCHOOL

In the `CountdownLatch` class, two instance methods – the synchronous `await()` and the asynchronous `countDown()`, are jointly defined in a single *chord*. A countdown latch is used to stop a thread until a specific number of tasks have completed. That thread initially calls `await()`, passing the number of tasks, and the worker threads call `countDown()` when these tasks have completed.

Usually, in most languages, the body of the *chord* will be executed by the same thread calling `await()`. Nonetheless, when a *chord* does not define any synchronous method, either a new thread will be spawned to service this call or a worker from some pool will be used.

Asynchronous methods and *chords* bring the shared-memory paradigm closer to the message passing approach, where communicating by exchanging asynchronous messages is central. Moreover, if we compare the SCHOOL code in listing 2.1 with an equivalent JAVA version in listing 2.2, we observe that the former is much more succinct and expressive. The syntax to create asynchronous methods in *chords* is also clearer and more concise than delegates in .NET languages or Future Tasks in JAVA, as they normally only require to be preceded by a keyword, such as `async`.

Making asynchronous communication clearer and easier is also very valuable given that asynchronous events are increasingly used at all levels of software systems and that the focus is shifting from shared-memory concurrency to message-oriented concurrency.

```
1  class CountdownLatch{
2    int permits;
3
4    void await(int permits) throws InterruptedException{
5      synchronize(this){
6        this.permits = permits;
7        wait();
8      }
9    }
10
11   void countDown(){
12     synchronize(this){
13       if(--permits == 0){
14         notifyAll();
15       }
16     }
17   }
18 }
```

Listing 2.2: Countdown latch without chords in Java

2.1.2 Join Calculus for Message Passing

In the message passing approach, inter-process communication relies on asynchronous message passing. Usually, each process has a single mailbox containing a queue of received messages which is analysed by applying pattern matching on the contents of the messages.

In this paradigm, *chords* usually receive the name of *join patterns* or simply *joins*. Languages that support *joins* include JoCaml (see section 2.3.2) and Jerlang (section 2.3.1). Listing 2.3 shows how *joins* can be used for synchronisation in Jerlang. More precisely, a new aggregate of the form {`found, X`} is created only when two messages matching the patterns {`get, X`} and {`set, X`} are received.

```
1  receive {get, X} and {set, X} -> {found, X} end
```

Listing 2.3: A one-cell buffer in Jerlang

We can compare the previous program with the program in listing 2.4 that presents a similar behaviour but does not use *join patterns*. The latter synchronises on the same type of messages, but does not preserve the order in which those are received. More importantly, the version in 2.4 is notably larger and harder to write, which increases the chance of making errors. Thus, *joins* greatly simplify the task of synchronising on the reception of asynchronous messages, and become a powerful and clean construct for message passing languages.

```
1   A = fun(ReceiveFunc) ->
2     receive
3       {get, X} ->
4         receive
5           {set, Y} when (X == Y) -> {found, X}
6         after 0 -> self() ! {get, X}, ReceiveFunc(ReceiveFunc) end
7       {set, X} ->
8         receive
9           {get, Y} when (X == Y) -> {found, X}
10        after 0 -> self() ! {set, X}, ReceiveFunc(ReceiveFunc) end
11      end,
12  A(A)
```

Listing 2.4: Synchronisation on two messages in ERLANG

2.2 Join Calculus

Similar to the Lambda Calculus, which was developed to model and analyse the behaviour of sequential systems, several process algebra or calculi have been defined to describe the behaviour of concurrent systems. Amongst them, the JOIN-CALCULUS is a process algebra based on the reflexive chemical abstract machine (RCHAM), a model of concurrency that exploits the similarities between chemical reactions and concurrent execution of processes.

The JOIN-CALCULUS has the same expressive power as the π-CALCULUS, a well studied process algebra, and full abstract encodings of each other exists in each direction [15]. Like many of the other process algebras, the JOIN-CALCULUS has influenced the development of programming languages and is considered the basis of the *multi-way join pattern* construct.

2.2.1 Process algebras

Process calculi or algebras are a formal approach for reasoning about concurrent systems. Process calculi model the interactions, communications and synchronisations between a set of independent agents or processes and give rise to model-checking tools for proving properties about concurrent systems.

In 1978, Tony Hoare published *Communicating Sequential Processes*[18] (CSP), a paper which essentially presented a programming language of the same name for describing patterns of interaction in concurrent systems that did not possess mathematically defined semantics. CSP provided a command based on Dijkstra's *parbegin* to specify concurrent execution, and communication between processes was based on synchronous message passing. In this version of CSP, both sender and receiver processes had to name each other as source and destination for the communication to take place.

But CSP gradually evolved into a process algebra, and the book *Communicating Sequential Processes*[1] published in 1984 presents the theoretical foundations of the calculus.

[1] Available from http://www.usingcsp.com/

This evolution was influenced by the Calculus of Communicating Systems (CCS) which was developed by Robin Milner. Milner originally published in 1980 a book of the same name [30] where he described the syntax, semantics and observation congruence of CCS. CCS allows one to express parallel composition of processes, non-deterministic choice, scope restriction and value passing. With the help of Hennesy-Milner Logic [17] properties such as safety, fairness or liveness are analysable.

As a continuation of the work on CCS, Robin Milner, along with Joachim Parrow and David Walker, developed the π-Calculus [31]. The π-Calculus can model the changing connectivity of interacting systems and forms the basis of many languages supporting distributed concurrent programming. The π-Calculus is more expressive than CCS because it provides channel mobility (channel names can be sent and received) and restriction (new private channels can be generated).

Similarly expressive when compared with the π-Calculus is the Join-Calculus. The Join-Calculus is an alternative representation of the reflexive CHAM, which extends the chemical abstract machine (CHAM) of G. Berry and G. Boudol [9] with reflexion and locality. The CHAM is based on the chemical metaphor used in the Γ language of Banâtre and Le Métayer [4] and is suited to model concurrent computations, as presented in the next section.

2.2.2 Introduction to the Reflexive CHAM and the Join-Calculus

Similarly to Turing machines for sequential problems, the chemical abstract machine CHAM of Bery and Boudol originated to reason about concurrent problems [9], and provides an expressive and meaningful abstraction where concurrent processes are modelled as chemical reactions. The reflexive CHAM model (RCHAM) is obtained from the generic CHAM by imposing locality and adding reflexion [15].

In the RCHAM, the state of a system is viewed as a chemical solution $\mathcal{R} \vdash \mathcal{M}$ in which molecules and atoms can interact with each other according to certain reaction rules. The multiset $\mathcal{M}$ represents the processes (molecules) running in parallel, and the reactions $\mathcal{R}$ define the current reaction rules. An atom is of the form $x\langle y\rangle$ and represents a pending message that transmits the value y trough port x. A molecule is a composition of several atoms or sub-molecules, joined with the operator "|".

Molecules can be reversibly created/divided using structural equivalence, represented by $\rightleftharpoons$. Chemical reactions $J \triangleright P$ are non-reversible and consume molecules that match the join pattern J to create new molecules P. In the following example, atoms $ready\langle laser\rangle$ and $job\langle 1\rangle$ are merged to form the molecule $ready\langle laser\rangle \mid job\langle 1\rangle$, and then the pattern in the reaction $ready\langle printer\rangle \mid job\langle file\rangle \triangleright printer\langle file\rangle$ is matched to reduce the molecule $ready\langle laser\rangle \mid job\langle 1\rangle$ to $laser\langle 1\rangle$:

$$
\begin{array}{rl}
 & ready\langle printer\rangle \mid job\langle file\rangle \triangleright printer\langle file\rangle \vdash ready\langle laser\rangle, job\langle 1\rangle, job\langle 2\rangle \\
\rightleftharpoons & ready\langle printer\rangle \mid job\langle file\rangle \triangleright printer\langle file\rangle \vdash ready\langle laser\rangle \mid job\langle 1\rangle, job\langle 2\rangle \\
\longrightarrow & ready\langle printer\rangle \mid job\langle file\rangle \triangleright printer\langle file\rangle \vdash laser\langle 1\rangle, job\langle 2\rangle
\end{array}
$$

The molecule $def\ D\ in\ P$ is structural equivalent to a reaction D and a molecule P. This type of molecule allows to introduce new reactions dynamically and as a result makes the RCHAM model reflexive. For example:

$$\begin{aligned} & \varnothing \vdash def\ D\ in\ ready\langle laser\rangle \mid job\langle 1\rangle \mid job\langle 2\rangle \\ \rightleftharpoons \quad & D \vdash ready\langle laser\rangle \mid job\langle 1\rangle \mid job\langle 2\rangle \end{aligned}$$

The syntactic description of the RCHAM molecules is known as the Join-Calculus. The RCHAM entirely defines the syntax (molecules), the structural congruence ($\rightleftharpoons$) and the reduction relation ($\rightleftharpoons^* \longrightarrow \rightleftharpoons^*$) of the Join-Calculus. The Join-Calculus is an alternative representation of the RCHAM model and can be defined as a rewriting system modulo structural equivalence.

2.2.3 RCHAM formally

As shown in figure 2.1, the syntax of the reflexive CHAM defines processes, join-patterns and definitions. A process P is either the asynchronous emission of a polyadic message $x\langle\tilde{v}\rangle$, a definition of port names or a parallel composition of processes. A definition D consists of a number of reaction rules of the form $J \triangleright P$ connected by the $\wedge$ operator. The reaction rules match join-patterns of messages J to trigger the processes P (see section 2.2.2).

$P \stackrel{def}{=}$		(Processes)
	$x\langle\tilde{v}\rangle$	(asynchronous message)
	$def\ D\ in\ P$	(local definition)
	$P \mid P$	(parallel composition)
$J \stackrel{def}{=}$		(Joins)
	$x\langle\tilde{v}\rangle$	(message pattern)
	$J \mid J$	(pattern conjunction)
$D \stackrel{def}{=}$		(Definitions)
	$J \triangleright P$	(reaction rule)
	$D \wedge D$	(composition)

Figure 2.1: Grammar of the reflexive CHAM

Analogously to the π-Calculus, names are the only values defined and can only be bounded in a join pattern J. The formal parameters $v_1, v_2, \ldots v_n$ received in join patterns are bounded in the corresponding processes P. As an example, after reducing the molecule $def\ x\langle v\rangle \triangleright P\ in\ x\langle y\rangle$ the name y will substitute v in P. More formally, the operational semantics of the RCHAM is given in figure 2.2.

The defined variables dv are those that appear as a channel in a message pattern. On the other hand, the received variables rv are those that appear as a message. The substitution σ_{dv} in the rule **str-def** introduces fresh variables to replace the defined variables in

$$
\begin{array}{lrcll}
\textbf{str-join} & \vdash P_1 \mid P_2 & \rightleftharpoons & \vdash P_1, P_2 & \\
\textbf{str-and} & D_1 \wedge D_2 \vdash & \rightleftharpoons & D_1, D_2 \vdash & \\
\textbf{str-def} & \vdash def\ D\ in\ P & \rightleftharpoons & D_{\sigma_{dv}} \vdash P_{\sigma_{dv}} & (\text{range}(\sigma_{dv})\ \text{fresh}) \\
 & & & & \\
\textbf{red} & J \triangleright P \vdash J_{\sigma_{rv}} & \longrightarrow & J \triangleright P \vdash P_{\sigma_{rv}} &
\end{array}
$$

Figure 2.2: Operational semantics of the reflexive CHAM

D. Similarly, the substitution σ_{rv} in the reduction rule **red** replaces the received variables as shown for the molecule $def\ x\langle v\rangle\ \triangleright\ P\ in\ x\langle y\rangle$ in the previous paragraph.

The reflexive CHAM can express common notions in concurrency such as replication, message forwarding and non-deterministic choice:

- **Message Forwarding** - $def\ x\langle y\rangle \triangleright y\langle u\rangle\ in\ P$ forwards messages on the local name x in P to the outside through y
- **Non-determinism** - $def\ s\langle\rangle \triangleright P \wedge s\langle\rangle \triangleright Q\ in\ s\langle\rangle$ can reduce to either P or Q as both joins can be matched
- **Replication** - $def\ loop\langle\rangle \triangleright P \mid loop\langle\rangle\ in\ loop\langle\rangle$ replicates the process P producing a new copy every time a reduction occurs
- **Continuation** - $def\ x\langle v_1, v_2\rangle \triangleright\ v_2\langle y\rangle\ in\ P$ returns a result y through the given channel v_2

2.3 Languages with support for Join-patterns

The π-Calculus have influenced languages such as Pict [34], and Occam [20] was heavily influenced by CSP. Similarly, the Join-Calculus has guided extensions to many programming languages due to the simplicity of its *multi-way join pattern*. In this section we cover some of the languages for which *join patterns* have been implemented, either as a library or as a language extension.

2.3.1 JErlang

JErlang is a Join-Calculus inspired extension of Erlang developed by H. Plocinicak and S. Eisenbach at Imperial College London [35]. Erlang, an industrially successful functional language developed at Ericcson to takle telecom problems[2], provides an Actor model paradigm for concurrency that relies on message passing for communication. JErlang overcomes the limitations of this paradigm by providing a new construct, a *join*, that facilitates the synchronisation between multiple processes without the need for shared memory.

[2]http://ftp.sunet.se/pub/lang/erlang/index.html

In ERLANG, processes are created with the **spawn** statement and inter-process communication relies on asynchronous message passing. The operator **!** sends a message with value *m* to the process with id *pid*: `pid ! m`. Each process has a single mailbox containing a queue of received messages which is analysed using the *selective receive* construct that allows to perform pattern matching on the contents of the message as shown below:

```
1  receive
2    {ok, Val1, Val2} when {Val1 == ok} -> Expr1;
3    {ok, Result} -> Expr2;
4    {error, Error} -> Error_Expr
5  after Timeout -> Timeout_Expr end
```

Listing 2.5: Selective Receive

JERLANG *joins* facilitate the synchronisation of messages on a single mailbox. Synchronisation is achieved by extending the syntax of the **receive** construct to allow multiple patterns concatenated with the **and** keyword:

```
1  receive {get, X} and {set, Y} when (X == Y) -> {found, X} end
```

Listing 2.6: Synchronisation on two messages with guards in JERLANG

Listing 2.6 shows how we can synchronise on two messages in the mailbox that match the pattern {get, X} and {set, Y}, where X and Y are equal. In this example, guards are used to compare the variables X and Y, but a more elegant solution would make use of non-linear patterns, which allow to use the same unbound variables on multiple patterns and synchronise on their values, as shown in listing 2.3 in section 2.1.2.

The implementation of **receive** in JERLANG assumes the semantics of *First-Match* regarding the contents of the mailbox. In listing 2.7, given the messages at line 1, the pattern at line 4 is satisfied before the one at 3. Moreover, as figure 2.8 shows, the *First-Match* semantics is extended to the order in which *joins* are evaluated, so that more specialised *joins* should appear before more general ones.

```
1  self() ! {foo, one}, self() ! {error, 404}, self() ! {bar, two},
2  receive
3    {foo, A} and {bar, B} -> {error, {A,B}};
4    {error, Reason} -> {ok, {error_expected, Reason}}
5  end
```

Listing 2.7: Impact of First-Match semantics on join

```
1  receive
2    {foo, A} and {bar, specific} -> ... %% specialised action
3    {foo, A} and {bar, B} -> ... %% general action
```

Listing 2.8: Usage of deterministic behaviour for joins resolution in JERLANG

To note is that the timeouts in listing 2.5 introduced with the **after** keyword are also supported for *joins*, which keeps the design of JERLANG consistent with the original ERLANG language. Equally synchronous calls are created as in ERLANG, by appending a process identifier value to the message so that the receiver knows with whom to communicate.

Nonetheless, JERLANG introduces the optional feature of propagation that allows to keep a message in the mailbox maintaining its original position after being used in a pattern, feature that is not present in the original ERLANG. By wrapping a pattern with the **prop** closure, any message matching that pattern will not be removed from the mailbox and as a result could be reused by other *joins*. In listing 2.9 any message matching the pattern {session, Id} at line 2 may be reused at 3.

```
1  receive
2    prop({session, Id}) and {action, A, Id} -> doAction(A, Id);
3    {session, Id} and {logout, Id} -> logout_user(Id)
4  end
```

Listing 2.9: Propagation in JERLANG

JERLANG has been implemented in two different forms, as a library for use with the standard ERLANG language and as a system with compiler and VM changes. In both cases, synchronisation on multiple messages required that these should not be removed immediately from the mailbox, but only when all the patterns in a *join* have been satisfied. In the case of the library, it maintains an internal queue where messages from the VM's mailbox are fetched in order. The solution for the system with compiler and VM modifications is more subtle.

A separate *search pointer* on the mailbox was introduced, independent from the original mailbox's pointer, which allows to search the mailbox and remove messages in a different order from the original ERLANG semantics. To deal efficiently with large mailboxes, each JERLANG process contains a hash table inspired on the *uthash* hash tables[3] that maps identifiers to the addresses of the messages.

For each pattern in a *join*, the compiler creates an anonymous function to test if it has been satisfied. To avoid unused variable warnings by the compiler when performing isolated tests for each of the patterns in a join, a reaching definition analysis is performed. Likewise, a live variable analysis avoids unbound variable errors.

To improve the performance of the *join solver*, a variant of the RETE algorithm, common in *Production Rule Systems*, was used. The version used does not run the tests on all the patterns, but only on those which belong to the currently tested join. In addition, pruning of the search space was performed to avoid the computation associated with messages that have equal values to those previously satisfied.

The reordering of the computations performed by the *join solver* also resulted in performance gains. Bringing forward the checks corresponding to guards or processing patterns in descending rank order allowed filtering invalid message combinations sooner.

[3]http://uthash.sourceforge.net

The rank of a pattern depends upon the number of variables that shares with other patterns and guards.

2.3.2 Jocaml

JoCaml has been developed at INRIA, the French institute for Computer Science, as an extension of Objective-Caml (OCaml) that implements the Join-Calculus. To note is that both OCaml and the Join-Calculus were born at INRIA, and that the Join-Calculus was also a language that they created prior to JoCaml. The current version of JoCaml is a re-implementation by C.Fournet of the original now unmaintained language created by Fabrice Le Fessant [11]. This new version provides a cleaner syntax and better compatibility with the standard OCaml language [4].

JoCaml programs consist of *processes* and *expressions* [28]. The execution of a JoCaml process produces no result and is asynchronous, while the execution of an expression, as it occurs in OCaml, is synchronous and returns a result. JoCaml also adds the notion of *channels* (also known as *names* or *port names*) which are used by processes to send messages. Channels are treated as first order entities, and can be sent as the value of a message.

To create new channels the `def` binding is used as shown in listing 2.10. This definition creates a new non-blocking channel `echo` with type `int Join.chan`, meaning that it can carry values of type `int`. Whenever a message is sent on `echo`, a new instance of the process `print_int; x 0` is spawned. This process is asynchronous and does not returned any value since `; 0` discards the value created by `print_int x`.

```
1 # def echo(x) = print_int x; 0
2 # ;;
3 var echo : int Join.chan = <abstr>
```

Listing 2.10: Definition of a new channel in JoCaml

Processes are an addition introduce by JoCaml used mainly for communication and synchronisation tasks. To execute processes concurrently one needs to precede them with the `spawn` keyword, as for example:

```
1   # spawn echo(1)
2   # ;;
3   -: unit = ()
4   # spawn echo(2)
5   #;;
6   -: unit = ()
```

Listing 2.11: Spawning two asynchronous-send processes

The code in listing 2.11 spawns two processes whose function is to send an asynchronous message through channel `echo` (the syntax for message passing is equal to that

[4] http://jocaml.inria.fr/

of a method call). When this program runs, it could print either 12 or `21` given the asynchronous nature of processes. An alternative notation with the same results makes use of the parallel composition operator `&`:

```
1  # spawn echo(1) & echo(2)
2  #;;
3  -: unit = ()
```

Listing 2.12: Parallel composition operator

Channels can also take more than one argument, as shown in listing 2.13. If one of the arguments is a channel `f`, a synchronous channel can be created by returning a value on the given channel `f`. For instance, the `succ` channel in listing 2.14 is a channel that returns `x+1` after it has printed `x`. Consequently, when the process in the other end receives `x+1` it knows that `x` has been printed.

```
1  # def strange_echo(x,y) = echo (x+y) & echo (y-x)
2  # ;;
3  val strange_echo : (int * int) Join.chan = <abstr>
```

Listing 2.13: Channel taking multiple arguments

```
1  # def succ(x,k) = print_int x; k(x+1)
2  # ;;
3  val succ : (int * int Join.chan) Join.chan = <abstr>
```

Listing 2.14: Explicit continuation

Notwithstanding, JoCaml provides a more convenient notation to create synchronous channels. To define a process that returns a value, a functional type has to be given to the process, and a `reply to` construct used, as demonstrated in 2.15.

```
1  # def succ(x) = print_int x; reply x+1 to succ
2  # ;;
3  val succ : int -> int = <fun>
```

Listing 2.15: Synchronous channel

Definitions can also take a *join pattern* in the place of the defined channel. A join pattern is a definition of multiple channels, joined with the parallel composition operator `&`, that can be used to enforce the synchronisation of messages on these channels. In example 2.16, to execute the process `print_endline(f^ " " ^ c) ; 0`, messages must be sent on both `fruit` and `cake`.

```
1   # def fruit(f) & cake(c) = print_endline (f^`` ''^c) ; 0
2   # ;;
3   val fruit : string Join.chan = <abstr>
4   val cake : string Join.chan = <abstr>
```

Listing 2.16: Join pattern in JoCaml

In JoCaml, join patterns exhibit a non-deterministic behaviour whenever possible. For the previous example, given spawned processes in listing 2.17, which two cakes will appear on the console is not clear. "Apple crumble" and "raspberry pie" could be a valid answer, but both combinations of fruits and cakes are correct.

```
1 # spawn fruit(``apple'') & fruit(``raspberry'')
2       & cake(``pie'') & cake(``crumble'')
3 # ;;
4 - : unit = ()
```

Listing 2.17: Non-deterministic join resolution

Equally, when the same channel is used in more than one join, as `pie` in 2.18, there can be situations where both patterns are satisfied, but which one gets fired is undefined. For example, if the messages `apple()`, `raspberry()` and `pie()` have been sent, both "apple pie" and "raspberry pie" are valid answers. To note is that the operator `or` is used to define multiple join patterns in one definition.

```
1   # def apple() & pie() = print_string ``apple pie'' ; 0
2   # or raspberry() & pie() = print_string ``raspberry pie'' ; 0
3   # ;;
4   val apple : unit Join.chan = <abstr>
5   val raspberry : unit Join.chan = <abstr>
6   val pie : unit Join.chan = <abstr>
```

Listing 2.18: Multiple join patterns in one definition

The argument of a channel can also be an algebraic pattern that will also be matched against during the join pattern resolution. In the following example, whether "empty" or "filled" will print will depend on whether the list passed as an argument of `isEmpty()` is empty or contains some elements.

```
1   # def isEmpty([]) & doPrint() = echo_string(``empty'')
2   # or isEmpty(x::xs) & doPrint() = echo_string(``filled'')
3   # ;;
4   val isEmpty : string Join.chan = <abstr>
5   val doPrint : unit Join.chan = <abstr>
```

Listing 2.19: Patterns used as arguments of channels

One has to be careful when using pattern matching in join patterns because, in contrast with plain OCaml, ambiguous patterns are indeed ambiguous. For example, if in the

previous example we use the pattern _ (matching any value) instead of `x::xs`, an empty list would match both `[]` and _, so whether "empty" or "filled" would print is undefined.

Join patterns can be made of any combination of synchronous and asynchronous methods. For instance, JoCaml allows to create joins with more than one synchronous channel, like the barrier in example 2.20 that defines a synchronisation point in the execution of two parallel tasks.

```
# def join1 () & join2 () = reply to join1 & reply to join2
# ;;
val join1 : unit -> unit = <fun>
val join2 : unit -> unit = <fun>
```

Listing 2.20: Parallel tasks synchronisation barrier

Support for distributed programming is also built-in into the language. The JoCaml name-server allows to exchange channel names between programs executed on different machines that do not initially share any port name. A server registers a resource using the `Join.Ns.register` function, and the client can look up its value with `Join.Ns.lookup` as shown in listing 2.21.

```
(* SERVER *)
# spawn begin
# def f (x) = reply x*x to f in
# Join.Ns.register Join.Ns.here "square" (f: int -> int); 0
# end
# ;;
- : unit = ()

(* CLIENT *)
# spawn begin
# let sqr = (Join.Ns.lookup Join.Ns.here "square" : int -> int) in
# print_int (sqr 2); 0
# end
# ;;
- : unit
```

Listing 2.21: Built-in support for distributed programming

2.3.3 Polyphonic C#

Polyphonic C# is an extension of C#, one of the .NET languages developed by Microsoft. C# is an object-oriented language that relies on mutual exclusion based mechanisms for coordination between threads. To create asynchronous methods, C# provides delegates, type-safe objects that can point to one or more methods which can be invoked at a later time.

Nonetheless, there is not a direct connection between delegates and the synchronisation mechanisms in C#, and writing concurrent applications can be unnecessarily hard. POLYPHONIC C# on the other hand extends C# with *chords* and asynchronous methods in order to facilitate concurrent programming [8], as introduced in section 2.1.1.

In POLYPHONIC C# asynchronous methods are declared with the keyword `async` and the methods in the signature of a chord are joined together with the `&` symbol. Moreover, a single *chord* can contain at most one synchronous method. Listing 2.22 provides a simple example where a chord is used in a one-cell buffer.

```
1  public class Buffer {
2    public string Get() & public async Put(string s) {
3      return s;
4    }
5  }
```

Listing 2.22: One cell-buffer in POLYPHONIC C#

The thread calling `Get()` (a synchronous method) on an object of type `Buffer` will execute the body of the *chord*, but only when a corresponding call to `Put()` (an asynchronous method) has occurred. Otherwise, it will block until one occurs. On the other hand, the thread calling `Put()` does not block, but unmatched calls to `Put()` are queued. To note is that if a *chord* does not define a synchronous method then the body is run in a new thread.

When more than a single call to each of the methods defined in a *chord* have occurred, exactly which calls are pair together is unspecified, so for example the following program (that uses the `Buffer` of listing 2.22) could print either "bluesky" or "skyblue".

```
1  Buffer buff = new Buffer();
2  buff.Put(``blue'');
3  buff.Put(``sky'');
4  Console.Write(buff.Get() + buff.Get());
```

Listing 2.23: Non-deterministic match up of calls in POLYPHONIC C#

Non-determinism can also be present when the same method is used in two different *chords*. In listing 2.24 if a call to `Get()` happens when calls to both `Put(string s)` and `Put(int n)` have been queued, exactly which *chord* will be reduced is unspecified.

Regarding polymorphism and inheritance, *chords* introduce a number of restrictions. For example, if we were to overwrite the method `Put(string s)` in a subclass of `Buffer`, we would be forced to also overwrite the methods `Get()` and `Put(int n)` in the subclass. Otherwise, we could have introduced a deadlock situation, where a *chord* can never be reduced. Consider, for example, the program in listing 2.25.

```
1  public class Buffer {
2    public string Get() & public async Put(string s) {
3      return s;
4    }
5    public string Get() & public async Put(int n) {
6      return n.ToString();
7    }
8  }
```

Listing 2.24: Non-deterministic buffer in Polyphonic C#

In the invalid program in listing 2.25, all the calls to `g()` on an object of class `D` would cause `body2` to run and calls to `f()` on the same object would deadlock forever. This behaviour would be particularly problematic when an instance of class `D` is passed to a fragment of code expecting an object of type `C` as the chances are that it would deadlock. As a result, a *chord* has to be created with either virtual or non-virtual methods but not both, and a subclass has to either overwrite all the methods in a chord or none. But notice that both restrictions are enforced by the compiler.

```
1  class C {
2    virtual void f() & virtual async g() { /* body1 */ }
3  }
4  class D : C {
5    override async g() { /* body2 */ }
6  }
```

Listing 2.25: Invalid class inheritance in Polyphonic C#

The compilation process involves translating a Polyphonic C# class into a plain C# class. The resulting class has the same name and signature as the source class, plus additional private state and methods that handle the required synchronisation which is statically defined by the *chords*. Essentially, synchronisation is compile down into a state automaton, as presented by Le Fessant and Maranget [24].

Moreover, the `async` declarations are mapped into `void` declarations. Polyphonic C# treats `async` as a subtype of `void`, but the behaviour of both is quite different. When translated, `async` methods are prefixed with the attribute `[OneWay]`. This is a .NET attribute that indicates that calls to a method should be non-blocking. In the `Buffer` example of listing 2.24, the method `public async Put(string s)` translates to `[OneWay] public void Put(string s)`.

The additional state to deal with synchronisation consists of the pending calls for all the methods that appear in a *chord*. For synchronous methods thread information is stored, and for asynchronous methods the messages. Nonetheless, this information is only used when a chord has fired.

To determine when a chord has fired a bitmap is used. This bitmap contains a single bit that records the presence of one or more pending calls for each method, and is compared against constant bitmasks, one for each *chord*. For a given *chord*, the corresponding bitmask will have a bit set for every method appearing in that *chord*. Consequently, checking whether a *chord* has been satisfied is very efficient.

To ensure the thread safety of the translated code, a single auxiliary lock protects the private synchronisation state of each object. Locking occurs briefly during each incoming call and involves a separate lock for each polyphonic object. But this lock is independent of the regular object lock, which may be used as usual to protect the rest of the state and prevent race conditions while executing *chord* bodies.

2.3.4 Join Java

G.Stewart von Itzstein from the University of South Australia developed JOIN JAVA as part of his doctoral thesis [5]. This language extends JAVA with asynchronous methods and *chords* [21], and has many similarities with POLYPHONIC C# (the reader is advised to read section 2.3.3 before continue).

Similarly to POLYPHONIC C#, in JOIN JAVA a *chord* can have at most one synchronous method, possibly zero. If all the methods in a *chord* are asynchronous, then its body is executed in a new thread. A syntactic difference is that the return type of an asynchronous method is `signal`, not `async`.

Furthermore, in JOIN JAVA the synchronous method can only appear as the first element and not in any other position in the *chord*. The authors claimed that this restriction helps with the readability of the code as it makes more obvious to the programmer what the synchronisation behaviour of the *chord* is.

JOIN JAVA allows the user to specify if the matching of *chords* is sequential or non-deterministic by adding a modifier to the class declaration. If we consider the `Buffer` class of listing 2.24, we can write an equivalent JOIN JAVA program as follows:

```
1  public class unordered Buffer {
2    public String Get() & public signal Put(String s) {
3      return s;
4    }
5    public String Get() & public signal Put(int n) {
6      return n.ToString();
7    }
8  }
```

Listing 2.26: Non-deterministic buffer in JOIN JAVA

This buffer exhibits a non-determistic behaviour because it has been qualified with the `unordered` modifier, so when the two *chords* are satisfied the choice of which will run is non-deterministic. In this case, the modifier could be omitted as `unordered` comes

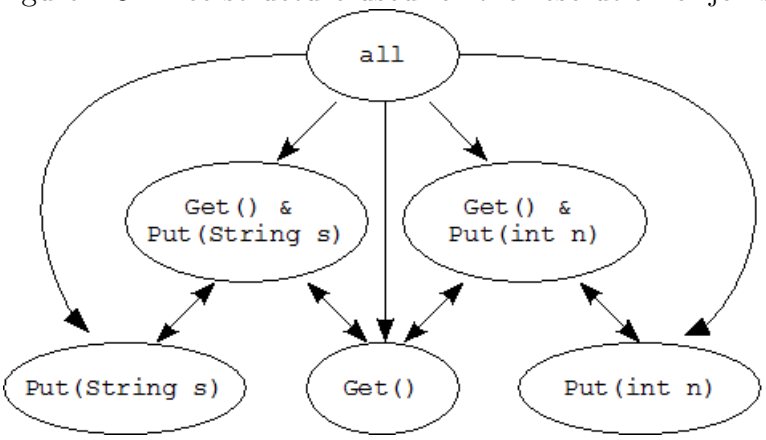
[5] A draft of the thesis can be found at http://Join Java.unisa.edu.au/test.htm

by default. If we had used `ordered` instead, when the two *chords* are satisfied, we will always respect the order in which they are defined within the class. Precisely, whenever `Put(String s)` and `Put(int n)` are called prior to `Get()`, the first *chord* is always the one fired.

Another difference with POLYPHONIC C# relates to how JOIN JAVA handles inheritance. In this language, inheriting from classes containing *chords* is not permitted, and those classes are treated as if they were `final`. This approach to solve the *inheritance anomaly* problem described by Matsuoka [29] is much more extreme and restrictive than the solution proposed by POLYPHONIC C#, where inheriting from classes containing *chords* is still possible but with some restrictions. The JOIN JAVA solution certainly keeps the implementation simpler, but does that at the cost of forbidding, for classes containing *chords*, one of the most defining characteristics of object-oriented programming, ie. inheritance.

Regarding the resolution of *join patterns*, a tree structure is used to represent the static structure of the patterns of a *Join* class. In this tree, the interior nodes represent the patterns and the leaves represent the individual methods, as shown below for the example of listing 2.26. The run time search space is reduced by only having to check the *chords* containing a given method. In addition, it avoids the state explosion problem associated with a state machine based solution, as presented in [24]. It is also more scalable than using bitmaps, as these are limited by the predefined maximum size of the bit-field.

Figure 2.3: Tree structure used for the resolution of joins



2.3.5 School

SCHOOL, the Small Chorded Object-Oriented Language, is a minimal language developed at Imperial College London by S. Drossopoulou, A. Petrounias, A. Buckley and S. Eisenbach to study chords formally in an object-oriented language. As such, the authors provided the formal operational semantics and type system for the language, and proved the soundness of the type system. SCHOOL only supports classes, inheritance and chords, but its extension SCHOOL+F also provides fields.

In SCHOOL a chord signature comprises at most one synchronous method (with return type) and any number of asynchronous methods (`async`), possibly 0. Multiple callers can coordinate by sending messages that, finally, lead to execution of a chord body when all the methods in the signature have been called. The caller of a synchronous method will block until the chord body executes, but asynchronous method callers will continue immediately. Hence, the receiving object must queue the asynchronous messages received until consumed by the joining of the chord. Listing 2.1 in section 2.1.1 shows how a countdown latch can be programmed in SCHOOL.

SCHOOL welcomes non-determinism whenever there is a choice, and does not make any assumption about the order in which chords are consumed. If the method `cancel()` is invoked before `print()` in listing 2.27, `print()` could return either "printed" or "cancelled", and no assumptions can be made about what value will actually return.

```
1  class Printer{
2    String print(){
3      return ``printed'';
4    }
5    String print() & async cancel(){
6      return ``cancelled'';
7    }
8  }
```

Listing 2.27: School non-deterministic chord resolution

2.4 Thorn

THORN [6] is a concurrent object oriented language in which lightweight isolated processes communicate by message passing. THORN supports the gradual evolution of dynamically-typed scripts into robust programs [32] and is currently being developed jointly by Purdue University and IBM Research T.J. Watson Research Center.

Currently there exists both an interpreter and a compiler for THORN. The interpreter, not particularly fast, is written in Java and is used to prototype constructs before implementing them in the compiler. The compiler has been designed to accommodate the evolution of the language itself through a plugin mechanism and targets the Java Virtual Machine.

THORN has been designed for domains such as client-server programming, embedded event-driven applications or distributed web services. It is particularly well-suited to support the rapid prototyping of concurrent applications, that can evolve into scalable, robust programs. As such, it provides an expressive module system and a flexible type system that enable the programmer to follow good software engineering practises. The main features of the language are covered in more detail below as presented in [10].

[6] http://www.thorn-lang.org/

2.4.1 Classes

THORN *classes* are similar to those of JAVA and C++, and they statically determine the structure of objects. In contrast with some scripting languages, adding or removing fields from objects dynamically is not permitted. Classes can take parameters that will generate member fields, as it occurs in SCALA.

```
1  class Point(x,y){}
```

Listing 2.28: Class declaration in THORN

The `Point` class could be equivalently defined with an explicit constructor, as in listing 2.29. Constructors are preceded with the `new` keyword, and might not refer to `this`. Disallowing the use of `this` in the constructor avoids leaking references to not fully initialised objects, although references to the objects referred by `this` can be passed from `init()`, method that if present is immediately called after the body of the constructor has been executed.

```
1  class Point{
2    val x; val y;
3    new Point(x', y') {x = x'; y = y';}
4  }
```

Listing 2.29: Class declaration with explicit constructor

Methods are preceded by the `def` keyword, and default setter and getter methods are provided for all instance variables, which are not visible outside its definition. As such, all accesses to fields are executed by a method call, and the expression `x.f` for object `x` and field `f` is just syntactic sugar for the method call `x.f()`.

Multiple inheritance is permitted, although with some restrictions. In 2.30, the class `TastyPoint` extends both `Point` and `Flavour`. If both `Point` and `Flavour` had a method with the same name and arity, `TastyPoint` would have to overwrite that method. Otherwise, either which method would be called from `TastyPoint` is unclear or some complex precedence rules would have to be introduced. Analogously, multiple inheritance of mutable state is forbidden. In other words, if classes `B` and `C` both have a `var` field `a`, another class cannot inherit from both of them.

```
1  class Flavour(fl) {}
2  class TastyPoint(x,y,fl) extends Point(x,y), Flavour(fl) {}
```

Listing 2.30: Multiple inheritance

2.4.2 Pattern Matching

THORN pattern matching capabilities are very powerful and provide a number of constructs that allow to create complex patterns. Those patterns can be used to match against the expected structure of the data and to extract the required data from objects and built-in data structures. Most built-in data types provide patterns analogous to their constructors. As an example, a list can be created with `[h, t...]`, expression that can also be used as a pattern to match a non-empty list, biding `h` to the first element and `t` to the tail. Other patterns include:

- `+p`, pronounced positively p, matches a non-null value that matches `p`
- `$(e)` matches the value of the expression `e`
- `(e)?` matches if the boolean expression `e` returns true
- `e:t` matches if expression `e` has type `t`
- `!p` succeeds iff `p` fails. This is known as the negative pattern.

Conjunctive patterns (`p && q`) and disjunctive patterns (`p || q`) allow to build more complex patterns. For instance, the pattern `r && {: source:$(sender) :}` matches a record whose source field is equal to `sender` and binds it to `r`. On the other hand, `[(3 || !(_:int))...]` matches lists whose integral elements are all 3, such as `[3, true]`.

Patterns are used in a variety of control structures. The `match` construct selects which clause to execute depending on which pattern a given value satisfies. For example, `match` can be used to write a function that sums the elements of a list:

```
1  fun sum(lst){
2    match(lst){
3        [] => {0;}
4      | [h,t...] => {h + sum(t);}
5  }}
```

Listing 2.31: Pattern matching used by the `match` construct

The same function may be written by applying pattern matching on the arguments (see listing 2.32). Patterns are also used in the expression `exp ~ pat`, that returns true if the value of the expression `exp` matches the pattern `pat`. The ~ operator can be used in the conditionals of `if`, `while` or `until` statements. The bindings created by ~ are available in the `then` branch of an `if` statement and in the body of the `while` loop. On the other hand, `until` loops can produce bindings after the loop.

```
1  fun sum([]) = 0
2   | sum([h,t...]) = h + sum(t);
```

Listing 2.32: Function definition applying pattern matching on arguments

2.4.3 Built-in Data Types

THORN provides many convenient built-in data types and syntax to initialise and pattern match on them:

- *Strings* are immutable and encoded using Unicode. The operator `$` interpolates values into strings: `x = ''John''; ''Dear $x''` evaluates to "Dear John".
- *Lists* are immutable ordered collections. `[]` represents the empty list and `[x,y...]` a list whose head element is `x` and tail is `y`. If `lst` is a variable holding a list, `lst(i)` gets the element at position i.
- *Records* are collections of immutable named fields. {`:name`$_1$`:value`$_1$`, name`$_2$`:value`$_2$`:`} is used both to construct a record and as a pattern.
- *Tables* are also built-in into THORN. They are equivalent to database tables and are mutable. Tables can be updated using built-in methods.

2.4.4 Pure data

A crucial distinction in THORN is made between mutable and immutable data. Avoiding mutable share data undoubtedly avoids many concurrency related problems, such as race conditions. Moreover, it avoids having to ensure that multiple copies of the same piece of data are kept consistent. Consequently, THORN only allows the communication of *pure* data between components. Pure data can only be constructed out of immutable values:

- Strings and numbers are always pure
- A variable is declared mutable with the `var` keyword, and the operator `:=` is used to assign new values to it. On the other hand, `val` and `=` are used for immutable variables
- List and records are pure if their fields are pure
- Tables are never pure
- Classes may be constraint and annotated as `pure` to make their instances pure. A pure class can have constant data but no mutable state. Nonetheless, it can be given mutable data to operate on, and it can use local mutable variables in its method bodies.

2.4.5 Components

Each THORN process, called a *component*, has a single strand of control and its own private data that does not share with other components. As such, all the communication between processes occurs exclusively by message passing. The values transmitted in a message have to be immutable and the receiver must have their class available if they are objects of a user-defined class (built-in types are universally understood).

Components are defined with the `component` construct. Similar to an object, a component may have local `val` and `var` data and functions `(fun)`, but also high-level communication operations (`sync` and `asyn` blocks). Each component defines a `body` containing the code that is run when spawned, as demonstrated below.

```
1  component LifeWorker{
2    var region;
3    async workOn(r) {region := r}
4    sync boundary(direction, cells) {...}
5    body{...}
6  }
```

Listing 2.33: Component definition

The `spawn` command starts a new component given its name or its body, and returns a handle that can be used to communicate with it. For example, in listing 2.34 `spawn` is used to start a new `LifeWorker` component and send it the asynchronous `workOn()` message.

```
1  c = spawn(LifeWorker)
2  c <-- workOn(r);
```

Listing 2.34: Spawning a component

2.4.6 High-level Communication

High-level communication provides named communication, with syntax similar to that of method calls. Synchronous communication is introduced with the `sync` keyword and asynchronous by `async` (see listing 2.33). An asynchronous message is sent with the `<--` operator, as seen in 2.34. On the other hand, to send a synchronous message `m` with parameter `x` to a component `comp` we use `comp <-> m(x)`. Sending a synchronous message returns the answer provided by `comp`.

In the `body` of a component, the `serve` statement can be used to wait for a *single* high-level communication event. Hence, a component that does something (`do_something`) indefinitely until it is told to quit can be defined as in example 2.35. Increasing the priority of `quit()` from the default 0 to 100 gives preference to `quit()` messages over `do_something()` when both have been received.

```
spawn{
  var done := false;
  async quit() prio 100 {done := true;}
  sync do_something() {...}
  body {while (!done) serve;}
}
```

Listing 2.35: Usage of `serve` and `prio`

2.4.7 Low-level Communication

THORN also provides a low-level communication model to send unadorned values from component to component. The statement `c<<<v` sends value `v` to component `c`. Each component has a mailbox to store the received messages, and these are retrieved using the `receive` statement, which scans the mailbox in descending order of priority. When a message that satisfies one of the patterns is found, its corresponding block of code is executed. In listing 2.36, if a record with an `stop` field has been received, the component stops. Otherwise it might post some data if the correct message has been received or will become bored after 10 seconds.

```
receive{
    {: stop: _ :} prio 1 => {return;} /* higher priority */
  | {: do: ``post'', data: x :} => {do_post(x);} /* prio 0 (default) */
  | timeout(10000) => {bored := true;}
}
```

Listing 2.36: Receive statement

```
component Receiver(){
  body{
    while(true){
      receive{
        "do_something" => {
          println("do_something");
        }
       | "kill" prio 10 => {
          println("kill");
          break;
        }
} } } }
```

Listing 2.37: Retrieval of messages with priority (`prio`)

It is important to emphasise how priorities (`prio`) influence the retrieval of messages from the mailbox. In the example in listing 2.37, if the mailbox of the `Receiver` component has received the messages "do_something", "do_something", "do_something", "kill" in this order, the next thing it would happen is that "kill" will get printed and the com-

ponent will terminate. The reason is that the pattern "kill" has higher priority and the message "kill" has been received, although it occupies the last position in the mailbox, but this fact is irrelevant.

Low-level communication is asynchronous in THORN and the sender of a message never blocks. However, a user can use this modality to program a synchronous application, as demonstrated in 2.38. In that program, the next thing the sender does after sending a query is wait for a reply. On the other hand, when the receiver gets a message, it immediately replies to the sender `s`, that can be retrieved by using the operator `from`.

```
1  receiver = spawn {
2    body{
3      while(true){
4        receive{
5          ``query'' from s => { s <<< ``reply'' ; }
6  } } } }
7  sender = spawn{
8    body{
9      while(true){
10       receiver <<< ``query'';
11       receive{
12         ``reply'' => { ... }
13 } } } }
```

Listing 2.38: Synchronous application using low-level communication

2.4.8 Type System

THORN presents an innovative approach to typing that integrates untyped and typed code and adds *like types*, an intermediate step between the two. This approach allows to evolve an initial dynamically-typed prototype into an efficient and robust statically-typed program. The default type in Thorn is `dyn`, which abbreviates dynamic and does not need to be written explicitly. Dynamic types can be substituted for *concrete types*, as used in statically typed languages, or by *like types*, that are an intermediate point between the two:

- If a variable is declared of type `like T`, static compiler checks ensure that the methods invoked on it belong to the interface of `T`.
- Binding and assignments are not checked statically as it occurs with concrete types, so like types require runtime checks to ensure that method calls are indeed valid.

Consider the class `Point` in example 2.39. In the method `move` we declare the formal parameter `p` to be of type `like Point`. This enables the compiler to throw a compile time error when we attempt to call `hog` on `p`, since `hog` does not belong to the interface of `Point`. Nonetheless, running `move` passing an instance of `Pair`, as defined in 2.40, will

succeed because `Pair` provides all the methods called by `move` making the runtime checks pass.

```
1  class Point(var x, var y){
2    def x() : int = x;
3    def y() : int = y;
4    def move(p: like Point){
5      x := p.x();
6      y := p.y();
7      # p.hog(); would raise compile time error
8    }
9  }
```

Listing 2.39: Like types in Thorn

```
1  class Pair(x,y){
2    def x() = x;
3    def y() = y;
4  }
```

Listing 2.40: Pair class

Like types help with code readability as they provide more information than dynamic types and nearly the same amount of flexibility. They also allow code completion and help with error detection, since for example mistyped method names can be easily detected.

2.4.9 Modules

Thorn's module system allows to encapsulate, package, distribute, deploy and link large programs. Similar to classes in Java, modules can import other modules and have state, as shown in listing 2.41. By default, a module exports all its defined members and hides all the members imported from other modules. Nonetheless, members can be hidden/re-exported by declaring them `private`/`public`.

An instance of a module can be either shared between multiple modules or local to a particular one. To import the shared instance of a module `M`, a second module should use `import M`. On the other hand, if we want to create a non-shared instance of `M` and call it A, `M` has to be imported with `import own A = M`.

```
1  module M{              # definition of module M
2    import N;            # shared instance of M
3    import own S = 0     # own instance of 0 as S
4    class A{}; val n = A();
5    var x: private = N.A(); # x not exported
6  }
```

Listing 2.41: Thorn module

2.4.10 De Bruijn index

The execution engine of THORN's interpreter accesses variables through their associated `Seal` object, which is a class providing all the information that THORN knows about a variable. Each occurrence of an identifier in the program needs a `Seal`, which can be shared with other occurrences of the same identifier. Essentially, the `Seal` of a variable is its *De Brujin index*.

The *De Bruijn index* [7] is a formal notation invented by the Dutch mathematician Nicolaas Govert de Bruijn for representing terms in the λ-calculus with the purpose of eliminating the names of the variables from the notation. Each De Bruijn index is a natural number that refers to a λ-abstracted variable. The index denotes the number of binders that are in scope between that occurrence and its corresponding binder.

For example, the term $\lambda\ \lambda\ 2$ corresponds to the K-combinator, written $\lambda x.\ \lambda y.\ x$ in standard λ-calculus notation. Similarly, the term $\lambda\ \lambda\ \lambda\ 3\ 1\ (2\ 1)$ corresponds to the S-combinator $\lambda x.\ \lambda y.\ \lambda z.\ x\ z\ (y\ z)$.

2.5 Summary

In this chapter, we have provided an overview of the two main paradigms for concurrent programming, ie. message passing and shared memory. We also gave an intuition on how *Join Calculus* inspired constructs can facilitate synchronisation. Sometimes, these constructs are termed *joins* in the message passing paradigm and *chords* in the shared memory paradigm, although in the literature they are also often used interchangeably.

We have also provided an introduction to *process algebras*, with special emphasis on the *Join Calculus*. Then, we reviewed some of the languages that support *Join Calculus* based constructs. Precisely, we covered JERLANG, JOCAML, POLYPHONIC C#, JOIN JAVA and SCHOOL. Finally, we presented a summary of the main features of the THORN language. With the exception of the type system, that allows to evolve dynamic scripts into typed programs, THORN does not introduce radically new features. What THORN achieves exceptionally is to group the best features of other languages into the same.

Classes are based on those in SCALA, instance variables are inspired by those in SMALLTALK, low-level communication comes from ERLANG, multiple inheritance is a simplify version of C++'s model, etc. All these features plus the actor model makes THORN a very powerful language to solve today's problems in Computing, specially in distributed environments. However, solving certain synchronisation problems in THORN can still be unnecessarily hard. In view of this, we created an extension, called JCTHORN, that incorporates *Join Calculus* based constructs. JCTHORN is presented in the next chapter together with its main features. In JCTHORN we also distinguish between joins and chords, although this language does not provide shared memory and belongs to the message passing paradigm. The main difference between the two lies on the level of communication, with joins belonging to the low level and chords to the high level.

[7] From http://en.wikipedia.org/wiki/De_Bruijn_index

Chapter 3

The language

This chapter introduces JCThorn to potential users of the language. Firstly, it demonstrates how some problems that cannot be easily solved in Thorn have a straightforward solution in JCThorn, section 3.1. Then, it describes in more detail the most visible extensions that JCThorn provides with respect to Thorn. These are *joins*, section 3.2, and *chords*, section 3.3. Finally, it presents the common features between both constructs (section 3.4) and describes a couple of situations where Thorn and JCThorn are not compatible (section 3.5). Before continue, the reader is advised to read the summary of the Thorn language in section 2.4.

3.1 Introduction

Solving a simple synchronisation problem like matching sellers and buyers of the same goods can be unnecessarily hard and error prone in Thorn. Even worse, a solution that only uses the high-level communication mode might not exists. As explained in sections 2.4.6 and 2.4.7 Thorn provides two levels of communication.

The low-level mode sends unadorned data between components and uses the `receive` construct to retrieve messages from the mailbox. In the other hand, in the high-level mode, each component declares a number of channels to which any component can address messages. To retrieve the messages sent on the channels, the `serve` statement is used. These high-level messages are also stored in the mailbox.

Sellers and buyers in the low-level communication mode

Returning to the problem of matching sellers and buyers, a first attempt at tackling this problem can be expressed using the low-level communication mode as shown in listing 3.1. In this server, the outer `receive` expects a message matching either the pattern {`:sell:X:`} or the pattern {`:buy:X:`}. The former matches any record that has a `sell` field and binds its value to the variable `X`, and the case for the latter is very similar.

We need both cases because we want to respect, as far as possible, the order in which the messages appear in the mailbox. For instance, the second rule will be triggered if a buy message appears first. In that case we will execute a second `receive` that will try to find a message satisfying the pattern {`:sell:$(X):`}. A pattern `$(e)` matches the value of the expression `e`, so {`:sell:$(X):`} will match a record whose `sell` field holds the value currently stored in variable `X`. The important factor to note here is that the `X` has been bound by the outer `receive`.

```
1  receive{
2
3    {:sell:X:} => {
4      receive{
5        {:buy:$(X):} => { /* pair found */ }
6    } }
7
8    | {:buy:X:} => {
9      receive{
10       {:sell:$(X):} => { /* pair found */ }
11 } } }
```

Listing 3.1: First attempt at matching sellers and buyers

A problem with this naive solution is that it might lead to a deadlock situation. For example, assume that the first matched message has been {`:buy:``Wisdom Teeth``:`}. Clearly, it might easily be the case that no one has such a bizarre item on sell, and in this solution we would wait for one seller to appear forever.

An alternative solution that avoids the deadlock situation is presented in 3.2. In this case, after receiving the message {`:buy:``Wisdom Teeth``:`}, we only wait for a maximum of 10 seconds for a seller before giving up, at which point we execute the body of the `timeout` clause. The statement `thisComp() <<< {:buy:X:}` resends the message {`:buy:``Wisdom Teeth``:`} to the current component because it had been removed from the mailbox when the outer pattern was satisfied, and we might need to use it later.

A drawback of this second solution is that it does not preserve the order of the messages in the mailbox. The message {`:buy:``Wisdom Teeth``:`} was in the forefront before being removed, but when resent, it will be enqueued to the end. As a result, trying to produce a solution that preserves the order is a non-trivial exercise.

In view of these difficulties, we have decided to add a new construct to the language that facilitates synchronisation. JCThorn provides the *join* operator `and` that allows us to easily solve this problem, as illustrated in listing 3.3. The rule in the `receive` construct will only fire when the two required messages, sell and buy, are present in the mailbox. In other words, instead of matching one message at a time as in Thorn, a join expresses that we need to match a combination of messages in order for a rule to be applicable. Remarkably, joins have reduced the number of required lines to solve the problem from 20 to 3.

```
1  while(<NOT_FOUND>) {
2    receive{
3
4      {:sell:X:} => {
5        receive{
6
7          {:buy:$(X):} => { /* pair found */ }
8
9          timeout(10000) => {
10           thisComp() <<< {:sell:X:};
11     } } }
12
13     | {:buy:X:} => {
14       receive{
15
16         {:sell:$(X):} => { /* pair found */ }
17
18         timeout(10000) => {
19           thisComp() <<< {:buy:X:};
20 } } } } }
```

Listing 3.2: Second attempt at matching sellers and buyers

```
1  receive{
2    {:sell:X:} and {:buy:$(X):} => {  /* pair found */ }
3  }
```

Listing 3.3: Solution to the sellers and buyers problem using joins

Sellers and buyers in the high-level communication mode

As hinted in the beginning, Thorn also provides a high-level communication mode. We will first show a component that uses the high-level mode and then try to justify why it is much more difficult to solve the buyers and sellers problem in this mode using the original Thorn.

Listing 3.4 shows a one-cell buffer that stores the lower case version of a given string and returns it when prompted. It declares a variable `value`, a function `toLower` and two channels, `get` and `put`. The built-in function `body` is where the execution of the component starts. Furthermore, the `serve` statement is used to wait for a high-level communication event. Any component can store a string on the buffer by sending it on the asynchronous channel `put`. This message is placed in the buffer's mailbox and retrieved when the buffer executes the `serve` statement. Similarly, any component can retrieve the string by sending a message on the synchronous channel `get`.

The sellers and buyers problem is very difficult to solve in the high-level mode because, as we have seen in listing 3.1, we need two nested `receive` statements with slightly different cases to be able to synchronise on the value of the goods. However, in the high-

level mode, we can only declare channels in the top level of the component, and the `serve` statement is, to some extent, equivalent to a `receive` construct in which its cases are the declared channels. As a result, no matter where a `serve` statement appears, its cases are always the same set.

```
1   component oneCellBuffer{
2     var value := '';
3
4     fun toLower(s) { /* return lower case version of s */ }
5
6     sync get() {
7       return value;
8     }
9
10    async put(s) {
11      value := toLower(s);
12      return;
13    }
14
15    body{
16      while(true){
17        serve;
18  } } }
```

Listing 3.4: One-cell buffer storing lower case strings

On the other hand, JCThorn provides a construct that we called *chords* that allows us to easily solve the buyers and sellers problems in the high-level mode. The solution is shown in listing 3.5 and, as we can see, it is very similar to the solution using joins. It declares a chord with two channels, `buy` and `sell`, whose body will only be executed when they are pending messages with the same contents in both channels.

```
1   component server{
2
3     async buy(X) and async sell($(X)) {
4       /* pair found */
5     }
6
7     body{
8       serve;
9     }
10  }
```

Listing 3.5: Solution to the buyers and sellers problem using chords

Joins or chords?

In THORN, the high-level communication events are treated as syntactic sugar and translated at compile time into low-level events. Obviously, JCTHORN follows the same approach and translates chords into joins. Hence, chords and joins are not that different.

In the JOIN CALCULUS literature, these two terms are often used interchangeably. In THORN we distinguish them because it is the only JOIN CALCULUS based language that provides both. In the next two sections we will present each of them in more detail. However, in the rest of this report, we may use the terms interchangeably given that most of their features are common. Only in the cases in which their behaviours are different shall we explicitly distinguish between the both of them.

3.2 Joins

As previously mentioned, the low-level communication mode of JCTHORN allows synchronisation on the receipt of multiple messages thanks to the use of *joins* in the selective `receive` construct. Joins in JCTHORN resemble joins in JERLANG, as we can observe from listing 3.6, which shows a one-cell buffer that synchronises on the reception of messages ``` ``get'' ``` and `{:put:x:}`.

```
1  receive{
2    ``get'' from c and {: put:x :} => { c <<< x;}
3  }
```

Listing 3.6: *Joins* for the low-level communication mode in JCTHORN

Any component can store a value `x` in the buffer by sending the message `{:put:x:}` to it. To retrieve the value, a component can send the message ``` ``get'' ```. When both messages are present in the mailbox of the buffer, this can then retrieve the sender of the ``` ``get'' ``` message and return the stored value. To retrieve the sender, the `from` operator is used, and to return the value, the low-level send operator `<<<` appears in the body of the join.

Low-level communication in JCTHORN is asynchronous. Therefore, the component sending the ``` ``get'' ``` message needs to execute a `receive` statement if he wants to retrieve the value replied by the buffer. Moreover, the buffer can reply to as many components as needed in the body of the join.

As we can see, each join has two parts, the *join declaration* and the *join body*. The join body is just the block that follows the arrow `=>`. The join declaration is a set of *join patterns* concatenated with the `and` operator. Each join pattern has the format described in figure 3.1. `C`, `S` and `E` are all algebraic patterns. `C` matches the contents of the message, `S` the sender and `E` the envelope. The envelope clause captures the message together with its metadata, and is normally used in the high-level communication mode to delegate work to helper servers (see next section, 3.3, for more details). Both the from and envelope parts are optional. In the remainder of the report we will refer to both algebraic patterns

and join patterns just as patterns when the meaning is clear from the context.

C [`from` S] [`envelope` E]

Figure 3.1: Format of a *join pattern*

Listing 3.7 shows a receive construct with 2 joins separated by the or (ie. '|') operator. Both joins match two singleton lists. This example helps to illustrate the difference between *free* and *bound* variables in algebraic patterns in JCThorn. JCThorn follows Thorn's approach of making this difference explicit in the syntax.

In the first join, `y` appears both free and bound. It is free in the first occurrence, so `[y]` will match any singleton list. On the other hand, `$(y)` matches an element whose value is equal to the value hold in variable `y`, so the second join pattern will match a singleton list holding the same element as the first matched list. We can also see that join patterns can match variables defined outside joins, as it occurs with the variable `x` in the second join. Clearly, the pattern `[$(x)]` will only match the list `[2]` in this example.

Thorn's approach avoids the common mistake of unintentionally use a bound variable as a wildcard pattern in imperative languages. Hence, it would be a syntactic error to use the pattern `[x]` instead of `[$(x)]` in the second join, given that `x` has already been defined in the first line. In that case we would get a `Duplicate definition` exception.

```
1  x = 2;
2  receive{
3    [y] and [$(y)] => { /* case 1 */ }
4    | [z] and [$(x)] => { /* case 2 */ }
5  }
```

Listing 3.7: Receive with two joins

Example 3.7 also demonstrates a situation where a message can satisfy multiple patterns in the same join. Any singleton list will match both join patterns in the first join, but JCThorn will ensure that this only fires when there are two singleton lists with the same element in the mailbox. In other words, any message cannot satisfy more than a single join pattern in a successful match.

Most languages that support joins, with the exception of Jerlang, forbid this situation from arising by requiring the channels in a join to be unique. Although their approach greatly simplifies the implementation of the language (see section 4.3), we believe that allowing them is crucial to creating a usable language. There are many problems that are easily solved by using the same pattern multiple times in a join, as for example the *Santa Claus* problem (section 6.1).

3.3 Chords

Chords in JCThorn resemble Join Calculus inspired constructs in languages such as JoCaml, Polyphonic C# and Join Java. Listing 3.8 shows a one-cell buffer implemented using chords in JCThorn. It presents the same behaviour as that of the buffer in 3.6, but it gives all the component code for completeness. As explained in section 2.4, `spawn` starts a new component, and the built-in function `body` is where execution starts.

```
1  spawn{
2    sync get() and async put(x) {return x;}
3    body {while (true) serve;}
4  }
```

Listing 3.8: *Chords* for the high-level communication model

This component defines one chord that synchronises on the receipt of messages on channels `get` and `put`. Since `get` is a synchronous channel, a component sending a message on it will block until the buffer fires the chord and returns the value stored in variable `x`. On the other hand, a component sending a message on the asynchronous channel `put` will not block. Moreover, the `serve` statement causes the buffer to wait until it has received the two required messages to satisfy the chord.

Similar to joins, a chord has two parts, the *chord declaration* and the *chord body*. The body is the block following the declaration, which is a set of chord patterns concatenated with the `and` operator. Each *chord pattern* introduces a channel, whose name has to be preceded by a modifier that specifies whether the channel is synchronous or asynchronous (`sync` and `async`). The channel name can also be followed by the `from` and `envelope` clauses. As already mentioned, the `from` clause is used to retrieve the sender of a message. On the other hand, the `envelope` clause returns the message together with its metadata, allowing a server thread to pass responsibility for answering a request to a worker thread. How to delegate a request is illustrated in listing 3.9, and how to receive the corresponding delegated request in 3.10 [1].

A server can use the built-in function `splitSync()` as an instruction not to send a response to a synchronous command. On the other hand, a worker needs the original message and its metadata to be able to respond to the request. It uses the built-in function `syncReply()` to this end. Importantly, the whole process is transparent to the client, who only needs to communicate with the server.

```
1  sync command() envelope e{
2    worker <-- subcommand(e);
3    throw splitSync();
4  }
```

Listing 3.9: Delegation of a request to a worker thread

[1] Both examples were taken from [10]

```
1  async subcommand(e){
2    workerResponse = ... /* compute it */
3    syncReply(e, workerResponse);
4  }
```

Listing 3.10: Receipt of delegated request by worker

Returning to chords, each can have at most one synchronous channel declaration, and it is a syntactic error to define more. This is also the approach taken in the POLYPHONIC C# and JOIN JAVA languages. Regarding the position of the synchronous channel declaration, JCTHORN does not present any restriction and follows the example of POLYPHONIC C#. In JOIN JAVA the synchronous channel can only appear in first position, restriction that we find unnecessary.

The reason why we only allow one synchronous channel declaration per chord is to maintain compatibility with THORN and avoid extending the `return` command. However, as listing 3.11 shows, it is not difficult to write a program that behaves as if both channels in the chord were synchronous. This program behaves like a barrier that defines a synchronisation point in the execution of two parallel tasks. The only subtlety is that the component using the asynchronous channel `wait2` will have to perform a selective receive in order to retrieve the server's response, ie. the string ``go''.

```
1  spawn{
2    sync wait1() and async wait2() from S {S <<< "go"; return;}
3    body {while (true) serve;}
4  }
```

Listing 3.11: Multiple synchronous channels in one chord

As a result of this hack, JCTHORN is as powerful as JOCAML, that allows multiple synchronous channels per chord. However, an advantageous difference with JOCAML, POLYPHONIC C# and JOIN JAVA is that JCTHORN allows the same channel to appear more than once in the same chord, which increases the expressiveness of the language.

3.3.1 Algebraic patterns on arguments

Channels in JCTHORN can perform algebraic pattern matching on arguments during chord resolution. Programs using this feature resemble functional programs written in languages like HASKELL. For instance, the code in listing 3.12 outputs ``empty'' if the given list on channel `isEmpty` is indeed empty, and ``not empty'' otherwise.

```
1  spawn{
2    async isEmpty([]) and async doPrint() {println("empty");}
3    async isEmpty(xs) and async doPrint() {println("not empty'');}
4    body {serve;}
5  }
```

Listing 3.12: Algebraic pattern matching on arguments to channels

Similarly, JoCaml also allows pattern matching on arguments to channels. However, an equivalent program written in JoCaml would be ambiguous given that the resolution of chords in this language is non-deterministic. When the list is empty, and considering that the pattern `xs` also matches the empty list, either message could print in JoCaml. JCThorn solves this problem by evaluating the chords in order of appearance.

Clearly, pattern matching on arguments is quite a powerful feature that programmers can use to write more elegant programs. Using the pattern matching operator `:` that performs type checking, we can write the program given in 3.13. This is a one-cell buffer that accepts either an integer or a string.

```
1    sync get() and async put(n:int) {return n;}
2    sync get() and async put(s:string) {return s;}
```

Listing 3.13: Type checking

3.3.2 Inheritance

Inheritance and chords integrate well in JCThorn because these are two orthogonal features. Contrarily, the inclusion of chords in Polyphonic C# and Join Java forced the designers to restrict the cases where inheritance is supported. The problem lies in the fact that inheritance is available between classes in these languages, but classes can also declare chords.

In Join Java, inheriting from classes that declare chords is completely forbidden. In Polyphonic C# this is allowed, but with some restrictions. For instance, the declaration of the `D` class in listing 3.14 is illegal. Otherwise, all the calls to `f()` on a `D` object would deadlock forever. This behaviour would be particularly problematic if an instance of class `D` were passed to a fragment of code expecting an object of type `C`, as the chances are that it would deadlock. More details are given in the background in section 2.3.3.

```
1  class C {
2    virtual void f() & virtual async g() { /* body1 */ }
3  }
4  class D : C {
5    override async g() { /* body2 */ }
6  }
```

Listing 3.14: Invalid class inheritance in Polyphonic C#

On the contrary, in JCThorn a component and a class are two different entities. Classes can only define methods, introduced by the `def` keyword, and not chords. Listing 3.15 shows a typical class declaration in JCThorn. Moreover, inheritance is only available between classes, which are clearly not affected by the introduction of chords. In other words, the introduction of chords in JCThorn has no effect on inheritance because components are the only entities that can declare them (as shown in listing 3.16), but inheritance is not available for them.

```
1  class Point{
2    var x; var y;
3
4    new Point(x', y'){ x := x'; y := y';}
5
6    def move(x', y'){ x := x'; y := y';}
7  }
```

Listing 3.15: Class declaration in JCThorn

```
1  component Buffer{
2    sync get() and async put(x) {return x;}
3    body {while (true) serve;}
4  }
```

Listing 3.16: Component declaration in JCThorn

3.4 Common features to both joins and chords

In this section we present the common features to both joins and chords. The reader is advised that we might use both terms interchangeably from now on.

3.4.1 Resolution and priorities

In order to fully understand how the `receive` and `serve` commands operate, it is important to also understand how the mailbox is handled. Each component has a mailbox where it receives both high-level and low-level messages. Internally, high-level messages are translated into low-level messages, which helps to explain why the mailbox is shared.

Messages from the mailbox are retrieved one at the time, until the current message along with any combination of previously retrieved messages satisfies a join, when executing a `receive`, or chord, when executing a `serve`. Moreover, the order in which joins and chords are declared is important, given that both the `receive` and `serve` constructs are specified to check the cases in order.

To illustrate how resolution operates, consider the case of a one-cell buffer that accepts strings and integers, listed in 3.17. Assume that its mailbox contains, in this order, the messages `put("hello")`, `put(5)` and `get()`. Subsequently, when the third message in the mailbox is retrieved (ie. `get()`), it is first checked against the first join. This is triggered because the pattern `put(n:int)` has already been satisfied by the second message in the mailbox. After reduction, the mailbox will only contain the message `put("hello")`.

```
1  spawn{
2    sync get() and async put(n:int) {return n;}
3    sync get() and async put(s:string) {return s;}
4    body {while (true) serve;}
5  }
```

Listing 3.17: In order resolution of chords in JCThorn

The *in order* semantics is important because it allows programmers to write simpler programs. With *in order* semantics, the three patterns in listing 3.18 will satisfy disjoint sets of messages, and we will have the certainty that `xs` will never match the empty or singleton lists. Assuming the non-deterministic semantics of other languages, an equivalent program would be more tedious to write, as shown in listing 3.19.

```
1  receive{
2    [] => { /* case 1 */ }
3    | [x] => { /* case 2 */ }
4    | xs => { /* case 3 */ }
5  }
```

Listing 3.18: In order resolution of joins in JCThorn

```
1  receive{
2    [] => { /* case 1 */ }
3    | [x] && ![] => { /* case 2 */ }
4    | xs && ![x] && ![] => { /* case 3 */ }
5  }
```

Listing 3.19: Unordered resolution of joins in other languages

Joins and chords can also be assigned numeric priorities with the `prio` operator, as demonstrated by listings 3.20 and 3.21. More generally, the default priority for the joins that have not been explicitly assigned one is 0. All the messages currently in the mailbox will have to be checked against a higher priority join before attempting to match a lower priority one.

```
1  receive{
2    ``get'' from c and {: put:x :} prio 10 => { /* body */ }
3  }
```

Listing 3.20: Explicit numeric priority to join

```
1  sync get() and async put(x) prio 10 { /* body */ }
```

Listing 3.21: Explicit numeric priority to chord

For instance, the program in listing 3.22 is very similar to that in 3.17, but with a slightly different behaviour. Assume that the mailbox of the buffer contains, in this order,

the messages `put("hello")`, `get()` and `put(5)`. In 3.17 the second join will fire after retrieving the second message in the mailbox. On the contrary, in 3.22 the first join will be triggered as it has higher numeric priority than the second join and can be satisfied by the second and third messages. In other words, all the messages currently in the mailbox are checked against the first join before attempting to reduce the second join when the explicit priority is used.

```
1  spawn{
2    sync get() and async put(n:int) prio 10 {return n;}
3    sync get() and async put(s:string) {return s;}
4    body {while (true) serve;}
5  }
```

Listing 3.22: Numeric priorities

Numeric priorities are important because, in many situations, we are only interested in the presence of a message in the mailbox rather than its position. For example, the presence of an `abort_critical` message might require immediate action, while another message appearing before it in the mailbox might not be as important.

3.4.2 Non-linear patterns and side conditions

Although non-linear patterns have already been used in this report, we will introduce them here in more detail. A pattern is non-linear when it contains the same variable more than once. In 3.23, the variable `x` appears twice. Hence, the resolution of the join also synchronises on the value that `x` will hold. As we already know, this program matches sellers and buyers of the same goods.

```
1    receive{
2      {: sell:x :} and {: buy:$(x) :} => { ... }
3    }
```

Listing 3.23: Non-linear patterns in JCThorn

It can be observed that JCThorn's non-linear patterns obey the Thornian syntactic distinction between bound and free variables. Consequently, the syntax is slightly different to that of non-linear patterns in languages like Prolog or Jerlang. Remarkably, JCThorn and Jerlang are the only two Join-Calculus-based languages implementing them. We will see later that the reason is that supporting them makes their implementation more complex.

Apart from synchronising on the value that a variable holds, the powerful pattern matching capabilities inherited from Thorn also allow the expression of more elaborate constraints. Algebraic pattern matching in Thorn eliminates the need for side conditions and guards and, as a result, these are not included in the language.

The JErlang program in 3.24 demonstrates a typical example where guards are needed. In that program, the goods will only be sold to the customer when he has paid

at least its full price. 3.25 shows an equivalent program written in JCThorn.

```
1    receive{
2      {price, X} and {pay, Y} when (X <= Y) => { /* sell good */ }
3    }
```

Listing 3.24: JErlang guards

```
1    receive{
2      {:price:X:} and {:pay:(Y && (Y <= X)?):} => { /*sell good*/ }
3    }
```

Listing 3.25: JCThorn pattern matching constraints

The value in the `pay` aggregate has to satisfy both patterns in the conjunctive pattern. Recall that `p && q` indicates a value that matches both `p` and `q`. So in 3.25 the left hand side of the conjunctive pattern does not present any restriction and merely binds the value to the variable `Y`. On the other hand, the right side restricts the value that `Y` can hold to be less than the value held in `X`.

Conjunctive patterns allow finer control of when pattern matching is stopped than side conditions do. In other words, failed matches can be resolved sooner. For a meaningful example consider the pattern `[x:int && (x != 0)?, $(32 div x)]` borrowed from [10]. This pattern is true on `[4,8]` and false (rather than dividing by zero) on `[0,0]`.

3.4.3 Timeouts

JCThorn inherits the `timeout` construct from Thorn. A `timeout` is used to limit the time a component waits for the arrival of a message in `receive` (3.26) and `serve` commands (3.27). The amount of time to wait is specified in milliseconds.

```
1    receive{
2      "message" => { ... }
3      timeout(1000) => { ... }
4    }
```

Listing 3.26: Timeout used by receive

```
1    serve timeout(1000){ /* timeout action */ };
```

Listing 3.27: Timeout used by serve

3.5 Incompatibilities with Thorn

During the design process of JCThorn, we assumed that the extensions were to be included in the original Thorn language. As such, JCThorn follows the same design principles of Thorn, and the majority of Thorn programs are also valid JCThorn programs. Unfortunately, there are a couple of cases of minor importance, described below, where compatibility is lost.

We decided to break compatibility because we believe that JCThorn proposes the most elegant path to incorporate chords into Thorn. When making this decision, we took into account that Thorn is still in the first stages of development without an established user community. This fact gives Thorn designers the freedom to make backwards-incompatible changes to the language without any major negative effects.

Had Thorn served an established user-based community, we would have certainly designed JCThorn differently. In that case, compatibility would have been more important than the elegance of the design.

3.5.1 Before and After

There is a syntactic difference between Thorn and JCThorn in `before` and `after` clauses, which arises from the semantic differences between the two languages. The `before` and `after` clauses next to the `serve` statement define share actions to be performed, respectively, before and after the execution of any chord.

The Thorn program in 3.28 illustrates how `before` and `after` are used. No matter which means of transport is chosen, the actions to be performed by the client before and after travelling are, respectively, leave the house and arrive at college.

```
1  spawn server{
2    sync takeUnderground() { ... }
3    sync takeBus() { ... }
4
5    body{
6      serve before(msg, client) {
7        leaveHouse(client);
8      } after(msg, client) {
9        arriveCollege(client);
10 } } }
```

Listing 3.28: Before and after clauses in Thorn

The built-in functions `before` and `after` can take from 0 to 2 arguments. If used, the first argument is always the message envelope and the second the message sender. This is unambiguous in Thorn because a chord declaration only defines one channel. On the other hand, a chord in JCThorn can define multiple channels. Consequently, it is not clear which sender should get bind to the argument variable.

JCThorn solves this problem by representing the arguments as lists. The indexes of the elements in the lists corresponds to the position of the chord pattern matching the message we are referring to. To see how this works, consider the equivalent JCThorn program of listing 3.29. In this case, given that all the chords are unary (they have a single channel declaration), the lists `msgs` and `clients` will hold a single element each.

This approach also solves the problem of components with variable length chords, as demonstrated in 3.30. In this program, clients will only go to college by car if there are at least three of them. When that is the case, the lists `msgs` and `clients` will hold three elements each, one per client.

```
1  spawn server{
2    sync takeUnderground() { ... }
3    sync takeBus() { ... }
4
5    body{
6      serve before(msgs, clients) {
7        leaveHouse(clients(0));
8      } after(msgs, clients) {
9        arriveCollege(clients(0));
10 } } }
```

Listing 3.29: Before and after clauses in JCThorn

```
1  spawn server{
2    sync takeUnderground() { ... }
3    sync takeBus() { ... }
4    async byCar() and async byCar() and async byCar() { ... }
5
6    body{
7      serve before(msgs, clients) {
8        for(i <- clients){
9          leaveHouse(clients(i));
10       }
11     } after(msgs, clients) {
12       for(i <- clients){
13         arriveCollege(clients(i));
14 } } } }
```

Listing 3.30: Before and after clauses in JCThorn

3.5.2 Catch

The second situation where THORN and JCTHORN differ relates to the `catch` clause that can also appear next to the `serve` statement. In this case, the reason for the incompatibility is twofold. On the one hand, implementation constraints in JCTHORN forbid its use (see section 4.4.2). On the other hand, we believe its removal creates a more consistent language.

The intended use of `catch` in THORN is to define an action to be performed when a message has been sent on an existing channel with the wrong number or type of arguments. The component in 3.31 defines a target component that creates a `plink` channel taking an integer argument. Clearly, if `T` receives the messages `plink("hello")` or `plink(2,3)`, it will output `"target missed"` because they contain the wrong arguments. In that case, both messages will be removed from `T`'s mailbox.

```
1  T = spawn target {
2    async plink(x:int) { println("target hit"); }
3
4    body{
5      serve catch { x => { println("target missed");}};
6  } }
```

Listing 3.31: Catch clause in Thorn

Interestingly enough, if `T` receives the message `boom(5)`, nothing will print because it does not define such a channel. `T` will not execute the `serve` or `catch` statements and it will not remove the message from the mailbox. However, we believe that it would be preferable that the behaviour in both presented cases were the same, either to catch or ignore both errors.

More importantly, consider the chess server program in listing 3.32. Each player is assigned a boolean value which it uses to identify herself to the server. When the server receives the move of the player whose turn is to move, it executes it and then changes the turn value. The `catch` clause is there to catch those situations where a move with an invalid format has been submitted.

Unfortunately, the `catch` clause matches more cases than desired. The *black* player (identified by `true`) waiting for its turn should be allowed to send her move to the server before the *white* player does. This situation could arise when the *black* player has already decided what her next move will be independently of what her opponent, who is to move next, decides to do. In that scenario, the correct behaviour of the server should be to wait for *white*'s move, execute it and then execute *black*'s. However, this is not what actually happens in THORN.

THORN treats *black*'s move as invalid and executes the catch statement. The server thinks that the first argument of *black*'s move is not correct because it does not match the value currently hold in the `turn` variable, given that it is *white*'s turn. The cause of this behaviour lies in the way THORN designers have implemented the `catch` clause, which relies on the use of algebraic pattern matching (more details in section 4.4.2).

Both to avoid the catch problem when using patterns that match the values hold in component state variables and to make the behaviour of the language more consistent, JCThorn has eliminated the `catch` clause next to `serve` statements from the language.

```
1   spawn chessServer {
2     var turn := false;
3
4     fun executeMove(player, move) { ... }
5
6     sync move($(turn), move) {
7       executeMove(turn, move);
8       turn := !turn;
9       return;
10    }
11
12    body{
13      serve catch { x => { println("invalid move format");}};
14  } }
```

Listing 3.32: Chess server

3.6 Summary

In this chapter we have introduced the extensions which JCThorn provides upon Thorn. The most visible features are *joins*, used in the low-level communication mode, and *chords*, used in the high-level mode. Both constructs serve the same purpose, which is to perform some actions when a combination of messages satisfying certain patterns has been received. In contrast, Thorn only allows matching a single message from the mailbox at a time.

Joins and chords overcome certain limitations of the Thorn language, and allow programmers to express concepts like finding the first pair of received messages that agree on their value without removing any other message from the mailbox. This idea, although very simple, cannot be expressed in Thorn.

As a result, solutions to certain synchronisation problems can be expressed much more easily using either joins or chords. Both constructs are, to some extent, equivalent, considering that chords are translated at compile time into joins. Consequently, the runtime resolution of both constructs is the same, which can be summarised as follows:

- All the messages in the mailbox will be checked against joins with a higher priority (explicitly assigned with the `prio` operator) before attempting to reduce those with a lower priority.
- Joins at the same level of priority are tested *in-order*, according to the order they are declared in the program.
- *First-match* semantics with regards to the messages in the mailbox.

Other features that both constructs provide are non-linear patterns, powerful pattern matching capabilities and timeouts. Additionally, chords support algebraic pattern matching on the formals to the channels, and integrate well with inheritance. Other languages had to introduce restrictions on inheritance, but JCThorn did not suffer from this problem since classes and components are two different entities.

Finally, we have also described the incompatibilities between Thorn and JCThorn. In JCThorn, the arguments to the `before` and `after` clauses are lists instead of single elements. Moreover, JCThorn has removed the `catch` clause next to the `serve` statement from the language to avoid the unexpected behaviour of programs using this feature along with interpolation patterns.

In the next chapter we will present how joins and chords have been implemented. Joins required modifications to the execution engine of the interpreter, and chords were translated at compile time into joins.

Chapter 4

The implementation

Chapter 3 has provided a description of JCThorn at the user level. Now we will focus on the implementation details that might be of interest to programming language designers. We will describe the changes made to the grammar and abstract syntax tree (AST) and then we will cover the implementation of joins and chords. The inclusion of joins required the extension of the interpreter's execution engine and the matching capabilities of the language. On the other hand, the high-level communication constructs in Thorn are translated at compile time into low-level communication constructs. Thus, the implementation of chords consisted of substantial modifications to this translation.

The reader should be aware that in this chapter we will only cover the elementary algorithmic framework. All of the optimisations are described in the next chapter, together with the state explosion problem whose effects they try to minimise.

4.1 Overview

There are two general approaches to programming language implementation, namely interpretation and compilation. At the moment, Thorn only provides an interpreter, written in Java. Although there exists an old compiler, this is outdated and no longer supported. Hence, we decided to implement JCThorn on the interpreter until a new version of the compiler, currently in development, is released.

The changes made to the interpreter are illustrated in figure 4.1. It shows the process of interpreting a JCThorn program (although quite abstractly) and the packages in the interpreter that received the most changes. A number prefixed by the `+` symbol means the number of lines of code that have been added to the corresponding package. Similarly, `!` stands for the number of lines that have been modified (see appendix E for a more comprehensive log of changes). Finally, the color codes together with table 4.1 show where the changes described in each of the parts of this report were located.

The creation of JCThorn's parser, the `fisher.parser` package, is a one-off operation. This is done using a tool called JavaCC that takes as input the file `grammer-fisher.jj`, which specifies the grammar of the JCThorn language. Parsing a JCThorn

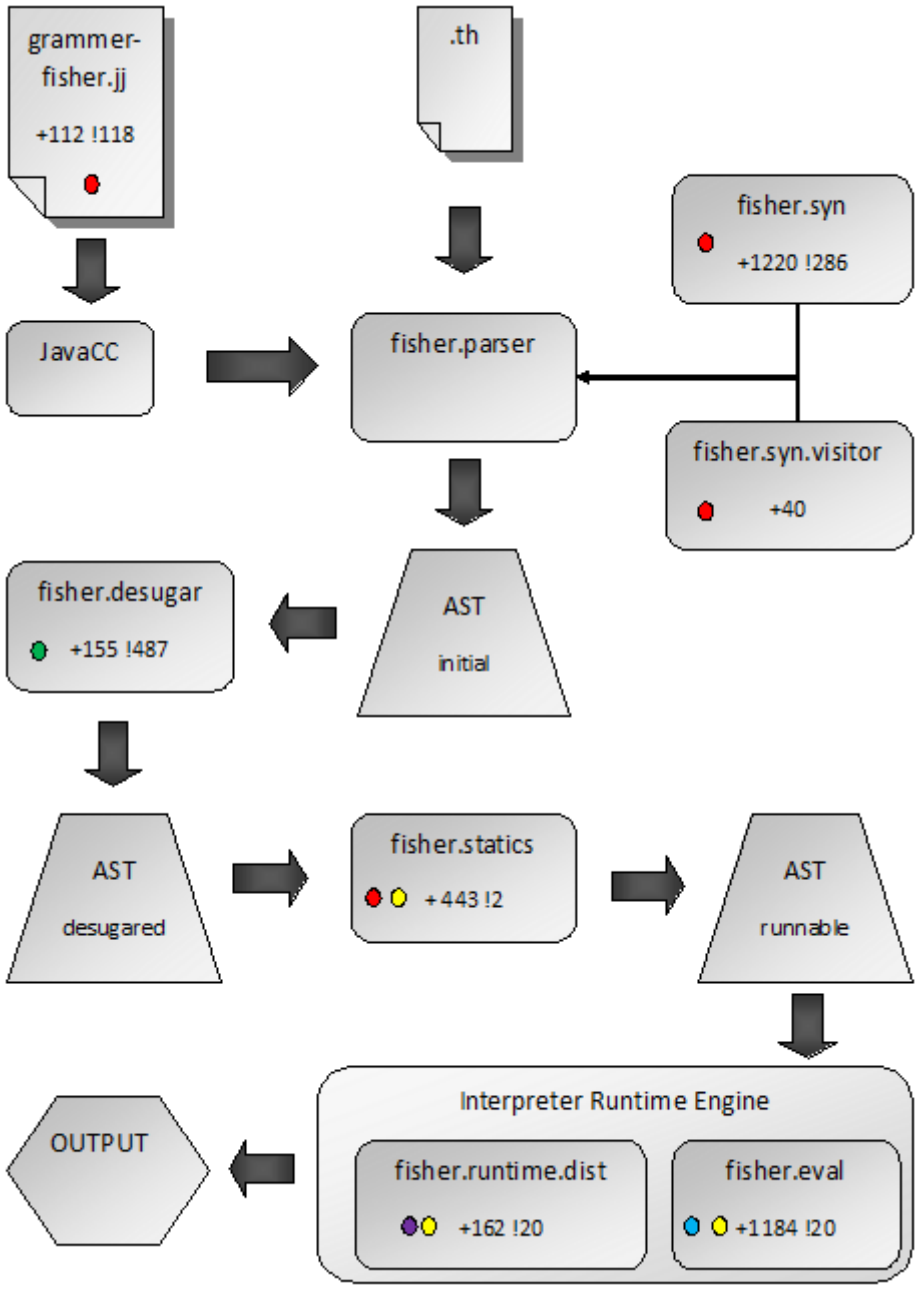


Figure 4.1: Interpretation process and localisation of changes

COLOR	LOCATION	TOPICS
red	section 4.2	Grammar, AST, de Bruijn indexes
purple	section 4.3.2	Interpreter runtime entities, scanning of mailbox
blue	sections 4.3.3 and 4.3.4	Evaluation of joins
green	section 4.4	Translation of chords into joins
yellow	chapter 5	Optimisations

Table 4.1: Topics of each of the sections of the report

program (`.th` extension) creates an initial abstract syntax tree (AST), whose nodes are instances of the classes defined in the `fisher.syn` package. Moreover, some of the visitor classes defined in `fisher.syn.visitor` are used to perform some modifications to this initial AST.

The classes in `fisher.desugar` eliminate syntactic sugar from the program. For example, chords are translated into joins during this step. The desugared AST becomes the input to some static analyses, defined in `fisher.statics`, that produce the runnable AST. Finally, the interpreter's execution engine takes this AST to produce the output of the JCThorn program.

4.2 Grammar and AST

To better understand the changes made to the grammar, we will first give the fragments of Thorn's grammar concerning the `receive` construct and `component` declarations. For simplicity and readability, we will show a somewhat abstracted version given in Extended Backus-Naur Form (EBNF). The actual grammar, although more complex, is also expressed in EBNF. This is the input given to JavaCC (Java Compiler Compiler) to generate the parser for JCThorn.

JavaCC takes a formal grammar provided in EBNF notation and outputs a parser written in Java. JavaCC generates top-down parsers, which limits its use to LL(K) grammars that do not allow left-recursion [12].

The grammar for Thorn's receive construct is given in 4.2. Non-terminals are written in slanted text and terminals in typewriter font. Square brackets are used to denote optional syntax, while the '*' symbol expresses that an item can appear 0 or more times. We can see that each *case* in the *receive* construct defines a maximum of three patterns to match a single message. On the other hand, figure 4.3 shows the grammar for a component declaration in Thorn. We observe that *async_decl, sync_decl* and *fun_decl* rewrite to a keyword plus the rule for *funbody*. This rule declares only a single channel, whose name might be followed by the `from`, `envelope` and `prio` closures.

receive	→	`receive` *cases* [*timeout*]
cases	→	*case* (*case*) *
case	→	*pattern* [`from` *pattern*] [`envelope` *pattern*] [`prio` *int*] *block*
block	→	*statement* (*statement*)*
timeout	→	`timeout` *int block*

Figure 4.2: THORN syntax of receive construct

component_decl	→	`component` *name formals process_member* (*process_member*)*
process_member	→	*sync_decl* | *async_decl* | *fun_decl*
	|	*body* | *import_stmt* | *class_decl* | *var_decl*
sync_decl	→	`sync` *funbody*
async_decl	→	`async` *funbody*
fun_decl	→	`fun` *funbody*
funbody	→	*name formals* [`from` *pattern*] [`envelope` *pattern*] [`prio` *int*] *block*

Figure 4.3: THORN syntax of component declaration

The *receive* construct in JCTHORN is modified to include a set of *join_declarations* instead of *cases*, as shown in 4.4. Each *join_decl* consists of a *join_signature* and a body (ie. *block*). Optionally, it can also be assigned a priority. We observe that the signature is a set of *join_patterns* concatenated with the `and` keyword, each of which matches a single message in the mailbox.

receive	→	`receive` *join_declarations* [*timeout*]
join_declarations	→	*join_decl* (*join_decl*) *
join_decl	→	*join_signature* [`prio` *int*] *block*
join_signature	→	*join_pattern* (`and` *join_pattern*)*
join_pattern	→	*pattern* [`from` *pattern*] [`envelope` *pattern*]

Figure 4.4: JCTHORN syntax of receive construct

component_decl	→	`component` *name formals* (*process_member*)*
process_member	→	*chord_decl* | *fun_decl* | *var_decl*
	|	*body* | *import_stmt* | *class_decl*
chord_decl	→	`sync` *channel_decl* (`and async` *channel_decl*)* *prio_block*
	|	`async` *channel_decl* (`and` *chord_decl_aux* | *prio_block*)
chord_decl_aux	→	`async` *channe_channel* (`and` *chord_decl_aux* | *prio_block*)
	|	`sync` *channel_decl* (`and async` *channel_decl*)* *prio_block*
channel_decl	→	*name formals* [`from` *pattern*] [`envelope` *pattern*]
prio_body	→	[`prio` *int*] *block*

Figure 4.5: JCTHORN syntax of component declaration

In the case of `component` declarations, JCThorn replaces the Thornian *sync_decl* and *async_decl* items for a *chord_decl* non-terminal in the set of possible *process_member*s, as demonstrated in figure 4.5. Each *chord_decl* consists of a set of *channel_decls* concatenated with the *and* keyword. This set is either followed by a priority and the body (*block* non-terminal), or just the body.

The set of *channel_decls* has to be nonempty and can contain at the most one synchronous channel, which can appear in any position. These two constraints are enforced by the grammar. Moreover, the *chord_decl* rule has been split in two to ensure that the grammar is in LL(1). In other words, the parser only needs a look-ahead of 1 in order to parse any component declaration correctly.

Regarding the AST, this was extended to include nodes for the following elements:

- Join declarations
- Join signatures
- Join patterns
- Chord declarations
- Chord bodies, which include the priority if defined
- Channel declarations

As a result, all the visitor classes had to be modified accordingly. Thorn's interpreter uses the visitor pattern during code compilation to perform some of the required tasks. Furthermore, we also had to alter the way seals are created for high-level communication constructs. Seals, the de Bruijn indexes, were introduced in section 2.4.10.

When compiling the program in listing 4.1, two `Seal` objects are produced. The first is associated with the first three occurrences of the identifier `x`, and the second with the last four. In Thorn, we cannot have a channel argument referring to a variable defined by another channel. However, with JCThorn, this achievable. The seal of all the occurrences of variable `x` in program 4.2 is the same.

```
1  spawn{
2    sync square(x){return x*x;}
3    sync cube(x){return x*x*x;}
4    body {serve;}
5  }
```

Listing 4.1: Example that demonstrates how Seal objects are created in Thorn

```
1  spawn{
2    sync buy(x) and async sell($(x)){return x;}
3    body {serve;}
4  }
```

Listing 4.2: Example that demonstrates how Seal objects are created in JCThorn

4.3 Joins implementation

In this section we explain how joins are implemented in JCThorn, but first review the approaches taken in other languages for join resolution. We then give an overview of the interpreter's runtime environment and the algorithm responsible of scanning the mailbox of each component. We conclude by examining in more detail the two phases in the evaluation of joins, namely the local and contextual matching phases.

4.3.1 Review of other languages

To date, there exist two main approaches to implement *Join Calculus* based constructs. The first one is based on deterministic automata and has been followed by languages like JoCaml[24] and Polyphonic C#, but it is not appropriate for JCThorn. The second one has been proposed by JErlang and is based on the Rete algorithm, traditionally used in *Production Rule Systems*.

The finite-state automata solution is used by languages that implement chords. The matching state of chords is stored in a vector that has one element for each defined channel. The value of each element can be 0, when there are no messages pending on the corresponding channel, or N otherwise.

As an example, consider the JoCaml program in listing 4.3. It defines three channels `A`, `B` and `C`. As a result, the vector holding the matching state would have a length of 3. After some arbitrary order has been decided, and assuming that there are pending messages on `B` and `C` and not in `A`, the state could be 0NN.

```
1  let A(n) | B() = P(n)
2  let A(n) | C() = Q(n)
3  ;;
```

Listing 4.3: JoCaml chords

A *matching status* is defined as a status that holds enough N, so that at least one chord can be fired. The state evolves towards matching statuses as messages arrive, and in the opposite direction when matching occurs. These transitions rely on the linearity of the patterns in a chord, and on the understanding that using the same channel more than once in the same chord is forbidden.

As such, if a message arrives that satisfies a chord pattern whose state was 0, this becomes N and we may ascertain to be one step closer to a matching status. In contrast, this would not be the case in JCThorn because it supports non-linear patterns and because a message can satisfy multiple join patterns of the same join.

For example, assume that the matching state for the join defined in 4.4 is 00 and that the message {: `pairItem:''apple'' :`} has just been received. Considering that it satisfies both patterns, the state would become NN. This means that the join can fire, but clearly this is not right because a message can only match a single pattern in a successful match. It is for this reason that the deterministic automata approach is not

appropriate for JCThorn.

```
1  receive{
2    {: pairItem:X :} and {: pairItem:Y :} => { /*pair constructed*/ }
3  }
```

Listing 4.4: JCThorn program that illustrates why deterministic automata cannot be used

The second approach was proposed by Hubert Plociniczak, author of JErlang [35], and is based on the Rete[1] algorithm. It uses two types of test functions – alpha reduction and beta reduction functions.

Alpha-reduction in JErlang consists of a set of test functions, one for each join pattern, which are tested when a new message arrives. To avoid unnecessary computation, JErlang runs only the alpha tests belonging to the currently tested join.

On the other hand, beta-reduction consists of a set of test functions, one for each subset of join patterns of each of the joins. Listing 4.5 shows a *gen_joins* program that declares a join with four patterns. The corresponding beta-reduction functions are presented in 4.6.

```
1  handle_join( {operation, Id, Op} and {num, Id, A}
2    and {num, Id, B} and {num, Id, C}, Status) when (A > B) -> ...
```

Listing 4.5: *Gen_joins* declaration of a four-pattern join

```
1  BetaFunctions =
2    [ fun([{operation, Id, _}, {num, Id, _}], _) ->
3        true end,
4      fun([{operation, Id, _}, {num, Id, _}, {num, Id, _}], _) ->
5        true end,
6      fun([{operation, Id, _}, {num, Id, A}, {num, Id, B},
7      {num, Id, _}], Status) when (A > B) ->
8        true end ]
```

Listing 4.6: Beta-reduction functions for the join in figure 4.5

Notably, JErlang joins also support non-linear patterns and allow patterns of the same join to match non-disjoint sets of messages. These two features are also crucial in the JCThorn language. Consequently, the algorithm we use is similar to Rete, although we do not create beta-reduction tests explicitly.

Alpha reductions closely correspond to the local matching phase described in section 4.3.3. On the other hand, beta reductions map to the context matching phase 4.3.4. We present the algorithm in this form because it will make the description of the optimisations more comprehensible.

[1] http://en.wikipedia.org/wiki/Rete_algorithm

4.3.2 Runtime

At runtime, the interpreter creates a `ComponentThread` object for each spawned component. Each `ComponentThread` holds a mailbox from which it retrieves the messages (aka `Letters`) that other components have send it. When the interpreter's execution engine encounters a receive node (`RecvNode`) when traversing the AST, it calls the `recv` function defined in the `ComponentThread` class on the component object in execution.

This `recv` function takes, as arguments, the `RecvNode` node and the `frame`, holding amongst others the identifiers' seals. It consists of three main steps:

1. Initialize the matching state for that particular receive
2. Scan the mailbox until match or timeout
3. If match, remove the matched letters from the mailbox, execute the body of the join and return. If timeout, execute the timeout action and return. Otherwise, execute step 2 forever.

The first step creates a `ComponentJoinState` object for each join in the current receive. This object holds, for each pattern in the corresponding join, a list of messages that have been satisfied so far. `ComponentJoinState` objects belong to `ComponentThread` objects, although they are based on syntactic objects, ie. each join object in the current `RecvNode`. The UML class diagram in 4.6 illustrates this association.

Figure 4.6: Association between `ComponentThread` and `ComponentJoinState` objects

The second step is a bit more interesting. It scans the mailbox until it finds a combination of messages that satisfy a join in the current receive. The resolution has to obey the semantics presented in 3.4.1, which in a nutshell, consists of:

- All the messages in the mailbox at a point in time have to be checked against higher priority joins before attempting to reduce those with lower priority
- Joins at the same level of priority have to be checked in order, according to their position in the program
- *First-match* semantics regarding messages in the mailbox for joins at the same level of priority. The first successful combination of messages in the mailbox has to be returned, that involving messages in the forefront

Algorithm 1 gives the pseudo-code for the fragment responsible for scanning the mailbox. This is based on the original Thorn algorithm, but we omit timeouts for simplicity.

The list *joinsByPrio* is created at compile time, and contains one element for each group of joins with the same priority. Each group is itself a list that mimics the order that joins are defined in the program.

Three levels of iteration are needed to ensure that it complies with the resolution semantics. The rest of the code is rather self-explanatory, with the exception of the `JoinMatcher.match` function. `JoinMatcher` is the class responsible of actually evaluating the joins. As we can see, its `match` function takes the join to evaluate, the matching state of the component concerning that join and the current letter and frame. This function is divided into two phases, local and contextual, presented below.

Algorithm 1 Fragment responsible of scanning the mailbox

```
 1: frame ← current frame
 2: joinsByPrio ← joins from current RecvNode
 3: for joinsAtCurrPrio in joinsByPrio do
 4:   for letter in mailbox do
 5:     for join in joinsAtCurrPrio do
 6:       state ← ComponentJoinState object of current join
 7:       matchedLetters ← JoinMatcher.match(join, state, letter, frame)
 8:       if matchedLetters not null then
 9:         mailbox.remove(matchedLetters)
10:         join.executeBody()
11:       end if
12:     end for
13:   end for
14: end for
```

4.3.3 Local matching phase

The *local matching phase* is the first stage in the execution of the `JoinMatcher.match` function, and closely corresponds to the alpha-reduction functions of the RETE algorithm. It focuses on the current message (ie. the `letter` passed as an argument) and the current join (also an argument). In our algorithm, this is the only phase with an associated matching state.

For each pattern in the current join, it tests whether the `letter` could potentially match that pattern in a successful reduction of the join. If that's the case, it updates the `state`, ie. the `ComponentJoinState` object passed as an argument. More precisely, it adds the `letter` to the `matchedLetters` list corresponding to that pattern (recall figure 4.6).

This phase is termed *local* because it restricts the examination to a single pattern at a time, without taking into account non-linear dependencies between patterns in the same join or the situation where a `letter` could satisfy a multiple of those patterns. As a result, the scenario where all the patterns in a join have been locally satisfied does not imply that there exists a successful match.

The matching of algebraic patterns in this phase required the creation of a new pattern matcher. This matcher only determines whether the `letter` can match a given algebraic pattern, but does not bind the contents of the `letter` to the variables defined in the pattern. In contrast, the original pattern matcher in THORN performs both tasks.

Listing 4.7 will help to illustrate the reason why the original matcher cannot be used. If binding were performed, the value of a message M, say 3, satisfying the first pattern, would bind to the variable `x`. Then, when performing the test for the second pattern, M would not be added to the associated state for this pattern because `!($(x))` would only match values distinct from 3. Clearly, this situation is not desired because we cannot know at this stage what value `x` would eventually hold on a successful match. In other words, we also want to add the message M to the list of potential matches for the second pattern in the join.

```
1  receive{
2    x and !($(x)) and [$(x)] => { ... }
3  }
```

Listing 4.7: Non-biding pattern matcher

The careful reader might have noticed that even when the non-biding matcher is used, we will still not add the message M to the list of potential matches for the second pattern. Considering that `x` will not be bound to any value when testing the second join pattern, we would assume that message M matches the algebraic pattern `$(x)`. Therefore, M will not match `!($(x))`, essentially failing again to add M to the list of potential matches.

The missing ingredient to solve this problem is the three-valued logic that the non-binding matcher uses. When this matcher encounters an *interpolation* pattern, of the form `$(...)`, or an *evaluation test expression* pattern, of the form `(...)?`, it returns that it is undefined whether the match is or not successful. Recall that `$(e)` matches the value of the expression `e` and `(e)?` matches if the boolean expression `e` evaluates to true.

As a consequence, more complex patterns are translated into standard ternary-logic operators. The conjunctive patterns of the form `p && q` map to the $\wedge$ operator, whose truth table is given in 4.2. The disjunctive patterns (`p || q`) map to the $\vee$ operator and the negation pattern (`!p`) to the $\neg$ operator, shown in tables 4.3 and 4.4 respectively.

		p		
		TRUE	**FALSE**	**UDF**
q	**TRUE**	TRUE	FALSE	UDF
	FALSE	FALSE	FALSE	FALSE
	UDF	UDF	FALSE	UDF

Table 4.2: Truth table for the expression $p \wedge p$

		p		
		TRUE	**FALSE**	**UDF**
q	**TRUE**	TRUE	TRUE	TRUE
	FALSE	TRUE	FALSE	UDF
	UDF	TRUE	UDF	UDF

Table 4.3: Truth table for the expression $p \vee p$

p	$\neg$***p***
TRUE	FALSE
FALSE	TRUE
UDF	UDF

Table 4.4: Truth table for the expression $\neg\, p$

4.3.4 Contextual matching phase

The second phase in the execution of the `JoinMatcher.match` function is the *contextual matching phase*, that is similar to the beta-reductions performed by the RETE algorithm, with the difference that we do not store matching information for this phase (we only create it on the fly when needed). This phase is responsible of finding a combination of messages that can satisfy a join. It takes into account non-linear dependencies between patterns in the same join and ensures that a given message matches at most one pattern in a successful match.

Algorithm 2 gives the pseudo-code for a simplified version of the recursive function that performs the contextual matching. When called from `JoinMatcher.match`, it is given the value `0` as the *index* argument and an empty set as the *partialSolution* argument.

The *index* stands for the position of the pattern in the join we are currently testing. Thus, when its value reaches the size of the join (its number of join patterns) we can return true because a successful combination of messages has been found. In that case, the successful combination would be stored in the *partialSolution* set. Obviously, the algorithm returns as soon as it finds a successful combination of messages.

For each pattern in the join, the *matchedLetters* variable will hold the potential matches that were found during the local matching phase. The `BindingPatternMatcher.match` function is responsible for performing the algebraic pattern matching. This is the original THORN matcher that also binds values to variables.

The call to the `BindingPatternMatcher.match` function is not completely correct. The reason is that this function takes an algebraic pattern, not a join pattern, as an argument. Considering that each join pattern can define as many as three algebraic patterns, the same number of calls might be needed. The first algebraic pattern in a join pattern matches the contents of the message, and the other two optional patterns match the sender and the envelope respectively.

Algorithm 2 `boolean contextualMatch(`*join*, *state*, *index*, *frame*, *partialSolution*`)`

```
 1: size ← number of patterns in the join
 2: if index = size then
 3:   return true
 4: else
 5:   joinPattern ← pattern at position index of join
 6:   matchedLeters ← list of letters for joinPattern in state
 7:   for currLetter in matchedLetters do
 8:     if currLetter not in partialSolution then
 9:       if BindingPatternMatcher.match(joinPattern, currLetter, frame) then
10:         partialSolution.add(currLetter)
11:         if contextualMatch(join, state, (index+1), frame, partialSolution) then
12:           return true
13:         else
14:           partialSolution.remove(currLetter)
15:         end if
16:       end if
17:     end if
18:   end for
19: end if
20: return false
```

4.4 Chords implementation

Chords in JCThorn are no more than syntactic sugar. When parsing a JCThorn program, an initial AST is created that closely resembles the structure of the program and contains nodes for chord declarations, chord bodies and channel declarations. Then, a new pass of the compiler translates these chords into joins. In this section, we describe how the translation is performed and explain why the `catch` is difficult to translate.

4.4.1 Translation

Thorn translates programs with channel declarations and `serve` statements into programs with `receive` constructs and functions. We first walk through this translation with an example, and then describe the changes that had to be made in order to support chords in JCThorn.

Thorn translation walkthrough

The program in 4.8 shows a server holding a database of customers. Clients store new customer entries by sending a record through, alarmingly enough, the `store` channel. They can also retrieve customer records by id using the `retrieve` channel.

```
component DatabaseServer(){
  var customers := table(id){name;address;};

  async store({: id:I, name:N, address:A :}) {
    customers.ins({: id:I, name:N, address:A :});
  }

  sync retrieve(Id){
    return customers(Id);
  }

  body{
    serve;
} }
```

Listing 4.8: THORN program used to walk through the translation of chords

Channel declarations are translated into functions, as demonstrated by listing 4.9. The identifiers of the functions are based on the channel names plus some added characters. Asynchronous channels map to functions taking three arguments: the sender of the message, the message's envelope and the list of formals declared by the channel. The body of the functions are identical to the body of the corresponding channels.

On the other hand, the functions that translate synchronous channels take a nonce as an extra argument, which is used by clients to match requests with replies. The body of the function returns a record with two fields, the response and the nonce. The response field holds the value returned in the channel body.

Regarding the `serve` statement, this is translated into a function call to a `CserveC` function taking four arguments, as illustrated in 4.10. The first two arguments hold functions whose bodies are, if defined, the blocks appearing next to the before and after clauses respectively. Otherwise, they would be just `null`. The third argument is the timeout given in microsecond and the fourth is a function holding the code to be executed when the timeout has been reached.

The body of the `CserveC` function holds a `receive` construct with one case for each channel declared in the component. Hence, in the translation of the database server, there are two cases – one for the `store` and other for the `retrieve` channel.

```
fun CstoreC(sender, enve, [{: id:I, name:N, address:A :}]) {
  customers.ins({: id:I, name:N, address:A :});
  }

fun CretrieveC(sender, msg, nonce, [Id]) {
  CresponseC = customers(Id);
  return {: response: CresponseC, nonce: nonce :};
}
```

Listing 4.9: Translation of channel declarations

```
1
2   fun CserveC(beefore, aafter, timN, timCmd) {
3     receive {
4       {: async:``store", arguments:args :}
5       from sender envelope enve => {
6           if (beefore != null){
7             beefore(enve, sender);
8           }
9           CstoreC(sender, enve, args);
10          if (aafter != null){
11            aafter(enve, sender);
12          }
13      }
14      | {: sync:``retrieve", arguments:args, nonce:nonce :}
15        from sender envelope enve => {
16          if (beefore != null){
17            beefore(enve, sender);
18          }
19          CresponseC = CretrieveC(sender, enve, nonce, args);
20          if (aafter != null){
21            aafter(enve, sender);
22          }
23          sender <<< CresponseC;
24      }
25      timeout (timN) { timCmd(); }
26    }
27  }
28
29  body{
30    CserveC(  fn (_,_) = null,   /* no before */
31              fn (_,_) = null,   /* no after */
32              null,              /* no timeout */
33              fn () = null );    /* no timeout body */
34  }
```

Listing 4.10: Translation of serve statement

The store case matches a record with two fields. The field `async` matches the identifier of the channel and the second field defines the variable `args` to which the arguments passed to the channel will bind to. This case also matches the message's sender and envelope. The case body executes the before actions if defined and then calls the function `CstoreC`, which translates the channel declaration, with the correct arguments. Finally, it executes the after clause if defined.

The case for the synchronous `retrieve` channel is quite similar. The only differences are that the record defines a third field holding the nonce, and that the server returns a response (`CresponseC`) to the client.

```
1   /*
2    * TRANSLATION OF:
3    * body{
4    *  server <-- store({:id:343, name``David", address:``Poplar":});
5    * }
6    */
7
8   fun hlStore (receiver, args) {
9     receiver <<< {: async:``store" , args: args :};
10  }
11
12  body{
13     hlStore(server,
14        [{:id:343, name:``David", address:``Poplar":}]);
15  }
```

Listing 4.11: Translation of asynchronous send statement

```
1   /*
2    * TRANSLATION OF:
3    * body{
4    *  reply = server <-> retrieve(343);
5    * }
6    */
7
8   fun hlRetrieve (receiver, args, timN, timCmd) {
9     CnonceC = newNonce();
10    receiver <<< {:async:``retrieve" , args: args, nonce:CnonceC:};
11    receive{
12      {: response: CresponseC, nonce: $(CnonceC) :}
13      from $(receive) => {
14        CresponseC;
15      }
16      timeout (timN) { timCmd(); }
17    }
18  }
19
20  body{
21     reply = hlRetrieve(server, [343], null, null);
22  }
```

Listing 4.12: Translation of synchronous send statement

Obviously, high-level send operations also get translated on the client side. Listings 4.11 and 4.12 illustrate the translation for both synchronous and asynchronous send operations. A function is created which is then called from the point in the program where the operation appeared. This function executes a low-level send operation and, in the synchronous case, a receive immediately after.

JCThorn translation

In order to support chords, JCThorn introduces a number of modifications to Thorn's translation. We will describe the main differences by considering the two chords declared in 4.13.

```
1  sync getNewName() and async update({: name:N :}) => { ... }
2  async sendLetter(L) and async update({: address:A :}) => { ... }
```

Listing 4.13: Example used to illustrate the translation of chords

Each chord declaration also becomes a function whose identifier is the concatenation of its defined channel names (listing 4.14). Like in Thorn, these functions take four arguments if the chord declares a synchronous channel and 3 otherwise. However, the arguments are lists whose elements are, respectively, the senders, envelopes and arguments of the messages matched by the chord.

```
1  fun CgetNewName_updateC( [syncSender, asyncSender0],
2      [syncEnve, asyncEnve0], [[{: name:N :}]], none ) { ... }
3
4  fun CsendLetter_updateC( [asyncSender0, asyncSender1],
5      [asyncEnve0, asyncEnve1], [[L], [{: address:A :}]] ) { ... }
```

Listing 4.14: Translation of chord declarations

Similarly, the serve statement is translated into a call to a CserveC function. Its receive construct consists of a set of joins instead of cases, with one join for each chord. This is illustrated in listing 4.15. We can observe that the `arguments` field of each record declares a conjunctive pattern instead of a simple wildcard pattern (ie. the variable `args`). Actually, this is also the case for the from and envelope algebraic patterns, but we omit it here for simplicity.

To understand why we need conjunctive patterns, consider the scenario where the mailbox holds, in this order, the messages:

1. {:sync:``getNewName", nonce:543254:}
2. {:async:``sendLetter", arguments:[``Dear ...'']:}
3. {:async:``update", arguments:[{:address:``Picadilly'':}]:}.

When the third message is retrieved, our solution ensures that it is the second join that correctly fires and not the first. The reason is that the value `[{:address:``Picadilly'':}]` only matches the pattern `[{:address:A:}]`, declared on the second join, but not the pattern `[{:name:N:}]`, declared on the first. Obviously, the left conjuncts are needed to be able to pass the arguments in the body of the joins to the corresponding function, either `CgetNewName_update_C` or `CsendLetter_update_C`.

In contrast, if we had followed Thorn design to just use the wildcard pattern `args`, the first join would have been satisfied. In that case, the second join pattern of the first

join would be {:async: ‘‘update’’, arguments:args:} instead of {:async: ‘‘update’’, arguments:(asyncArgs0 && [{:name:N:}])}. As such, it would incorrectly match the message {:async:‘‘update", arguments:[{:address:‘‘Picadilly’’:}]:} and, as a consequence, we would be calling the function Cget_put_C with the wrong arguments.

```
1   receive{
2
3     {:sync:‘‘getNewName", nonce:} from syncSender envelope syncEnve
4     and {:async:‘‘update", arguments:(asyncArgs0 && [{:name:N:}]):}
5     from asyncSender0 envelope asyncEnve0 => {
6
7       if (beefore != null) {
8         beefore([syncEnve,asyncEnve0], [syncSender,asyncSender0]);
9       }
10      CresponseC = CgetNewName_update_C([syncSender, asyncSender0],
11          [syncEnve, asyncEnve0], [asyncArgs0], nonce);
12
13      if (aafter != null) {
14        aafter([syncEnve,asyncEnve0], [syncSender,asyncSender0]);
15      }
16      syncSender <<< CresponseC;
17    }
18
19    |
20
21    {:async:‘‘sendLetter", arguments: (asyncArgs0 && [L]):}
22    from asyncSender0 envelope asyncEnve0
23    and {:async:‘‘update", arguments:(asyncArgs1 && [{:address:A:}]):}
24    from asyncSender1 envelope asyncEnve1 => {
25
26      if (beefore != null) {
27        beefore([asyncEnve0,asyncEnve1], [asyncSender0,asyncSender1]);
28      }
29      CresponseC = CsendLetter_update_C([asyncSender0, asyncSender1],
30          [asyncEnve0, asyncEnve1], [asyncArgs0, asyncArgs1], nonce);
31
32      if (aafter != null) {
33        aafter([asyncEnve0,asyncEnve1], [asyncSender0,asyncSender1]);
34      }
35    }
36
37    timeout (timN) { timCmd(); }
38  }
```

Listing 4.15: Translation of serve statement

In addition, there is a further difference between the translations. This relates to the `before` and `after` functions. As we can see, in JCThorn the arguments are lists and not single elements. The reasons have already been exposed in section 3.5.1.

4.4.2 Catch translation problems

We have decided not to include the `catch` clause next to `serve` statements for the reasons presented in section 3.5.2. Furthermore, the decision to drop it has also been influenced by the complex implementation that would be required.

The `catch` clause in Thorn is used to define exception handlers to deal with the situation where a message has been sent on an existing channel with the wrong argument. Its implementation relies on calling a function with the wrong arguments, which throws an exception that will trigger the execution of the handlers.

Looking at listing 4.16, that contains the relevant fragments from the translation of the database server (listing 4.8), we can more precisely describe how the implementation operates. Any message on the channel `store` will match its corresponding case of the `receive` defined inside `CserveC`. The reason is that the wildcard pattern `args` matches any value, including the string `‘‘apples’’`. However, the call to `CstoreC` will fail in that situation because `‘‘apples’’` is not a record with the format that `CstoreC` expects, throwing the required exception.

```
1   fun CserveC(beefore, aafter, timN, timCmd) {
2     receive {
3       {: async:‘‘store”, arguments:args :}
4       from sender envelope enve => {
5           ...
6           CstoreC(sender, enve, args);
7           ...
8       }
9       ...
10    }
11  }
12
13  fun CstoreC(sender, enve, [{: id:I, name:N, address:A :}]) { ... }
```

Listing 4.16: Fragments from the translation of the database server (4.8)

In JCThorn, considering that the `arguments` field defines more restrict patterns (the conjunctive patterns we have described), we need a slightly different approach which is presented in 4.17. `‘‘Apples’’` sent through `store` will not match its corresponding entry in the receive because it expects and argument of the form `[{:id:I, name:N, address:A :}`. As a result, we have added an extra case to the `receive` in order to catch the error. The catch case ensures that the message has been sent on the existing channel `store` (positive conjunct) and that it contains the wrong arguments (negative conjunct). However, this approach does not always work.

```
1   fun CserveC(beefore, aafter, timN, timCmd) {
2     receive {
3       {: async:``store", arguments:(args &&
4       [{:id:I, name:N, address:A:}]) :}
5       from sender envelope enve => {
6           ...
7           CstoreC([sender], [enve], [args]);
8           ...
9       }
10      ...
11      /* Catch case */
12
13      | !({:async:``store", arguments:[{:id:I, name:N, address:A:}]:})
14      && {: async:``store", arguments:args :} => {
15        /* throw required exception */
16  } } }
```

Listing 4.17: JCThorn catch case

```
1   /*
2    * catch case for chord:
3    *  sync wait(x) and async continue($(x)) { ... }
4    */
5
6   fun CserveC(beefore, aafter, timN, timCmd) {
7     receive {
8       ...
9       /* Catch case */
10
11      | !( {:sync:``wait", arguments:[x]:}
12           || {:async:``continue", arguments:[$(x)]:} )
13      && ( {:sync:``wait", arguments:args0:}
14           || {:async:``continue", arguments:args1:} ) => {
15        /* throw required exception */
16  } } }
```

Listing 4.18: Catch case throwing a compile time exception

The solution presented fails in those situations where there are non-linear dependencies between the channels defined in the same chord. Listing 4.18 shows the catch case for one of such chords. Considering that the chord declares two channels, each conjunct of the pattern is a disjunction of two patterns.

The problem with this example is that it will not compile. The compiler will throw an *undefined variable exception* because it detects that the variable `x` in the pattern `$(x)` has not been defined. In other words, `x` will not be bound to any value when attempting to evaluate that pattern. We could probably fix this problem by performing a static analysis of the program. However, we have decided to eliminate the `catch` clause next to `serve` statements because it also leads to unexpected results, as presented in 3.5.2.

4.5 Summary

We have covered the implementation of JCThorn in this chapter. First, we described the changes that had to be made to the grammar in order to support joins and chords. The incorporation of these constructs also required the introduction of new nodes in the AST.

Then, we examined the implementation of joins in more detail. We gave an overview of the interpreter's runtime environment and the state that needs to be associated with each component in order to store matching information. We also discussed in detail both the algorithm responsible of scanning the mailbox and the algorithm responsible for the evaluation of joins.

In the evaluation of joins, we determined that JCThorn cannot follow the finite-state automata of languages like JoCaml. JCThorn proposes an algorithm similar to Rete (used in JErlang) that executes in two phases. The local phase restricts the examination to a single pattern at a time without taking into account non-linear dependencies. In contrast, the contextual phase is responsible for finding a combination of messages that can satisfy all the patterns in the join and their dependencies.

On the other hand, the implementation of chords consisted of extensive changes to the translation that is performed at compile time. This translation converts the nodes in the AST that represent chords into nodes for function declarations, function calls and `receive` statements. We have also presented why translating `catch` clauses next to `serve` statements is an intricate exercise.

So far we have only covered the basic algorithm framework. In the next chapter we develop the algorithm further and present the optimisations that aim to minimise the negative effects of the state explosion problem. In other words, we study how to reduce the number of message combinations that may need to be tested during the contextual matching phase.

Chapter 5

State explosion problem and optimisations

The performance of the basic algorithm presented in chapter 4 is unacceptable when the size of the mailbox is large, in terms of the number of messages received. The root of this problem lies with the explosion in the number of possible matching states, which is documented in section 5.1. In a nutshell, this refers to the huge number of message combinations that might be tested before a successful sequence is found.

To deal with this problem, JCThorn borrows ideas from other languages and also proposes a number of novel techniques. The successful optimisations included in the language are presented in section 5.2. However, not all the optimisations provided positive results. Those that were not finally included in JCThorn are described in 5.3.

Overall, the optimisations have been very effective at combating the state explosion problem (quantitative results given in section 6.4). The performance of JCThorn when handling large mailboxes greatly exceeds that of languages with an equivalent level of complexity such as Jerlang.

The good results achieved are clearly the outcome of the time spent on fine-tuning the algorithm. We attached a lot of importance to performance because we believe that nobody will use a language, no matter how expressive, if it does not perform well. As such, we wanted to prove that joins can be implemented efficiently.

5.1 State explosion problem

The state explosion problem has already being studied in the context of join resolution by Luc Maranget and Fabrice Le Fessant [24], authors of the JoCaml language. However, this problem gets worse for JCThorn because it allows greater freedom in the definition of joins.

As presented in section 4.3.1, JoCaml bases its solution in finite-state automata. The matching state of chords is stored in a vector that has one element for each defined channel, which can take the values 0 or N. A *matching status* is a status that holds

enough N, so that at least one chord can fire.

The finite-state automata solution relies on the fact that the increase in the number of N elements in the vector is inversely proportional to the distance to a matching state. In other words, if the values of two elements in the vector have become N (when they were previously 0) we are two steps closer to a matching status. This allows us to bound the number of possible states to 2^c, where c is the number of defined channels, as presented in [24].

In contrast, JCThorn does not guarantee that relation due to non-linear dependencies and the possibility of a message satisfying multiple patterns of the same join. For example, we can have two messages that could independently match each of the two patterns in a join, but do not satisfy the non-linear dependencies between the two patterns. As a result, the upper bound on the number of states is different in JCThorn.

This upper bound cannot be computed statically, and depends on the runtime state of the program. Let c be the number of patterns in a join, and $matches(i)$ the set of letters that have been matched by the pattern at position i. More precisely, $matches(i)$ is the set of potential matches stored on the `ComponentJoinState` object and computed during the local matching phase. Given these definitions, we can compute the upper bound as follows:

$$upper\_bound(c) = \prod_{i=1}^{c} | \, matches(i) \, |$$

or in big O notation:

$$O(n^c)$$

where n is the maximum length of any of the $matches(i)$ sets.

This is the worst case scenario and represents the maximum number of different combinations that might be tested during the contextual phase. As an example, consider the join declared in listing 5.1. A board game will only start when the three elements needed to start a particular game are available. Assume that at a particular point in the execution, the matching state is:

$$\begin{aligned}
matches(1) &= \{\, players(\text{“}go''),\ players(\text{“}chess''),\ players(\text{“}checkers'')\,\} \\
|\, matches(1) \,| &= 3 \\
\\
matches(2) &= \{\, board(\text{“}stratego''),\ board(\text{“}monopoli''),\ board(\text{“}backgammon'')\,\} \\
|\, matches(2) \,| &= 3 \\
\\
matches(3) &= \{\, players(\text{“}risk''),\ pieces(\text{“}life''),\ pieces(\text{“}mastermind''), \\
&\qquad pieces(\text{“}reversi'')\,\} \\
|\, matches(3) \,| &= 4
\end{aligned}$$

It is clear that in that scenario, none of the games can be started because not all of the three required elements are available. The upper bound is the total number of combinations of players, board and pieces, ie. $3 \times 3 \times 4 = 36$.

```
1    sync players(x) and async board($(x)) and async pieces($(x)) {
2      /* start game */
3    }
```

Listing 5.1: Elements of a board game

When messages can satisfy more than one pattern in the join, the upper bound is slightly lower. For the chord in 5.2, if there are three `son()` messages in the mailbox, the upper bound is 3! instead of 3^3. The reason is that each `son()` message belongs to the three $matches(i)$ sets, but a single message cannot satisfy more than one pattern in a successful match.

```
1    async son() and async son() and async son() {
2      /* triplets born */
3    }
```

Listing 5.2: Birth of triplets

Consequently, the state explosion problem will significantly impact those programs with large mailboxes, in terms of the number of messages received, in which the $matches(i)$ sets are likely to be bigger. In order to reduce the problem, two types of optimisations have been implemented which either:

1. prune the search space or
2. use heuristics to skip both the local and contextual matching phases or just the latter

Optimisations like *fail fast* or *combinations with current message only* belong to the first group and optimisations like *repeating receive* and *skip contextual* to the second. We believe that the second class of optimisations are novel to JCThorn because other languages have only focused on pruning the search space.

5.2 Successful optimisations

In this section, we describe the successful optimisations that have been included in the language. Some of the optimisations are mutual complementary and some are built on top of each other. As such, the order of presentation does not follow the order of importance. However, we would like to highlight that the *repeating receive* and *skip contextual* optimisations have been particularly effective in the case of large mailboxes.

5.2.1 Fail Fast (FF)

Fail Fast avoids testing combinations that start with an invalid sequence of messages. Sometimes we can detect that a combination will not be successful only by testing the first few patterns in the join. At that point, we can prune all the branches that start with that sequence.

In the board game example (listing 5.1), we can detect that there are not valid combinations (in the given situation) only by testing the first two patterns in the join because they have to agree on which game to be played. Likewise, instead of testing each combination involving the first two patterns four times, once for each message matched by the third pattern, we can just test it once. Even for this simple example, this optimisation reduces the number of operations from 36 to $3 \times 3 = 9$.

5.2.2 Combinations with Current Message Only (CCMO)

This optimisation avoids testing the same combinations of messages over and over again when scanning the mailbox. During join resolution, the mailbox is scanned in order until a valid combination of messages is found. As an example, assume that the fourth message in the mailbox has been tested but no successful combination found. When checking the fifth message it would be pointless to check again the combinations only involving the four first messages. In this case, *CCMO* ensures that only combinations involving the fifth message are checked.

CCMO has been implemented by modifying the `contextualMatch` algorithm given in section 4.3.4. For each pattern in the join, apart from the set of potential matches, we also need to record if they satisfy the current message or not. The algorithm ensures that those combinations of messages involving the current message are checked first. It also has to guarantee that it only fails after all possible combinations involving the fifth message have been checked, which is trickier than it appears.

5.2.3 Repeating Receive and Context Independence (RRCI)

RRCI's goal is to reuse the acquired matching information in previous executions of the same `receive` or previous executions of `serve` statements. More precisely, RRCI allows to start the scan of the mailbox from the last checked message in the previous execution of the same `receive`.

For instance, imagine that when executing `receive` A we have tested the first 9 messages and no match was found. Subsequently, we test the tenth message and discover that a three pattern join has been satisfied. At that point, we remove the three matched letters from the mailbox, execute the body of the join and continue with the rest of the program. At a later stage, we encounter `receive` A again. In most situations we could start the scan of the mailbox from the eight message if no other `receive` has been executed in between. Clearly, we already know that the first seven messages do not match any join in A because they have been tested in the previous execution of A.

RRCI relies on the fact that there is only one thread per component, and that messages can only be removed from the mailbox by this thread when executing a `receive` construct. This optimisation addresses those programs containing the common pattern illustrated in listing 5.3. Servers frequently answer the same type of requests over and over again, possibly preceding or following each reply by some actions (`<statements_before>` and `<statements_after>` respectively). RRCI is effective when those actions do not contain `receive` constructs themselves.

```
1   while(<condition>){
2
3     <statements_before>
4
5     receive{
6       ...
7     }
8
9     <statements_after>
10  }
```

Listing 5.3: Receive inside while

Furthermore, RRCI is also effective when executing `serve` statements without `receive` statements interleaved. In the example in 5.4, RRCI will apply to all the `serve` statements except the one at line 8. We know that each serve statement is translated into a function call to a `CserveC` function containing a single `receive` statement. Hence, in all the cases the `receive` statement that gets executed is the same. However, at line 8 we cannot reuse the matching information because we have just executed a different `receive` at line 7.

```
1   while(<condition>){
2     serve;
3   }
4   serve;
5   x = 5;
6   serve;
7   receive{ ... }
8   serve;
9   serve;
```

Listing 5.4: Repetition of serve statements

Moreover, RRCI in this form will only be used when all the joins defined in the given `receive` are context independent. We say that a **join** is **context independent** if it defines all the variables that it uses. For example, the first join in the authentication server in 5.5 is not context independent because it matches the value hold in variable `password`, which is defined outside the join. On the other hand, the second and third joins are context independent. By generalisation, we say that a **receive** is **context independent** if all its joins are. Consequently, the `receive` in 5.5 is not context independent.

```
password = ``1g?_m8@2cv'';
receive{
  {:enter:$(password):} and {:reenter:$(password):} => {
    /* allow access */
  }
  | {:enter:X:} and {:reenter:$(X):} => {
    /* deny access */
  }
  | {:enter:X:} and {:reenter:Y:} => {
    /* try again */
  }
}
```

Listing 5.5: Program that performs authentication of users

Context independence ensures that when testing the same messages in the mailbox for a second time against the same join we obtain the same results. Therefore, performing the tests once is enough and we can skip them thereafter. When the join is context dependent this is not the case.

As an example, consider the server in 5.6. A client will not be allowed to execute step 1 before the server has been initialised. If a client attempts to do that by sending the message {:execute:1:}, the server should wait until it has been initialised and only then execute the first step. To be able to execute it, it is clear that it needs to compare the message {:execute:1:} against the pattern {:execute:$(step):} for a second consecutive time. Hence, we cannot skip the tests in this situation.

```
var step := 0;
while(step < 10){
  receive{
    {:execute:$(step):} => {
      /* do something */
      step := step + 1;
    }
    | ``initialise'' => {
      /* initialise */
      step := 1;
} } }
```

Listing 5.6: Context dependent receive

At this point, we can characterise the two modes of operation of RRCI more precisely:

- If a receive is context independent, we can start the scan of the mailbox from the letter that follows the last checked message in the previous execution
- If a receive is not context independent, we should start the scan from the first message in the mailbox. However, we are still allowed to skip the tests for those joins that are context independent

In order to be able to determine if a join is context independent, we need to perform a static analysis of the program at compile time. Basically, we have to construct two sets of variables, *uses* and *defines*, and make sure that all the variables in the former are also in the latter.

Moreover, the implementation of RRCI also requires that each component remembers which `receive` statement was executed last. Then, when the component starts executing a new receive, it might be allowed to reuse some of the collected information in the previous run only if the current `receive` is the same as the last executed `receive`. How much of this information will be allowed to be reused depends on the context independence property.

5.2.4 Just-in-time Update of State

As a consequence of the RRCI optimisation, we have to update the matching state for each of the joins in the current receive every time a successful combinations of messages is found. Apart from removing the matched letters from the mailbox, we also need to remove them from the list of potential matches for each of the patterns of each of the joins. In implementation terms, these are the lists stored in each `ComponentJoinState` object described in figure 4.6 of section 4.3.2.

However, it might be wasteful to update the state of every join after a successful match, considering that we might not even execute the same `receive` again. A better approach would be to only update the state when needed. Fortunately, there exists a method to achieve that.

In our implementation, we store a flag with each of the letters to specify whether or not it is still valid. A letter becomes invalid when it has been removed from the mailbox as a result of a join firing. Given this information, we can postpone updating a `ComponentJoinState` object until it is really needed.

That time is when we are about to start the contextual matching phase for the corresponding join. Specifically, this is just before starting the execution of the recursive function `contextualMatch` given in algorithm 2 of section 4.3.4. In a naive implementation, it would mean that we will update the state every time we execute the function `JoinMatcher.match` when scanning the letters in the mailbox (recall algorithm 1 of section 4.3.2). Nonetheless, the *SC* optimisation described in the next section allows us to considerably reduce the number of times the update is required.

Updating the state for a join means that for each of its patterns, iterate over the list of potential matches and remove those letters that are invalid. As we have already said, these lists are stored in its corresponding `ComponentJoinState` object.

5.2.5 Skip Contextual Phase (SC)

SC is a very simple optimisation that produces very positive results. It relies on some heuristics to skip both the update of state and contextual matching phase or just the latter. Two variations of SC are performed at two different stages.

During the local matching phase, we collect information regarding what patterns have been previously satisfied by messages in the mailbox and about what patterns have been satisfied by the current message. When the local matching phase has finished, we will only move to the next step (ie. the just-in-time update of the state) when both of these conditions are met:

1. All the patterns in the join have been previously satisfied by at least one message
2. At least one pattern in the join contains the current message as a potential match

When condition 1 does not hold, it is safe to skip the contextual phase because we know that a successful combination cannot be found. When condition 2 is false, we know that all the possible combinations have been tested when scanning the previous messages in the mailbox. In other words, the current message does not add new combinations because it does not satisfy any of the patterns in the join.

A slightly different version of SC is performed after the just-in-time update of state and before the contextual matching phase. At this point, we will only proceed to the contextual phase if both of these conditions hold:

1. Again, if all the patterns in the join have been satisfied by at least one message
2. The number of distinct messages in the sets of possible matches for the patterns in the join is at least as big as the size of the join

At this point we need to check condition 1 again because, after updating the state,some letters might have been removed. As such, condition 1 will not longer hold if all the potential matches of a pattern were invalid letters when testing this for the first time.

The justification for the second check relates to messages satisfying multiple patterns of the same join. If we recall the program in 5.2 about the birth of triplets, we can illustrate why. The join defined in that programs expects three messages with the same shape, ie. `son()`. However, if only one such message is present in the mailbox, this will still be added to the list of potential matches for each of the patterns in the join. Clearly, we cannot find a matching combination at that point because we need three messages, not one. Nonetheless, if this check were not performed we would still try to find one in this case.

5.2.6 Rank Reordering of join Patterns (RR)

RR is an optimisation proposed by Christian Holzbaur et al in the context of Constraint Handling Rules [19] and later included in JErlang. In JCThorn, RR complements the fail fast optimisation by moving to the front the patterns in the join that define more constraints. This allows us to detect invalid combinations sooner.

RR uses the rank of a pattern, which gives an indication of how much of its variables are shared when deciding the order of the patterns in the join. At compile time, the rank of a pattern is computed by performing a simple static analysis of the join. This analysis first counts the number of occurrences of each variable in the join and then calculates the rank of each pattern by summing the occurrences of each of its variables, as illustrated in 5.7.

```
1  receive{
2    [x,$(x),y] and [$(x),$(y),z] => { ... }
3  }
4
5  /*
6   * occurrences(x) = 3
7   * occurrences(y) = 2
8   * occurrences(z) = 1
9   * rank([x,$(x),y]) = 2 * occurrences(x) + occurrences(y) = 8
10  * rank([$(x),$(y),z]) = 6
11  */
```

Listing 5.7: Calculation of the rank of join patterns

At that point, the patterns will be arranged according to their rank. The algorithm picks the pattern H with highest rank, includes its depend-upon patterns in decreasing order of rank, then H itself and finally its dependant patterns. It then repeats this process until all the patterns have been included. The depend-upon patterns are the patterns that define variables which H uses. Conversely, the dependent patterns are the patterns that use variables that H defines or uses. Listing 5.9 demonstrates this transformation for the join given in 5.8 as written by the programmer.

```
1  z and $(z) and x and $(x) and [$(x),y,$(y),$(y)] and $(y) => {...}
```

Listing 5.8: Join as written by the programmer

```
1  x and [$(x),y,$(y),$(y)] and $(y) and $(x) and z and $(z) => {...}
```

Listing 5.9: Join after its patterns have been reordered by rank

In 5.8, the first four patterns only say that there has to be two pairs of equal messages in the mailbox. By contrast, the first four patterns in 5.9 expresses that there is a pair of equal messages in the mailbox, such that the value of that message also appears as the first item of a four-element list (whose last three elements are the same) and, moreover,

that their value also appears as a single message in the mailbox.

Clearly, the first four patterns in 5.9 define more constraints that those in 5.8. As a result, the reordering will usually allow us to detect invalid sequences sooner as it is normally harder to satisfy 5 constraints than it is 2. In 5.9, if there does not exist a list which meets those conditions, we will not bother about finding pairs of equal messages in the mailbox. On the other hand, in 5.8 we would need to test all the pairs of messages in the mailbox until we realise that there is not such a list.

The reader at this point might be thinking, "Why not then move the list pattern to the front?" as shown in 5.10 in order to be able to detect that there is not such a list sooner. This is done by swapping the interpolation pattern involving `x` in the list (ie. `$(x)`) with a wildcard pattern (ie. `x`), and performing the inverse substitution with the pattern in the join that previously defined `x`. While this is the optimal solution in this example, this approach is problematic in the general case.

```
1  [x,y,$(y),$(y)] and $(y) and $(x) and $(x) and z and $(z) => {...}
```

Listing 5.10: Optimal reordering of patterns

Interpolation patterns can match the value of more complex expressions as shown in 5.11. Reordering the patterns in this case (listing 5.12) would involve calculating the inverse expression of $(5 \times x)^{12}$, which is $\sqrt[12]{x} \div 5$. Not only is this transformation complex, but also undesirable. Any performance benefits gained due to the reordering will most likely be lost when calculating the root, which is a far more expensive operation than exponentiation. For consistency and time constraints, we have decided not to implement this more complex reordering in any case.

```
1  x and y and [$((5*x)^12), $((5*y)^12)] => {...}
```

Listing 5.11: Interpolation patterns involving more complex expressions

```
1  [x,y] and $(sqrt_base12(x)/5) and $(sqrt_base12(y)/5) => {...}
```

Listing 5.12: Undesired reordering of patterns

5.2.7 Uniquely Satisfied Patterns (USP)

USP is an optimisation whose goal is to reduce the search space. Its domain of action are those joins that define patterns matching sets of messages related by subset inclusion. When the more specific patterns only satisfy one message in the mailbox, it is pointless to attempt to find combinations in which those messages are mapped to the more general patterns.

As a motivating example, consider the join in 5.13. Clearly, the wildcard patterns `x` and `y` are more general than the list pattern `[z]`. Precisely, `x` and `y` can match any message but `[z]` can only match singleton lists. If we assume that there are 5 integers

in the mailbox and only a singleton list we know that the latter will need to match the third pattern.

However, in the worst case we might need to test as many as 20 combinations of messages until we map the list to the third pattern (each of the first two patterns satisfy 5 messages each, so $5 \times 4 \times 1 = 20$ given that each message can only be used once). On the other hand, if we had reserved the list for the third pattern before the start of the search, we would ensure that the first tested combination is successful in that situation. This is precisely what USP does.

```
1   x and y and [z] => {...}
```

Listing 5.13: Example that motivates the use of the USP optimisation

While the scope of USP might be narrow, it does make a difference when it is applicable. We have included it in JCThorn because its complexity is very low. The number of operations that USP performs is linear with the number of patterns in the join, which is normally very small. Moreover, USP is only performed just before the search of combinations (ie. the execution of the function `contextualMatch`). As such, and thanks to the SC optimisation, we know that the number of times that it is actually executed is low.

5.2.8 Single pattern joins

When a join only defines a single pattern we can perform the tests against the messages in the mailbox once instead of twice (one in the local phase and one in the contextual phase). The contextual phase is only needed to find a successful combination of messages when joins match more than one. In the event that a join only matches a single message, we can skip the contextual phase altogether.

5.3 Failed optimisations

In this section we describe a couple of optimisations that did not produce the expected results. Naturally, these are not included in the final version of the JCThorn language and they are only presented here because they provide valid insight about the resolution of join patterns.

5.3.1 Letter Contents Cache (CC)

After profiling the execution of a program that solves the single-lane bridge problem, we discovered that an important percentage of the execution time was spent on the local matching phase (exactly 10%). Given that this number is considerably high, we attempted to reuse the matching information of messages with equal contents in order to skip this phase.

It is common that many of the messages found in a component's mailbox have equal

contents. Considering that two messages with equal contents produce the same results when executing the local matching phase, we can perform the tests once and reuse this information when checking the second message. In order to accomplish this, we need an efficient mechanism for identifying the contents of the messages and a map between content identifiers and matching information.

To identify the contents, we use a cache storing the last N messages retrieved from the mailbox so that we limit the memory requirements. This cache is a hash map that associates contents with identifiers and it is shared by all the joins in the current receive. When retrieving a message from the mailbox for the first time, we calculate its hash to determine if a message with equal contents have been recently retrieved. If so, we assign it the identifier stored in the cache and otherwise, we create a new fresh identifier and update the cache. On the other hand, when retrieving an old message we use the corresponding identifier that has previously been calculated.

Each identifier is an integer (for which a hash can be computed efficiently) that it is used to retrieve the matching information from a second cache that also stores information about the last N elements. It is also a bounded hash map, but in this case each join in the current receive has its own. If matching information for the given identifier is found in this cache, we use it to update the matching state of the join and skip the local matching phase.

This optimisation was successful at reducing the percentage of the execution time spent on the local matching phase. This dropped from 10% to 2.4% for the single-lane bridge program. However, it did not produce noticeable performance improvements overall, so we decided to discard it from the final version of JCThorn to reduce its complexity. The reasons why no gains were achieved were most likely related to the associated overheads and the fact that we still need to update the matching state for the patterns.

5.3.2 Runtime reordering

Patterns in a join can also be reordered at runtime according to how many messages in the mailbox they satisfy. This was one of the first optimisations attempted, but was later abandoned because it conflicted with the rank reordering presented in 5.2.6 which we believe produces better results.

The runtime reordering is based on the number of matches found during the local phase and stored in the state for each of the patterns in a join. Before proceeding to the contextual matching phase, the patterns are reordered in increasing order of matches. As a motivating example, consider the join in 5.14 and assume that the mailbox holds the messages `[1]`, `[2]`, `4` and `5`. At that point the patterns `x` and `y` match all four messages while the patterns `[x]` and `[v]` only match two. Hence, they would be reordered as shown in 5.15. This reordering avoids testing combinations with the list messages mapped to the wildcard patterns. The logic behind is the same as for the *uniquely satisfied patterns* optimisation (section 5.2.7).

```
1   x and y and [z] and [v] => { ... }
```

Listing 5.14: Join as defined by the programmer

```
1   [z] and [v] and x and y => { ... }
```

Listing 5.15: Join after being reordered at runtime

However, we believe that the number of constraints that a pattern defines (rank reordering) is more relevant than the number of messages it satisfies (runtime reordering). In other words, moving the patterns that define more constraints to the front will increase the likelihood of allowing us to detect invalid combinations sooner. Moreover, the rank reordering also has the advantage that it is performed only once at compile time, while the runtime reordering would have to be executed every time we start a search for combinations of messages.

5.4 Summary

In this chapter we have introduced the state explosion problem, which essentially consists of the large number of message combinations that may be tested before a successful match is found. We have characterised the worst case to be in $O(n^c)$, where n is the maximum number of potential matches for any pattern in the join, and c is the size of the join, ie. the number of join patterns.

Then, we presented what optimisations are included in JCThorn in order to reduce the effects of the state explosion problem. Some optimisations, like *Fail Fast*, prune the search space to reduce the number of combinations that are tested during the contextual matching phase. Others, like *Skip Contextual*, avoid trying to find a combination when we know before hand that one does not exist, essentially skipping the contextual matching phase altogether.

Not all the optimisations were successful, and we give an example of two that did not finally made it into the final version of JCThorn. These are the *Letter Contents Cache* and the *Runtime Reordering*.

In the next chapter we will evaluate how this optimisations influence the performance of the JCThorn language, both in typical concurrent problems and in a benchmark that measures the performance in situations where the mailboxes grow large. We also provide a comprehensive analysis of the expressiveness of the language and comment on other factors such as scalability, correctness and integration with the original Thorn language.

Chapter 6

Evaluation

How does JCThorn fare in comparison to Thorn in terms of its usefulness and relative performance? The purpose of this report was to investigate and ultimately answer this critical question. In this chapter we present a thorough study of both expressiveness (section 6.1) and performance (6.4), that will provide greater insight into the answer. We conclude with our belief that there is indeed demand for joins and that the performance of JCThorn is rather good, particularly when handling large mailboxes.

Worthy to note is that the following system was used in all of the experiments presented in this chapter:

Lenovo Y430
Intel Core2 Duo, P7450, 2.13GHz
4GB RAM
Windows 7 Professional, 64-bit version

6.1 Expressiveness

The expressiveness which joins provide is a key reason why they appeal to programmers. In this section, we first present a number of problems that we believe are easier to solve in JCThorn than in Thorn. We also describe cases where programmers may abusively use joins and create more complex and difficult to read solutions. We then attempt to precisely identify and characterise situations in which it is more convenient to use joins. Finally, we illustrate how explicit numeric priorities affect the resolution of joins and the expressiveness of the language.

6.1.1 Solutions enhanced by joins

Joins can make the solution of some problems simpler and as such, we present a selection of problems in this section which are easier to solve in JCThorn than in Thorn.

Board Game problem

We define the *Board Game* problem as the problem of finding the first combination of messages appearing in the mailbox that represent the three required elements needed to start a specific board game. These three elements are the players, the pieces and the board. We have invented this problem because it clearly shows the limitations of the Thorn language while it has a trivial solution in JCThorn, as shown in listing 6.1.

```
1  component boardGameServer{
2    body{
3      receive{
4        {:players:X:} and {:board:$(X):} and {:pieces:$(X):} => {
5          /* start game of X */
6  } } } }
```

Listing 6.1: Solution to the Board Game problem in JCThorn

This solution ensures that no other combination appears before in the mailbox. In other words, the last message of the solution found will appear before the last message of any other potential solution. For example, if we assume that the mailbox contains the messages listed below, in the given order, this solution will start a game of draughts.

1. {:board:‘‘chess’’:}
2. {:players:‘‘draughts’’:}
3. {:pieces:‘‘chess’’:}
4. {:board:‘‘draughts’’:}
5. {:pieces:‘‘draughts’’:}
6. {:players:‘‘chess’’:}

Conversely, writing a solution for the Board Game problem in Thorn is harder and prone to errors. A programmer could be tempted to write the `boardGameServer` in listing 6.2. This program has three nested `receive` statements in order to match each of the elements and considers all possible permutations to try to match the first combination appearing in the mailbox. However, it does not return the first successful combination in every situation.

For the mailbox presented above, it will start a game of chess instead of draughts. The reason is that the first message in the mailbox is {:board:‘‘chess’’:} and, thus, it is bound by the outer `receive`. At that point, it will only try to find the remaining two

elements needed to start a chess game and will ignore elements from other games. Worse still, if it fails to find the remaining two elements, a deadlock results.

A valid solution to the Board Game problem in THORN that finds the first successful combination is presented in listing 6.3. We observe that it is significantly more complex than the solution in JCTHORN. It has to maintain three lists – one for each type of element needed. While scanning the mailbox, it adds the `current` message to the corresponding list until a valid combination is found. This is detected when the three lists contain the corresponding element for the `current` game. Finally, we have to return all the messages that have been retrieved but not matched to the mailbox.

Unfortunately, this solution does not preserve the order of the messages in the mailbox since the retrieved messages will be appended to the end when returning them to the mailbox, and this fact might be pertinent in other problems. Clearly, JCTHORN's solution is not only superior because it has reduced the programmer's effort from 50 lines to 6, but also because it preserves the order in the mailbox of the messages that have not been matched.

Dining Philosophers problem

The *Dining Philosophers* problem is probably the most well-known problem in Computer Science, and certainly in the field of concurrent programming. It was posed and solved by Dijkstra in 1965 and, since then, everyone inventing a new synchronisation primitive has demonstrated how elegant it is by solving the dining philosophers problem. Hence, it is a good problem to attempt to tackle for us.

The problem states that five philosophers are seated around a circular table, and that each has a plate of Spaghetti. Between each pair of plates is one fork, so there are as many forks as plates. However, a philosopher needs two forks to eat the spaghetti. Thus, when he gets hungry, he tries to acquire both his left and right forks. If successful, he eats, then puts the forks down and thinks for a while until he gets hungry again.

A simple solution to the Dining Philosophers problem is achieved by introducing a waiter at the table[1]. The waiter, as shown in listing 6.4, receives requests from the philosophers before they take up the forks. Only when both of the forks that a philosophers needs are free does the waiter reply and allows him to eat. The JCTHORN solution relies on messages of the form {`:fork:Id:`} that tell the waiter that the fork with the given id is free. Then, we can easily match a philosopher request with his left and right forks by using the join in the body of the waiter. As a result of using a join, the waiter will only remove a request from the mailbox when he can satisfy it.

In contrast, the solution to the Dining Philosophers in THORN is more complex, as shown in appendix A.1. This solutions also uses a waiter, but in this case, he has to maintain two data tables – one holding information about the forks and one about the philosophers. These are needed because the waiter might remove requests from the mailbox when the required forks are not available. As such, we need to record which philosophers are waiting and which forks are or are not free. This fact considerably complicates

[1] http://en.wikipedia.org/wiki/Dining_philosophers_problem

the solution that more than doubles the number of lines of code required.

```
1  component boardGameServer{
2    body{
3      receive{
4
5        {:players:X:} => {
6          receive{
7            {:board:$(X):} => {
8              receive{
9                {:pieces:$(X):} => { /* start game of X */ }
10             }
11           }
12           | {:pieces:$(X):} => {
13             receive{
14               {:board:$(X):} => { /* start game of X */ }
15       } } } }
16
17       | {:board:X:} => {
18         receive{
19           {:players:$(X):} => {
20             receive{
21               {:pieces:$(X):} => { /* start game of X */ }
22             }
23           }
24           | {:pieces:$(X):} => {
25             receive{
26               {:players:$(X):} => { /* start game of X */ }
27       } } } }
28
29       | {:pieces:X:} => {
30         receive{
31           {:players:$(X):} => {
32             receive{
33               {:board:$(X):} => { /* start game of X */ }
34             }
35           }
36           | {:board:$(X):} => {
37             receive{
38               {:players:$(X):} => { /* start game of X */ }
39       } } } }
40 } } }
```

Listing 6.2: Invalid solution to the Board Game problem in Thorn

```
1   component boardGameServer{
2
3     fun contains(e, l) { ... }
4     fun remove(e, l) { ... }
5
6     body {
7       var players := [];
8       var boards := [];
9       var pieces := [];
10      var found := false;
11      var current := null;
12
13      while(!found){
14        receive{
15          {:players:X:} => {
16            players := [{:players:X:}, players... ];
17            current := X;
18          }
19          | {:board:X:} => {
20            boards := [{:board:X:}, boards... ] ;
21            current := X;
22          }
23          | {:pieces:X:} => {
24            pieces := [{:pieces:X:}, pieces... ];
25            current := X;
26        } }
27
28        if( contains({:players:current:}, players) &&
29            contains({:board:current:}, boards) &&
30            contains({:pieces:current:}, pieces) ){
31          found := true;
32      } }
33
34      remove({:players:current:}, players);
35      remove({:board:current:}, boards);
36      remove({:pieces:current:}, pieces);
37      for( p <- players ){
38        thisComp() <<< p;
39      }
40      for( b <- boards ){
41        thisComp() <<< b;
42      }
43      for( p <- pieces ){
44        thisComp() <<< p;
45      }
46
47      /* start game of X */
48  } }
```

Listing 6.3: Valid solution to the Board Game problem in THORN

```
1  spawn DiningPhil {
2
3    body{
4      component Waiter(guests){
5        body{
6          while(true){
7            receive{
8              {: phil:Id, name:N :} from P and {: fork: $(Id) :}
9              and {: fork: $((Id+1) mod guests) :} => {
10               P <<< {: left:Id, right:((Id+1) mod guests) :};
11     } } } } }
12
13     component Phil(n, id , waiter){
14       body{
15         while(true){
16           /* think */
17           waiter <<< {: phil:id, name:n:};
18
19           receive{
20             {: left:L, right:R :} => {
21
22               /* eat */
23               waiter <<< {: fork: L :};
24               waiter <<< {: fork: R :};
25     } } } } }
26
27
28     w = spawn Waiter(5);
29     for( F <- 0 .. 4 ){
30       w <<< {:fork:F :};
31     }
32
33     phils = [
34       spawn Phil("Kant", 0, w),
35       spawn Phil("Hume", 1, w),
36       spawn Phil("Marx", 2, w),
37       spawn Phil("Plato",3, w),
38       spawn Phil("Nietzsche",4, w) ];
39   }
40 }
```

Listing 6.4: Solution to the Dining Philosophers problem in JCThorn

Santa Claus problem

The *Santa Claus* problem [37] is a concurrency problem that has been solved in a number of languages supporting joins, such as JErlang [35] and Polyphonic C# [7]. However, we do not believe that this is a problem that significantly showcases the expressive power of joins since an alternative solution without them is not that much more complex.

In this problem, Santa Claus sleeps in the North Pole until it is awakened by a group of all nine reindeer or a group of three (out of ten) elves. Elves visit Santa anytime they have difficulties making the toys, and reindeer, just before Christmas. Thus, a group of reindeer has preference over a group of elves. However, when three elves are having their problems solved, any other group should not disturb Santa. Similarly, Santa cannot attend to anyone while he is delivering toys with the reindeer.

A simple solution in JCThorn is given in listing 6.5. Santa is defined with two joins, one for each type of worker, that match either nine reindeer or three elves. We can see that the reindeer have preference over the elves because they have been assigned an explicit numeric priority of 10. The rest of the solution is rather self-explanatory.

In contrast, a solution in Thorn has to introduce secretaries to marshal the groups of workers, as demonstrated in appendix A.3. There is one secretary for each type of worker, Robin and Edna, that behave as two buffers that fire when they reach the required size. We need these two secretaries because Santa's time is priceless and he should not waste time marshalling the groups. We observe that in this case, Thorn's solution in only about one third larger than JCThorn's solution.

```
1  spawn SantaClaus {
2    body{
3
4      component worker(type, santa){
5        body{
6          while(true){
7            santa <<< type;
8            receive{
9              "ok" => {
10               /* work */
11               santa <<< "done";
12     } } } } }
13
14     component santa(){
15       body{
16         var group;
17         while(true){
18           /* sleep */
19           receive{
20
21             "reindeer" from R1 and "reindeer" from R2
22             and "reindeer" from R3 and "reindeer" from R4
23             and "reindeer" from R5 and "reindeer" from R6
24             and "reindeer" from R7 and "reindeer" from R8
25             and "reindeer" from R9 prio 10 => {
26               group := [R1,R2,R3,R4,R5,R6,R7,R8,R9];
27             }
28
29             | "elf" from E1 and "elf" from E2 and "elf" from E3  =>{
30               group := [E1,E2,E3];
31             }
32           }
33           /* wake up */
34
35           for (e <- group){
36             e <<< "ok";
37           }
38
39           /* help workers */
40
41           for (e <- group){
42             receive{ "done" => {}};
43     } } } }
44
45
46     s = spawn santa();
47     rs = %[spawn worker("reinder", s) | for j <- 1 .. 9];
48     es = %[spawn worker("elf", s) | for j <- 1 .. 10];
49 } }
```

Listing 6.5: Solution to the Santa Claus problem in JCThorn

6.1.2 Solutions worsened by joins

There are situations in which programmers might try to use joins to solve problems when they are not really needed, essentially counteracting its usefulness. In this section we present one of those situations.

Single-lane Bridge problem

The *Single-lane Bridge* problem [27] is also a typical problem included in the curriculum of undergraduate computing students to demonstrate the intricacies of concurrent programming. It helps to illustrate properties such as safety and liveness, but we will only address the former for the sake of simplicity.

This problem states that a bridge over a river is only wide enough to permit a single lane of traffic. Consequently, cars can only move concurrently if they are moving in the same direction. A safety violation occurs if two cars moving in different directions enter the bridge at the same time.

To avoid safety violations, cars have to request the bridge for permission before they enter it. Only when it is safe to do so, would the bridge allow them to enter. The bridge can be programmed in Thorn as shown in listing 6.6. We observe that the bridge is initially empty, and once a car has entered, it moves to one of two states – either flow north or flow south. These represent the cars moving in the respective directions. Only when the bridge becomes empty again will the bridge allow cars in the opposite direction to pass.

The logic in Thorn's solution is very clear from the structure of the program. In contrast, JCThorn's solution, given in listing 6.7, is a bit more messy, harder to read and slightly longer. In this case, the bridge keeps a record of how many cars are currently crossing by sending to itself messages of the shape {`:south:N:`}, {`:north:N:`} or ‘‘`empty`’’. The joins defined in its body enforce the safety conditions, and the bridge only replies to requests when there are other cars crossing in the same direction or when the bridge is empty.

```
1  module BRIDGE{
2    component Bridge(){
3      var souths := 0;
4      var norths := 0;
5
6      fun flowSouth(){
7        while(souths > 0){
8          receive{
9            "enter_south" from s => {
10             souths := souths + 1;
11             /* allow south to enter */
12             s <<< "allowed";
13           }
14           | "exit_south" => {
15             /* record exit of south */
16             souths := souths - 1;
17     } } } }
18
19     fun flowNorth(){
20       while(norths > 0){
21         receive{
22           "enter_north" from s => {
23             norths := norths + 1;
24             /* allow north to enter */
25             s <<< "allowed";
26           }
27           | "exit_north" => {
28             /* record exit of north */
29             norths := norths - 1;
30     } } } }
31
32     body{
33       while(true){
34         receive{
35           "enter_south" from s =>{
36             souths := souths + 1;
37             /* allow south to enter */
38             s <<< "allowed";
39             flowSouth();
40           }
41           | "enter_north" from n  => {
42             norths := norths + 1;
43             /* allow north to enter */
44             n <<< "allowed";
45             flowNorth();
46 } } } } } }
```

Listing 6.6: Solution to the Single-lane Bridge problem in Thorn

```
1  module BRIDGE{
2    component Bridge(){
3      body{
4        self = thisComp();
5        self <<< "empty";
6
7        while(true){
8          receive{
9            {: south: N :} and "enter_south" from s  => {
10             /* allow south to enter */
11             M = N + 1;
12             self <<< {: south: M :};
13             s <<< "allowed";
14           }
15           | {: north: N :} and "enter_north" from n  => {
16             /* allow north to enter */
17             M = N + 1;
18             self <<< {: north: M :};
19             n <<< "allowed";
20           }
21           | {: south: N :} and "exit_south"  => {
22             /* record exit of south */
23             M = N - 1;
24             if(M == 0){
25               self <<< "empty";
26             } else {
27               self <<< {: south: M :};
28             }
29           }
30           | {: north: N :} and "exit_north"  => {
31             /* record exit of north */
32             M = N - 1;
33             if(M == 0){
34               self <<< "empty";
35             } else {
36               self <<< {: north: M :};
37             }
38           }
39           | "empty" and "enter_north" from n =>{
40             /* allow north to enter */
41             self <<< {: north:1 :};
42             n <<< "allowed";
43           }
44           | "empty" and "enter_south" from s =>{
45             /* allow south to enter */
46             self <<< {: south:1 :};
47             s <<< "allowed";
48 } } } } } }
```

Listing 6.7: Solution to the Single-lane Bridge problem in JCThorn

6.1.3 When to use joins

Up to this point, we have analysed a number of solutions to different synchronisation problems for both Thorn and JCThorn. Now, we will try to uncover which are the characteristics of the class of problems that are more easily solvable in JCThorn.

Firstly, we believe that joins should only be used to synchronise on the receipt of messages and not to store the state of a component. In the Single-lane bridge problem, JCThorn's solution uses messages to record how many cars are crossing the bridge. This approach is difficult to justify if we accept that the number of cars crossing can be more conveniently stored in a state variable. Moreover, the number of cars crossing is only known to the bridge, so it does not need to get this information from other components.

Regarding the Dining Philosophers problem in JCThorn, one could argue that the messages of the form `{:fork:Id:}` are only used to record the state and should only be known to the waiter. However, these messages are also used by the philosophers to inform the waiter that he has finished using them, so it does make sense that they are messages and not state variables. Moreover, avoiding the separation of state and communication reduces the effort associated with maintaining the state separately and ensuring its consistency. The overheads are shown in Thorn's solution.

In the Santa Claus problem, joins also avoid the separation of state and communication. We know that we need either nine reindeer messages or three elves messages, and joins express this idea concisely. In Thorn, we only match one message at a time, so we have to introduce additional state data structures to record how many messages of each type we have received.

Regarding the Board Game problem, and the matching sellers and buyers problem introduced in section 3.1, these clearly illustrate the limitations in expressiveness of the Thorn language. The analysis provided insights into which points are harder, or even impossible to express in Thorn, but straightforward in JCThorn. Some of them are hard to express in Thorn by themselves, and others might only be difficult when used in combination in situations, such as when:

- There exist non-linear dependencies between the messages that can form a successful combination. For instance, in the Board Game problem, the three elements have to belong to the same game.

- The first valid combination of messages in the mailbox has to be retrieved

- The relative position of the different type of messages in the mailbox that form a successful combination is undefined

- The order in the mailbox for the messages that have not been matched has to be preserved

Joins or chords?

We have just examined which is the class of problems that are more easily solvable in JCThorn. A question that is left to be answered is when to use joins and when chords. The choice is primarily stylistic, since both constructs have the same expressive power.

A program that uses joins can be translated into a program using chords as listings 6.8 and 6.9 show. We observe that we need to declare as many chords as joins appear in the receive, and define a new channel that in this case we named `port`. The algebraic patterns of a join become the patterns of the arguments to the channel. Moreover, case 1 shows that by explicitly matching the sender of a message, a chord can behave as if all the channels were synchronous.

```
1  component server{
2    body{
3      receive{
4        [x] from S1 and $(x) from S2 => {
5          /* case 1 */
6          S1 <<< x;
7          S2 <<< x;
8        }
9        | ``foo'' and x => { /* case 2 */ }
10 } } }
```

Listing 6.8: Program that uses joins

```
1    component server{
2      sync port([x]) and async port($(x)) from S2 {
3        /* case 1 */
4        S2 <<< x;
5        return x;
6      }
7
8      async port(``foo'') and async port(x) { /* case 2 */ }
9
10     body{
11       serve;
12   } }
```

Listing 6.9: How to write the program in 6.8 using chords

Programs using chords can also be easily translated into programs that use joins, as listings 6.10 and 6.11 illustrate. Each chord can be translated into a join that matches records with two fields. The first field is the channel name of a given chord pattern and the second the arguments. As a result, the choice between chords and joins is mostly stylistic. However, chords are a bit slower because they are translated at compile time into joins, so performance might also influence the decision between the two.

```
1  component server{
2    sync get(x) and async put($(x)) from S2 { /* case 1 */ }
3
4    async get(y) and async remove($(y)) { /* case 2 */ }
5
6    body{
7      serve;
8  } }
```

Listing 6.10: Program that uses chords

```
1  component server{
2    body{
3      receive{
4
5        {:channel:``get'', args:x:}
6        and {:channel:``put'', args:$(x):} => { /* case 1 */ }
7
8      | {:channel:``get'', args:y:}
9        and {:channel:``remove'', args:$(y):} => { /* case 2 */ }
10 } } }
```

Listing 6.11: How to write the program in 6.10 using chords

6.1.4 Numeric priorities

As we have seen in section 3.4.1, JCThorn provides two different methods of increasing the priority of a particular join. The first can be termed as *positional priority* and relies on the in-order evaluation of joins. The second, the *numeric priority*, explicitly assigns priorities to joins.

In positional priority, joins at the same level of numeric priority are evaluated in the order they are declared. When scanning each of the messages in the mailbox, we will test the current message against joins declared earlier first. On the other hand, all the messages in the mailbox will be tested against joins with a higher numeric priority before attempting to reduce joins with lower numeric priority.

An interesting question is what effect the different type of priorities have in the resolution of joins. We answer this question by analysing two solutions to the Santa Claus problem, one where reindeer are explicitly assigned a higher numeric priority, and the other where reindeer only have positional priority. The results are presented in figure 6.1.4.

This chart shows the average number of synchronisation points achieved in a 10-second run of the Santa Claus program. To calculate the average, 100 hundred runs were executed. The synchronisation points are divided between those involving reindeer and elves. While positional priority allows us to reach a higher number of total synchronisations than numeric priority, 2707 points against 1914, the number of synchronisations

points performed involving reindeer is considerably lower, 676 against 1288.

These results are not surprising if we consider how the resolution of joins operates (described in section 3.4.1). We can conclude that numeric priorities are much more effective at detecting events that have higher relevance, which in the case of the Santa Claus problem, is the existence of a group of nine reindeer.

Moreover, we believe that the different types of priorities in JCThorn increase the level of expressiveness of the language since the programmer can more precisely determine which behaviour he desires the program to have. In contrast, most of the languages with *Join Calculus* based constructs do not provide priorities at all (ie. non-deterministic resolution) and all of those that do only support positional priorities (for example JErlang).

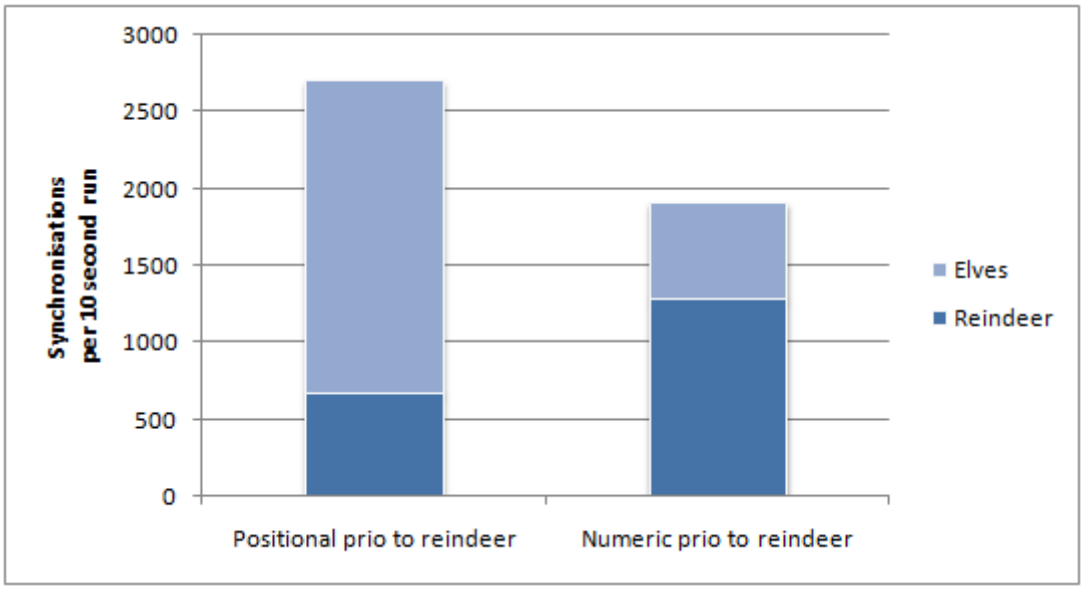


Figure 6.1: Synchronisations achieved by the different types of priority in the Santa Claus problem

6.2 Correctness

In order to evaluate the correctness of our extensions to the Thorn language, apart from certifying the correct behaviour of the multiple programs we have created, we also extended its test suite. The same number of tests in the original suite pass, with the exception of those involving the incompatible features between Thorn and JCThorn.

We have created multiple tests that have considerably eased the development of JCThorn. Both valid and invalid changes were quickly detected when tests stop or start passing respectively. As an example, the test in listing 6.12 ensures that numeric priorities work as expected. The strange statements that start with the symbols `~!@` ensure that joins fire in the right order. These are used to integrate the tests with JUnit and are inherited from Thorn.

```
1  spawn Sendie {
2    body{
3      ~!@testStartLatch();
4
5      sendie = thisComp();
6
7      rec = spawn Rec{
8        body{
9          receive{ "start" => {} }
10
11         var i := 2;
12         while (i > 0){
13           receive{
14             "apple" and "crumble" from z => {
15               ~!@phase(2, "crumble");
16               z <<< "done";
17               i := i - 1;
18             }
19             | "apple" from z and "pie" prio 10 => {
20               ~!@phase(1, "pie");
21               z <<< "done";
22               i := i - 1;
23             }
24             timeout(1000) {}
25     }   } } }
26
27     rec <<< "crumble";
28     rec <<< "apple";
29     rec <<< "apple";
30     rec <<< "pie";
31     rec <<< "start";
32
33     recv{ _ ={~!@hit()@!~1;}};
34     recv{ _ ={~!@hit()@!~1;}};
35
36     ~!@checkphase(
37       {: phase: 1, bag: ["pie"]:},
38       {: phase: 2, bag: ["crumble"]:}
39     )@!~  ;
40 } }
```

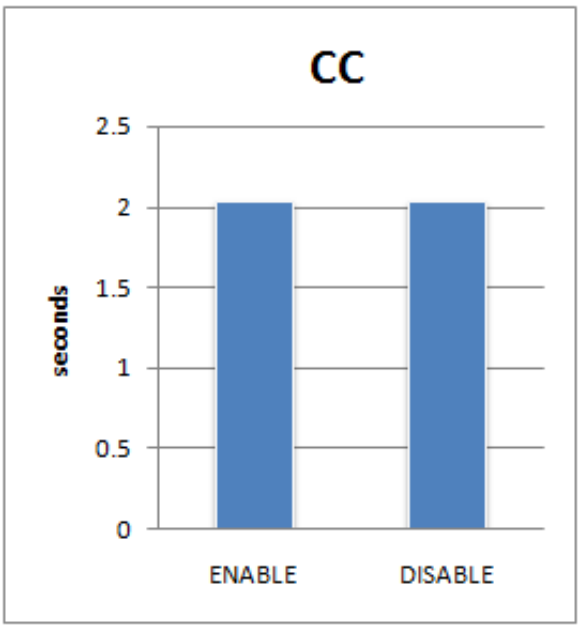
Listing 6.12: Example of a test from JCThorn's test suite

6.3 Microbenchmarks to test optimisations

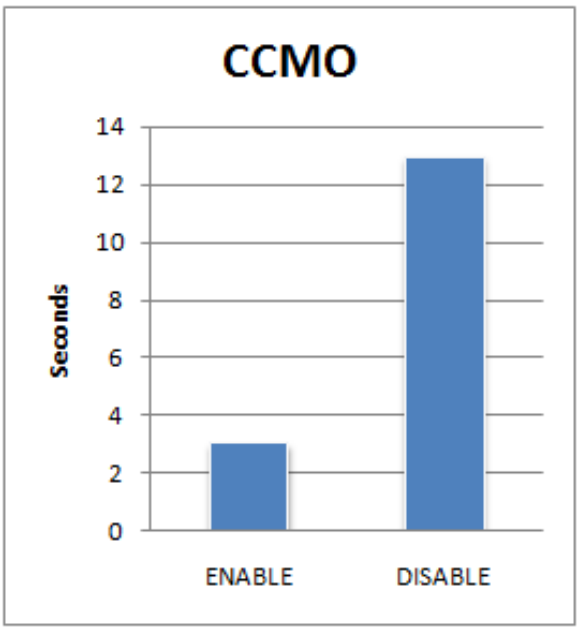
Microbenchmarks can give an idea of how the extent of the effectiveness the optimisations could provide. We have created some artificial tests in which they apply, and measured the period of time it took to reach a constant number of iteration points with both the optimisations enabled and disabled.

For example, the microbenchmark presented in listing 6.13 evaluates how effective the Rank Reordering (RR) optimisation can be. It is a client-server application in which the client repeatably sends the same set of messages to the server. The server, on the other end, tries to fire a join as often as possible. This join contains five patterns, the first four matching singleton lists and the fifth a two-element list. At compile time, if RR is enabled, the patterns in the join will be reordered.

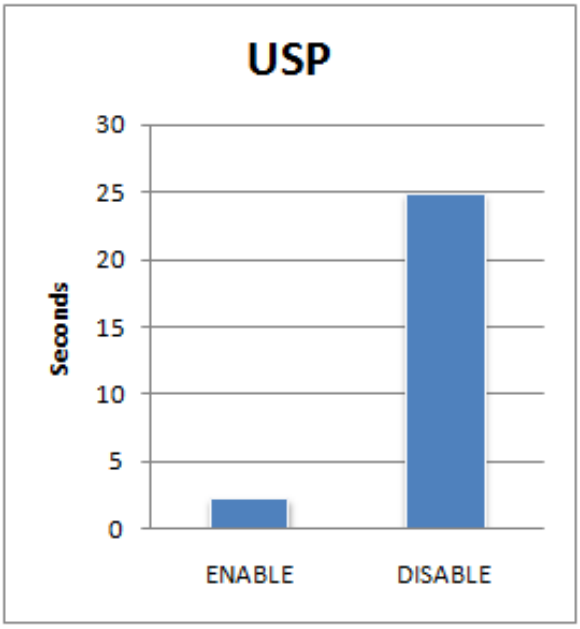
The results of executing this microbenchmark with RR both enabled and disabled are shown in chart *(e)* of figure 6.2. The chart shows the average time calculated by running the test 100 times. We observe that with RR enabled, it takes less than two tenth of a second to complete the test, while it takes over 90 seconds when disabled. The remainder of the charts correspond to running similar experiments to test other optimisations with the corresponding microbenchmarks given in appendix B. The reader should note that the scales are different from chart to chart.



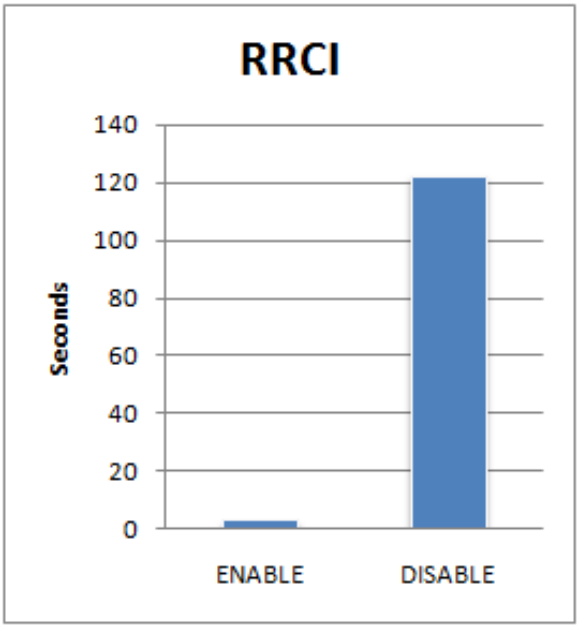
(a) Letter Contents Cache



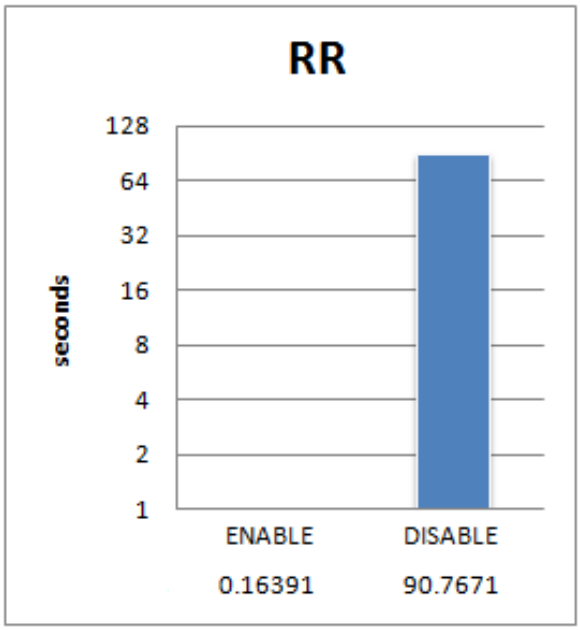
(b) Combinations with Current Message Only



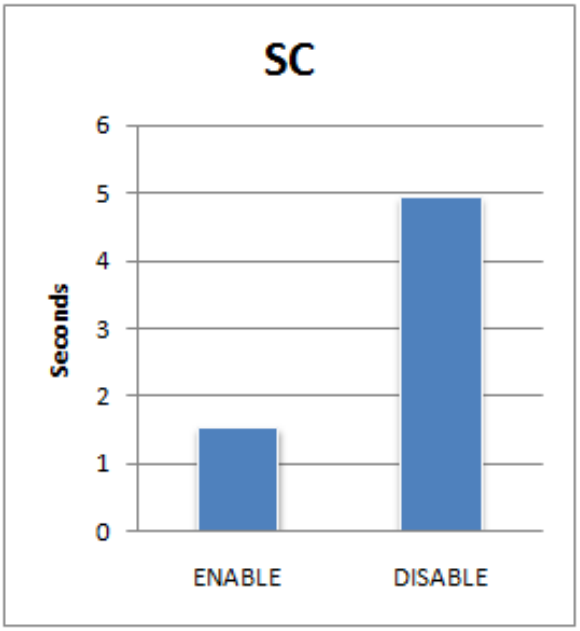
(c) Uniquely Satisfied Patterns



(d) Repeating Receive and Context Independence



(e) Rank Reordering



(f) Skip Contextual

Figure 6.2: Evaluation of some of the optimisations

```
1  spawn microRR {
2    body{
3      import FromJava.*;
4      ITER = 11;
5      outer = thisComp();
6      start = currentTime();
7
8      component client(s){
9        body{
10         /* Note, mailbox will overflow,
11         [10,11] and [5] not matched every round */
12         for(i <- 1 .. ITER){
13           s <<< [10,11];
14           s <<< [5];
15           s <<< [4];
16           s <<< [3];
17           s <<< [2];
18           s <<< [1];
19           s <<< [1,4];
20     } } }
21
22     component server(){
23       body{
24         for(i <- 1 .. ITER){
25           receive{
26             [x] and [y] and [z] and [v] and [$(x),$(v)] => {
27               /* do nothing */ }
28         } } }
29         outer <<< "done";
30     } }
31
32     s = spawn server();
33     c = spawn client(s);
34
35     recv{"done"  => {}};
36     end = currentTime();
37     execTime = (end - start) / 1000;
38     println("$execTime");
39 } }
```

Listing 6.13: Microbenchmark to test the Rank Reordering optimisation

6.4 Performance

Performance is the other critical factor that will have a direct and prominent impact on the acceptance of the extensions proposed by JCThorn. No matter how expressive the language, programmers will not use it if it makes applications slow. As a result, we devoted a lot of time into optimising and fine tuning the algorithm, which we believe has produced substantial improvements with regards to execution time.

We first analyse how each of the optimisations affect the solution of typical concurrent problems. Then, we present how solutions that use and do not use joins compare. We also examine the execution times of the same application when executed in Thorn's and JCThorn's interpreters. We conclude by revealing the results obtained from running a benchmark to measure how JCThorn performs when mailboxes grow large.

6.4.1 Effect of optimisations in typical concurrent problems

In section 6.3 we have analysed how the optimisations affect tailored microbenchmarks in order to understand the extent of their effectiveness. In this section we will show how they affect typical concurrency problems in situations where the mailboxes do not get very large.

Santa Claus

Figure 6.3 shows how disabling each of the optimisations in turn affects the execution time of JCThorn's solution to the Santa Claus problem, given in section 6.1.1 (with the numeric priority removed). We measured the time it took to perform 100 synchronisations, and calculated the average of 100 runs.

Surprisingly, the Contents Cache (CC) optimisation had no effect, although messages can only take two values – either "reindeer" or "elf". The reason is probably because of the CC related overheads and that we still need to update the matching state for each of the patterns. Disabling the Combinations with Current Message Only (CCMO) and Uniquely Satisfied Patterns (USP) optimisations did not increase the execution time because Skip Contextual (SC) ensures that whenever we test a combination, this one is successful. When SC is disabled, we try to find combinations when there are not enough messages in the mailbox, which explains the increase in execution time. The reason is that single messages match all the patterns in a join. Rank Reordering (RR) had no effect because all the patterns in each join have the same rank. Finally, the Repeating Receive and Context Independence (RRCI) optimisation had no perceivable effects because the mailboxes did not grow large.

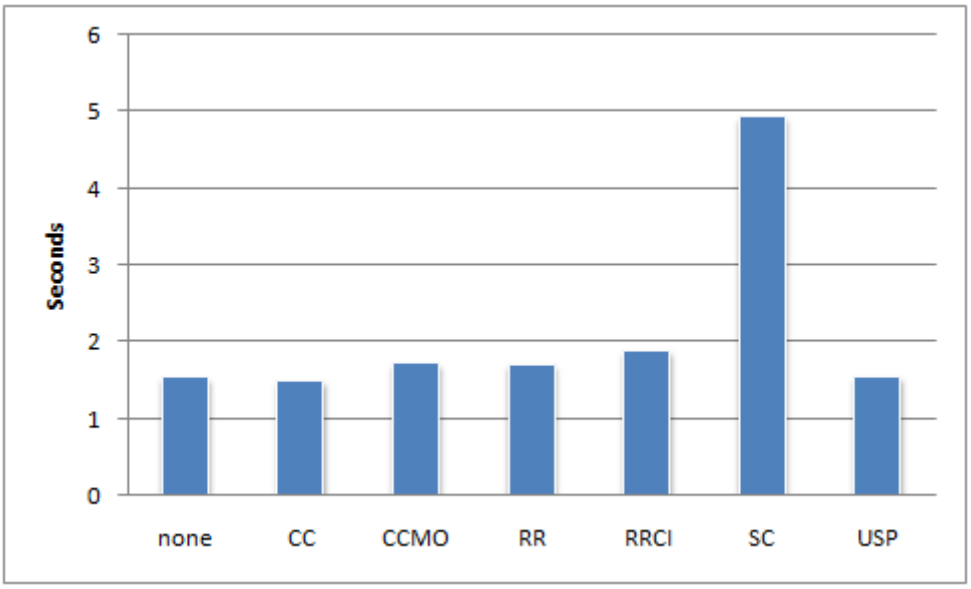


Figure 6.3: Effect of disabling each of the optimisations in turn on the Santa Claus solution

Single-Lane Bridge problem

Figure 6.4 shows the effects of disabling the optimisations on the Single-Lane Bridge solution written in JCThorn and presented in section 6.1.2. It shows the average time, in seconds, that 125 cars took to cross the bridge in each of the directions out of 100 runs. The reader should note that the scale is logarithmic.

We observe that disabling the RRCI optimisation increased the execution time from around 3 seconds to almost 128 seconds. The cause lies in the fact that requests from cars wanting to cross the bridge in the opposite direction of the cars currently in the bridge are queued. Thus, if the bridge does not remember the position of the last checked request, it will have to check all the queued requests after each reply.

Dining Philosophers problem

Finally, figure 6.5 shows the effects on JCThorn's solution to the Dining Philosophers problem, presented in section 6.1.1. We measured how long it took for the five philosophers to go through 50 thinking-eating cycles each, and present the average of 100 runs.

In this solution, disabling the optimisations did not have a perceivable effect on the execution time. For example, RR is not effective in the Dining Philosophers problem because there are only 5 philosophers at the table, so the number of requests that the waiter receives is very low. SC is not effective because although we might try to find combinations to satisfy a join when they are not enough messages in the mailbox, we would detect this situation when testing the first combination – only two patterns in the waiter's join can be satisfied by the same messages.

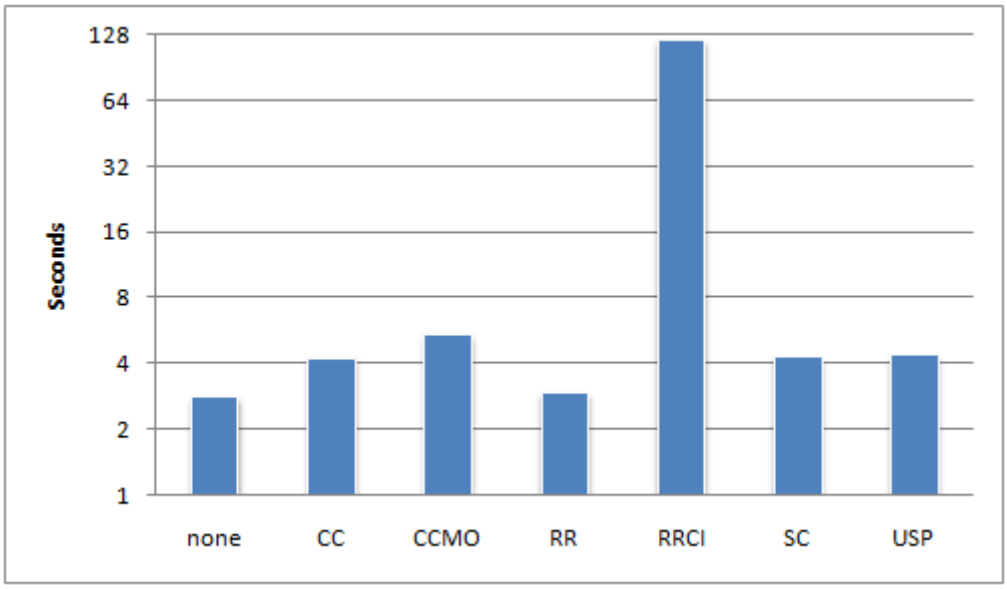


Figure 6.4: Effect of disabling each of the optimisations in turn on the Single-Lane Bridge solution

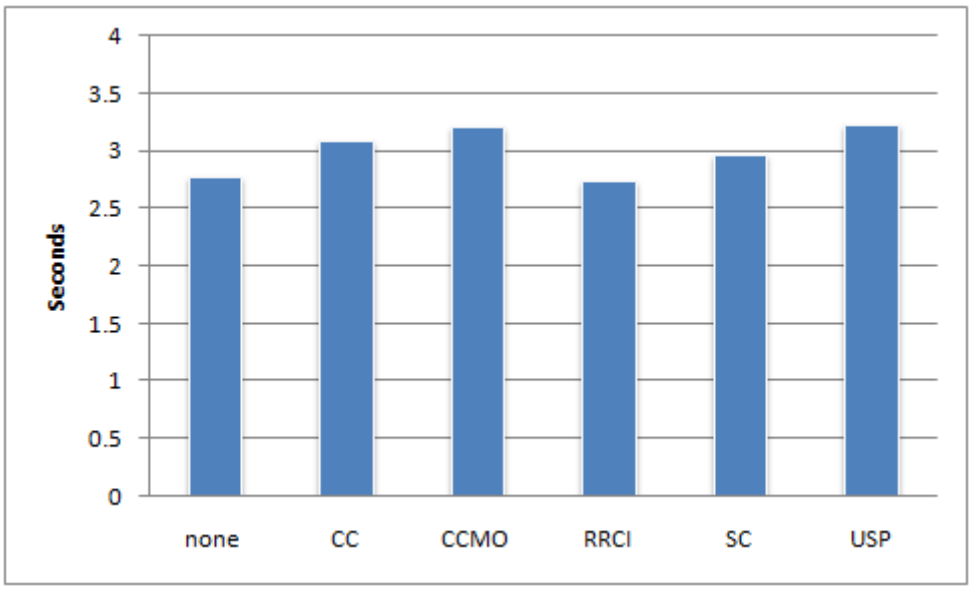


Figure 6.5: Effect of disabling each of the optimisations in turn on the Dining Philosophers solution

6.4.2 Thorn versus JCThorn

So far we have only analysed what effects the optimisations have had in join resolution. Now, we shall compare the relative performance of the original Thorn interpreter against the JCThorn interpreter, and how solutions with and without joins perform.

Santa Claus problem

Figure 6.6 shows the average time taken by different solutions of the Santa Claus problem to reach increasing numbers of synchronisation points out of 100 runs. Remarkably, when run in JCThorn's interpreter, the solution that does not use joins (given in appendix A.1) reaches the required number of synchronisation points in about two thirds of the time as compared to when it was executed in Thorn's interpreter.

We can explain this fact if we take into account that some of the optimisations introduced also affect programs that do not use joins. In particular, the repeating receive optimisation (RRCI) remembers the position in the mailbox of the last checked message in the previous execution of the same `receive`, which is also useful for those `receive` constructs that do not declare joins.

Regarding the solution that uses joins, presented in section 6.1.1, it runs slightly faster than the solution that does not. This might be the result of a combination of factors. For example, the solution using joins does not create two secretaries to marshal the groups of reindeer and elves, and it does not have to append one item to a list every time a message is received from one of the workers.

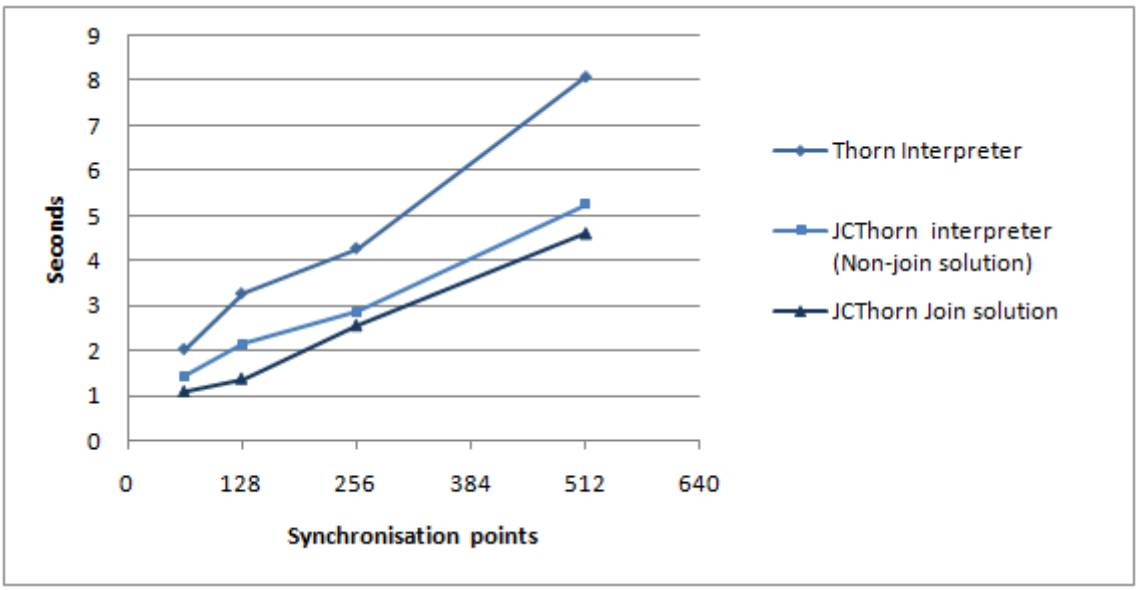


Figure 6.6: Comparison of the different solutions to the Santa Claus problem

Dining Philosophers problem

For the Dining Philosophers problem, JCThorn's interpreter is again faster than the original Thorn interpreter when executing the solution presented in appendix A.1, that does not use joins. The results are shown in figure 6.7 and presents the average time it took for the five philosophers to go through 50 thinking-eating cycles each out of 100 runs.

Disappointingly, the solution using joins (listed in section 6.1.1) runs over four times slower than the solution that does not. This is a consequence of the nature of join resolution. As we can observe from listing 6.14, the receive in the body of the waiter declares a join that matches a philosopher with its two corresponding forks. As explained in section 5.1, finding a valid combination is a relatively expensive operation, with a polynomial-time worst case.

On the other hand, listing 6.15 shows the receive in the solution that does not use joins. We observe that once we have received the request from a philosopher, ie. the message `{:take:Id, name:N:}`, we can find if the corresponding forks are available in constant time. The syntax `forks(Id)` efficiently retrieves the corresponding tuple from the table containing information about each of the forks. This is a constant time operation because the `ID` field has an associated index. Moreover, to discover if it is free, we use the syntax `V ~ P` that returns true if the value `V` matches the pattern `P`.

Returning to figure 6.7, we observe that the solution using chords (appendix A.2) is slightly slower than the solution using joins. This is probably the result of the extra work, like additional function calls, that is carried out in the translation.

```
1  receive{
2    {: phil:Id, name:N :} from P and {: fork: $(Id) :}
3    and {: fork: $((Id+1) mod guests) :} => {
4        ...
5  } }
```

Listing 6.14: Receive in the body of the waiter (Join solution)

```
1  receive{
2    {: take:Id, name:N :} from P  => {
3      if( forks(Id) ~ {:taken:false:}
4          && forks((Id+1) mod guests) ~ {:taken:false:} ){
5        ...
6  } } }
```

Listing 6.15: Receive in the body of the waiter (Non-join solution)

In view of the results, we can conclude that specialised solutions will normally perform better than the join resolution, as the latter has to work for the general case. However, joins allow programmers to solve certain problems with less effort, as it is the case for the Dining Philosophers problem. Thus, they will have to make compromises and evaluate

what applications are performance critical, and in what situations fast development is more important. Nonetheless, the important achievement of joins is that they provide programmers with choice, which is always valuable.

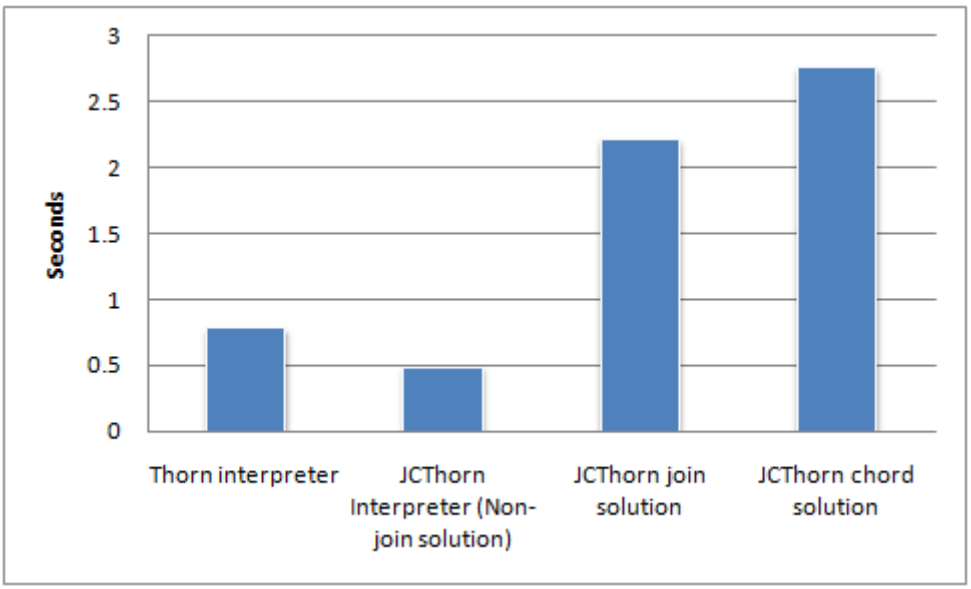


Figure 6.7: Comparison of the different solutions to the Dining Philosophers problem

Single-Lane Bridge problem

Listing 6.8 shows the performance of the different solutions to the Single-Lane Bridge problem. In this case, JCThorn's interpreter also performs better than Thorn's interpreter for the non-join solution (presented in section 6.1.2).

However, the join (section 6.1.2) and chord (appendix A.4) solutions are not only harder to understand, but also considerably slower. The likely cause is the fact that each of the `receive` constructs in the non-join solution have only two cases, while the `receive` in the body of the bridge for the join solution declares 6 joins. The higher number of joins means that more tests with negative results will be performed, which are clearly unnecessary.

6.4.3 Large mailboxes

We believe that the most important factor in the performance of the joins solver is how it behaves under heavy load, particularly in those situations where the mailboxes are large. In this section we recreated a benchmark designed by Hubert Plociniczak, author of the JErlang language [35], with the aim of measuring this factor.

The code for the benchmark is given in appendix C.2. It consists of a server that receives messages from a number of clients. The server declares two joins to retrieve the different messages sent by the clients, which share the synchronous message ``` ``notify'' ```.

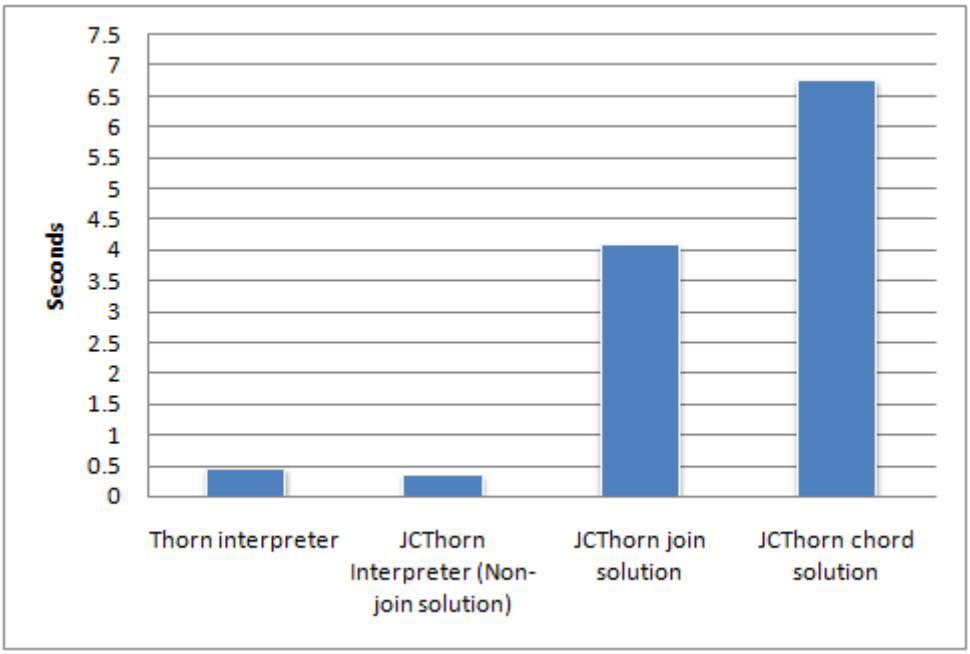


Figure 6.8: Comparison of the different solutions to the Single-Lane Bridge problem

The benchmark measures how many replies the server is able to execute in a limited amount of time.

The clients, after sending each request, sleep for a random amount of time between 0 and 1 seconds in order to simulate the non-deterministic behaviour of any distributed and concurrent application. However, to get consistent results, a pseudo-random number generator is used. For simplicity, we have used the `Random` class declared in the JAVA API with a seed value of 17 (JAVA methods can be called from JCTHORN programs).

For each test, the number of clients sending synchronous messages is kept constant. On the other hand, the number of clients sending asynchronous messages is increased until the mailbox is flooded and the performance of the solver considerably drops. This test is performed for different lengths of time. The reader should be aware that the scale of the horizontal axis is logarithmic.

The test in figure 6.9 increases the number of asynchronous clients from 4 to 1024 and keeps the number of synchronous clients at a constant value of 10. We observe that the results are very positive, and with 256 asynchronous clients (128 in the 120 second run), the server still manages to satisfy large numbers of requests. From then on, it degrades gracefully and with 512 clients the mailbox is clearly flooded. Nonetheless, the server still manages to satisfy a low, but still significant, number of requests.

These results clearly surpass the performance of JERLANG's join solver in the same scenario, presented in [35]. In JERLANG, the peak in the number of requests (at slightly over 900 for the 120 second run) is reached when the number of asynchronous clients is just 8. Then, the number of synchronisations drops to less than 150 when there are just 12 clients. JCTHORN is clearly superior when handling large mailboxes, and with 128

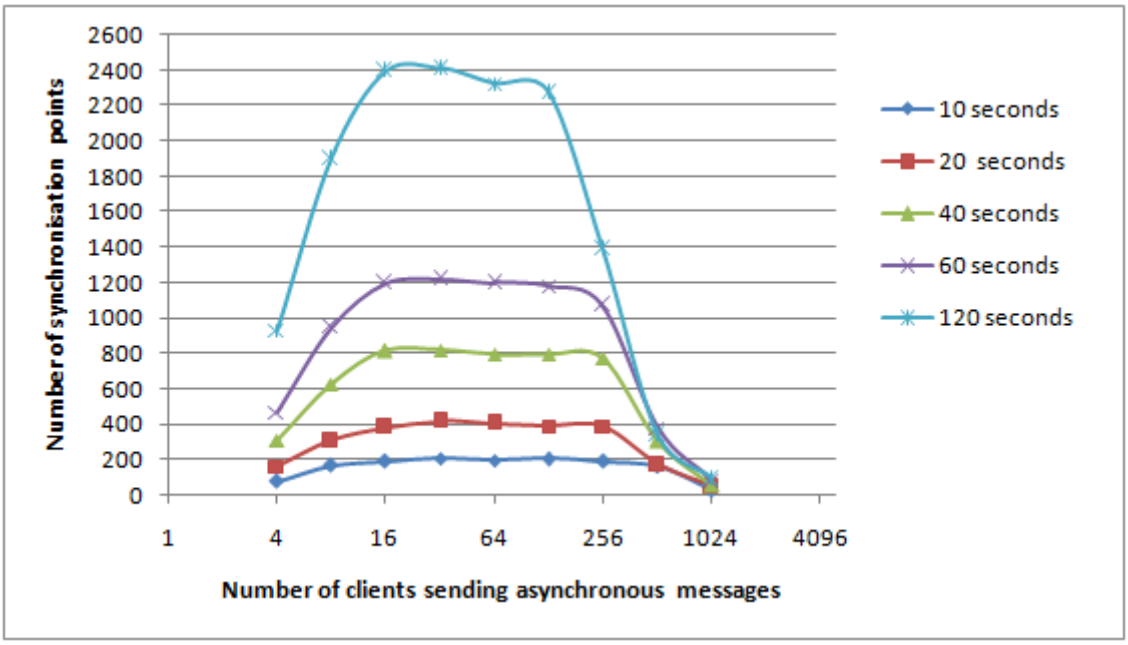


Figure 6.9: Effect of increasing the number of asynchronous clients on the amount of synchronisation points that the server is able to analyse in a give amount of time. The number of synchronous clients is 10

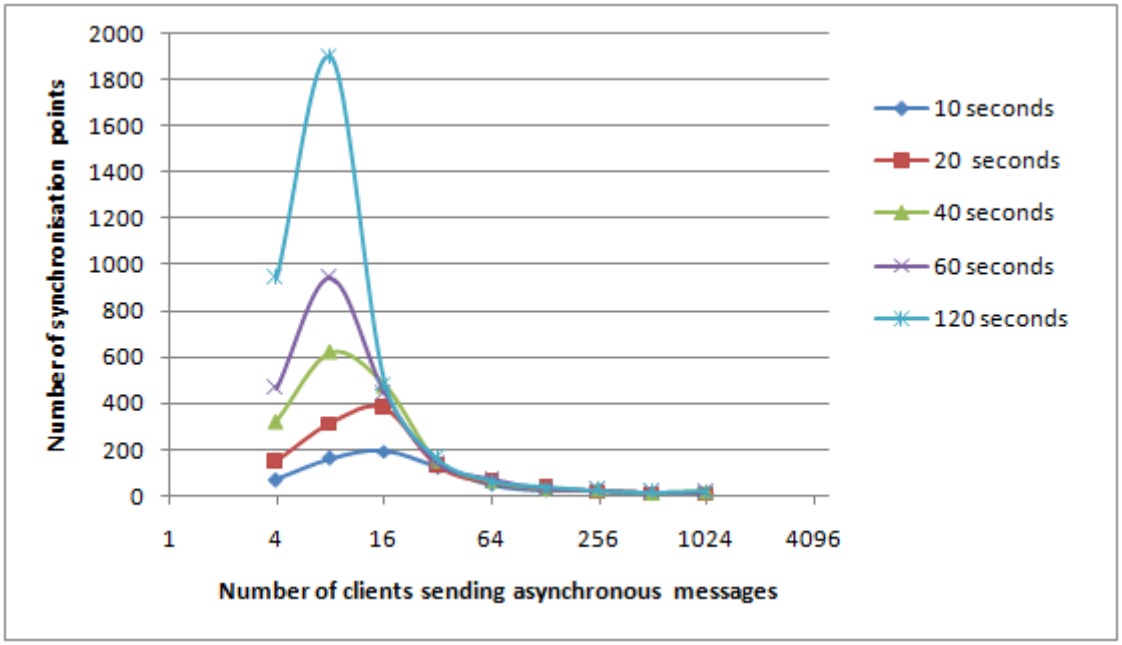


Figure 6.10: Effect of increasing the number of asynchronous clients on the amount of synchronisation points that the server is able to analyse in a give amount of time. The number of synchronous clients is 10. SC optimisation disabled

asynchronous clients, the server is able to answer over 2200 requests. With 256 clients the figure drops to 1400, which means that it is able to handle 20 times more clients that JErlang.

In view of these huge differences in performance between the two languages, the reader might question how faithful our implementation of the benchmark is. However, the code from JErlang's benchmark is publicly available[2] and, moreover, we get very similar results to those in JErlang when we disable the Skip Contextual (SC) optimisation, as shown in figure 6.10.

We observe that when SC is disabled, the peak in the number of synchronisation points happens when the number of asynchronous clients is 8, the same as in JErlang. When the number is 16, the mailbox is already flooded, although the server still manages to answer an important number of requests.

To understand why the SC optimisation has such a dramatic effect, we should consider the joins declared in the body of the server after being reordered according to the rank of their patterns. These joins are shown in listing 6.16. Clearly, when we increase the number of asynchronous clients, the number of messages in the mailbox satisfying all the patterns, with the exception of ``notify'' (ie. the synchronous message), will increase proportionally.

When SC is disabled, we try to find a combination before all the patterns in the join have been satisfied. Hence, as we increase the number of asynchronous clients, we will be testing more combinations that do not involve ``notify'' messages. Obviously, they will not be successful and, essentially, we will be wasting a lot of time doing unnecessary computations.

```
1  receive{
2    {: packetValue:V, id:Id :} and {: buy:$(Id) :}
3    and "notify" from S => {
4      ...
5    }
6    | {: packetValue:_, id:Id :} and {: secure:$(Id) :}
7    and "notify" from S and {: deposit:V1 :} => {
8      ...
9  } }
```

Listing 6.16: Joins declared in the server - after rank reordering

On the other hand, it is also interesting to observe what happens when the Rank Reordering (RR) optimisation is disabled. In JErlang, RR had negative effects on performance in the benchmark, so the designers were force to include a flag in the compiler to allow programmers to enable or disable this optimisation. In contrast, it has no negative effects in JCThorn. This is shown in figure 6.11, which is barely distinguishable from that shown in figure 6.9. Moreover, the benchmark runs faster with RR enable in the situation presented in appendix C.1.

[2] http://www.doc.ic.ac.uk/~susan/jerlang/

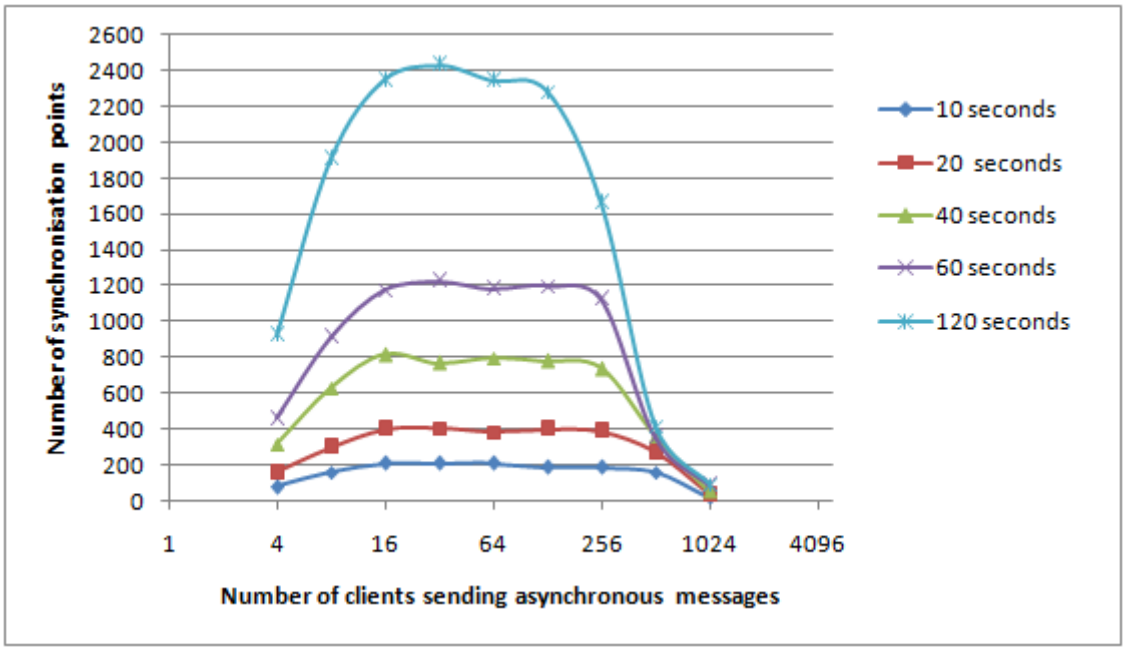


Figure 6.11: Effect of increasing the number of asynchronous clients on the amount of synchronisation points that the server is able to analyse in a give amount of time. The number of synchronous clients is 10. RR optimisation disabled

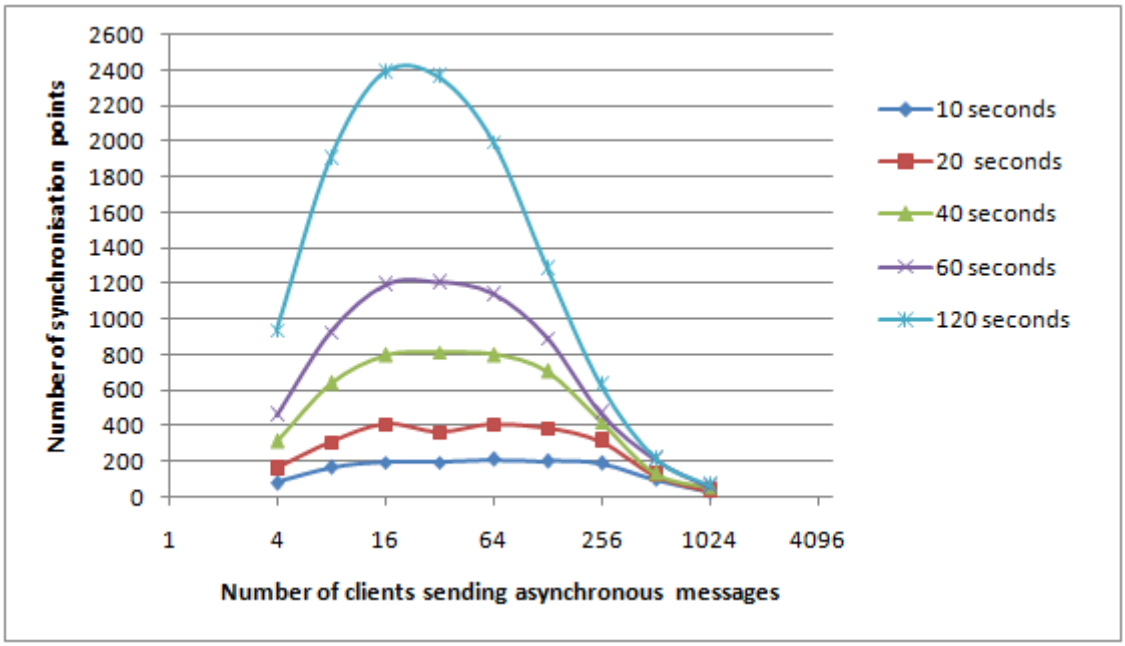


Figure 6.12: Effect of increasing the number of asynchronous clients on the amount of synchronisation points that the server is able to analyse in a give amount of time. The number of synchronous clients is 10. RRCI optimisation disabled

The reason why RR does not negatively affect the performance of JCThorn is that SC ensures that whenever we test a combination, this will be successful. JErlang does not provide the SC optimisation, so when RR is enabled it will take longer to detect that there are no ‘‘notify’’ messages in the mailbox, as this pattern will appear last. Hence, it will test many incomplete combinations. If RR is enabled, ‘‘notify’’ is the first pattern in the join, so it can detect that a message satisfying it has not been received sooner.

Figure 6.12 illustrates how the performance degrades when the Repeating Receive and Context Independence (RRCI) optimisation is disabled. We observe that, in this case, the number of synchronisation points achieved in a 120-second run starts to fall when we reach 32 asynchronous clients instead of 128 (recall figure 6.9). This situation is completely expected, since with RRCI disabled we always start the scan of the mailbox from the beginning, and not from the last checked message. As we increase the number of asynchronous clients, there will be more asynchronous messages than ‘‘notify’’ messages in the mailbox. Hence, as joins fire, the ‘‘notify’’s will be removed from earlier positions, and every time we start a new scan we will have to checked more asynchronous messages until we get to the next ‘‘notify’’.

Finally, figure 6.13 shows the results of running with 40 synchronous clients instead of 10. The peak in the number of synchronisations happens for the 120-second run at the point of 64 asynchronous clients. The number of synchronisations clearly increases prior to that point and drops thereafter.

This contrasts with figure 6.9, in which the number of synchronisations remains fairly stable from the 16-asynchronous-client point to the 128 point. Hence, we deduce that the number of synchronisations was restricted by the number of synchronous clients, which clearly was not enough when we only had 10. We also confirm this finding in appendix C.1, where we reach higher numbers of synchronisation points by allowing synchronous clients to send messages without interleaved waits.

More importantly, figure 6.13 shows that the server is able to serve a maximum number of about 9500 requests in a 120-second run when the number of asynchronous clients is 64. This is more than 27 times the maximum number of points that JErlang is able to solve in the same experiment (ie. 350 synchronisations when there are 12 asynchronous clients).

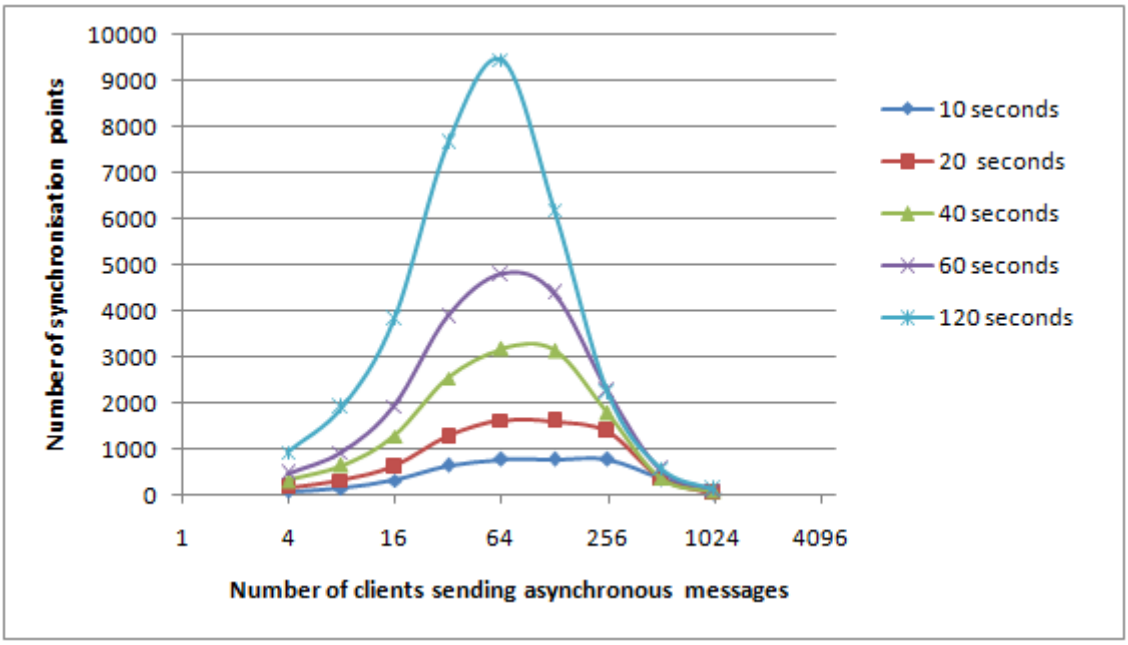


Figure 6.13: Effect of increasing the number of asynchronous clients on the amount of synchronisation points that the server is able to analyse in a give amount of time. The number of synchronous clients is 40

6.5 Scalability

In this section we analyse the effect of joins on the degree of concurrency of the JCThorn language. We base our analysis on the Joe Armstrong challenge [2], that was later adapted by Plociniczak [35] in order to measure the scalability of JErlang. The extended challenge can be summarised as following:

1. Put N processes in a ring
2. Associate each process with 3 consecutive neighbours
3. Send synchronisation messages to the last two neighbours and a main, simple message to the first neighbour
4. Each process performs a join on the main message and the two synchronisation messages coming from the precedent processes
5. Send the main message round the ring M times
6. Increase N to see how long does it takes to process the message
7. Increase M to see how long does it takes to process the message

The code for the Armstrong challenge in JCThorn can be found in appendix D.1. We have also created a second version, presented in appendix D.2, in order to fairly compare the Thorn and JCThorn interpreters. This version replaces step four by:

- Each process executes nested receives, one for each message that receives from its three precedent processes

Figure 6.14 shows the results of running the first and second versions of the Armstrong challenge with the JCThorn and Thorn interpreters respectively. We measure the time taken to run it with rings of different sizes, from 4 to 1024 processes, for different numbers of rounds, 2 and 4. The most notable feature is that both interpreters produce very similar results, so we believe that the introduction of joins did not have a negative effect on the scalability of the language.

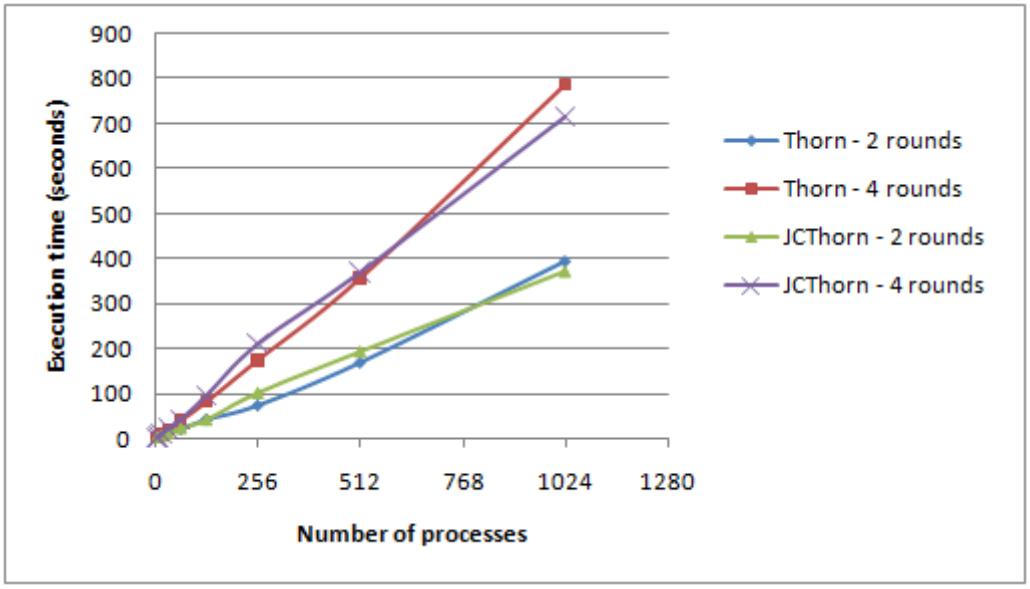


Figure 6.14: Armstrong Challenge results

6.6 Integration

Overall, we believe that JCThorn represents an intuitive extension of the Thorn language. Programmers that are familiar with Thorn can start writing JCThorn programs without requiring much additional training. Join and chord resolution maintains the in-order, first-match evaluation and treats explicit numeric priorities in the same way. Moreover, we believe that the addition of the join operator (ie. `and`) to the syntax fits coherently.

JCThorn also respects the Thornian distinction between bound and free variables in patterns and supports the `before` and `after` clauses next to `serve` statements, albeit with some minor deviations due to the semantic differences between the languages. However, the most radical feature has been the removal of the `catch` clause next to the `serve` statement from the language. The reason was to avoid the unintuitive behaviour that programs exhibit when they make use of both `serve` clauses and interpolation patterns.

6.7 Summary

In this chapter we have provided an in-depth analysis of the extensions that JCThorn proposes. We first analysed the expressiveness of joins by comparing different solutions to the Board Game problem, that we have invented, and to other classic concurrent problems. These provided some insight into the characteristics of the class of problems that are more easily solvable in JCThorn, and helped us to determine that joins can express concepts that are impossible or very hard to articulate in Thorn.

Then, we presented what measures were taken to ensure the correctness of the language. We also described the results obtained by running a set of microbenchmarks with the aim of determining how effective the optimisations included in the language could ever be. Performance was indeed a very important topic in the analysis, and the next section was solely dedicated to it. We first examined the effect that each of the optimisations had in the execution time of applications solving typical concurrent problems. Then, we compared JCThorn's interpreter against Thorn's interpreter and discovered that the former is faster when executing the same application.

We also discovered that the solution to the Santa Claus problem in JCThorn using joins is not only easier to write, but also faster. However, specialised solutions to some other problems will perform better than solutions using joins, as it is the case with the Dining Philosophers problem. In these situations programmers will need to make compromises, and evaluate if the reduction in development time of using joins compensates for the inevitable performance loss.

A benchmark was also created to measure how JCThorn behaves when mailboxes grow large. We discovered that JCThorn outperforms languages with the same level of complexity, such as JErlang. JCThorn is able to efficiently handle 20 times more clients in the same scenario, and the peak in the number of requests that the server is able to serve is up to 27 times larger. The differences in performance are due to the number of optimisations included in JCThorn, which were shown to have very significant effects. In particular, Skip Contextual was incredibly effective when handling large mailboxes, and Repeating Receive also had noticeable effects.

We have also determined that the introduction of joins into JCThorn had no negative effects on the scalability of the language, ie. large numbers of processes are handled as effectively. Finally, we concluded by commenting on how naturally the changes that JCThorn proposes can be integrated into the original Thorn language.

Chapter 7

Conclusions

We have extended the THORN programming language with constructs based on the *Join Calculus*, a process algebra suited for implementation. The result is JCTHORN, a language that we believe will greatly help programmers to solve synchronisation problems. We have seen that THORN has limitations and cannot easily express certain component behaviours, for example, finding the first combination of messages with equal value in the mailbox. In contrast, expressing this action is straightforward in JCTHORN. The root of the problem in THORN lies in the fact that the language relies on message-passing for synchronisation - it does not provide any form of shared memory - and can only retrieve a single message at a time from a component's mailbox. Retrieving multiple messages in one operation is precisely the function of *joins* and *chords*, which are the two main additions included in JCTHORN.

Solutions to classic concurrent problems have also been examined. We determined that *joins* and *chords* make the life of a programmer easier in many situations. For instance, the solution to the `Santa Claus` problem does not only run faster, but is also easier to write. In other situations, programmers may need to make compromises, and evaluate if the gains in development time compensate for the non-prohibited performance cost of using *joins*. Remarkably, the peformance of JCTHORN considerably outperforms that of other languages like JERLANG, which also supports *joins* and has a similar level of complexity. We determined that JCTHORN is particularly efficient even when mailboxes have many messages. Moreover, JCTHORN's interpreter also outperforms THORN's interpreter when executing the same program. This is the result of the multiple optimisations, some novel and others already known, that have been included in JCTHORN.

As a result, we believe that the constructs proposed by JCTHORN could be easily incorporated into the original THORN language. This is also true because we have paid a lot of attention to ensuring that JCTHORN respects THORN's design principles. Accepting *joins* would only require two minor compromises by THORN designers. The first is to remove catch clauses next to serve statements from the language, which we believe should not have been in place originally, because when used in conjunction with interpolation patterns, they result in unexpected program behaviour. However, should the designers want to keep them, we have hinted at an implementation path that could be considered. The second modification would consist of changing the type of arguments of `before` and

`after` clauses to lists. We believe that the expressive power of *joins* clearly compensates for accepting these two backwards incompatible changes, because there is no established THORN user community. However, THORN designers may have a different opinion. If that were the case, many propositions in this report are still relevant. In particular, some optimisations did result in improvements in the execution time of original THORN programs.

7.1 Future Work

We expect the new THORN compiler to be released very soon. When this happens, an interesting project would be to implement JCTHORN in that compiler. Proving that this can be done efficiently will support the potential adoption of *joins* by the original THORN language. Other future work could include investigating further how the performance of the *join* solver could be improved. The Rank Reordering optimisation could be enhanced by performing a more complex static analysis of the *joins*. Interpolation patterns matching the value of a variable, and not of an expression, inside the *join* pattern with a higher rank could be replaced by a wildcard pattern defining that variable. This modification will avoid having to prematurely insert patterns upon which others are dependent, and will allow higher rank patterns to be positioned closer to the front of the *join*.

The mailbox is currently implemented by an array-based data structure - the JAVA thread-safe class `CopyOnWriteArrayList` - in which all mutative operations are implemented by making a fresh copy of the underlying array. However, this might not be the most efficient data structure to support the operations performed during *join* resolution, namely sequential access, random access and appending. It would be valuable to examine how other data structures might affect performance. Regarding the contextual matching phase, we do not store partial results. Although we ensure that we only test combinations involving new messages, parts of these combinations may only consist of messages that have already been checked. Thus, we could avoid duplicating work done by storing partial results. However, maintaining this state has associated overheads, so it would be interesting to measure how it would impact performance.

Finally, we could evaluate further the expressiveness of the JCTHORN language. Examining solutions to some other concurrent problems will help us understand better what the real value of *joins* is.

Appendix A

Alternative solutions to typical concurrent problems

A.1 Dining Philosophers - Thorn's solution

This is THORN's solution to the Dining Philosophers problem. It uses a waiter to which the philosophers ask for permission before using the forks. The waiter records the state of each of the philosophers and forks in two tables, `phil` and `forks` respectively.

```
1  spawn DiningPhil {
     body{
3      component Waiter(guests){

5        var forks := table(id){
           var taken;
7        };
         var phil := table(id){
9            var waiting;
             var phil;
11           var name;
         };
13       fun serveFork(Id, p, philName){
             forks(Id) := {:taken:true:};
15           forks((Id+1) mod guests) := {:taken:true:};
             p <<< {: left:Id, right:((Id+1) mod guests) :};
17       }

19       body{

21         /* initialize state of forks and philosophers */
           for(i <- 0 .. (philNum - 1)){
23           forks(i) := {:taken:false:};
           }
25         for(i <- 0 .. (philNum - 1)){
             phil(i) := {:waiting:false,phil:null,name:null:};
27         }
           var waiting := [];
```

```
29
        while(true){
31        receive{

33          {: take:Id, name:N :} from P  => {
              if(forks(Id) ~ {:taken:false:}
35                && forks((Id+1) mod guests) ~ {:taken:false:}){
                serveFork(Id, P, N);
37            } else {
                phil(Id) := {:waiting:true,phil:P,name:N:};
39            }
            }
41          | {: drop:Id, name:N :} from P  => {
              /* drop forks */
43            forks(Id) := {:taken:false:};
              forks((Id+1) mod guests) := {:taken:false:};
45            left = (Id + guests - 1) mod guests;
              right = (Id+1) mod guests;
47
              /* allow left to eat if waiting and other fork available */
49            if(phil(left) ~ {:waiting:true,phil:pLeft,name:nameLeft:} &&
                  forks(left) ~ {:taken:false:} ){
51              phil(left) := {:waiting:false,phil:null,name:null:};
                serveFork(left, pLeft, nameLeft);
53            }

55            /* allow right to eat if waiting and other fork available */
              if(phil(right) ~ {:waiting:true,phil:pRight,name:nameRight:} &&
57                forks((right+1) mod guests) ~ {:taken:false:} ){
                phil(right) := {:waiting:false,phil:null,name:null:};
59              serveFork(right, pRight, nameRight);
    } } } } } }
61
    component Phil(n, id , waiter){
63    body{
        while(true){
65        /* thinking */
          waiter <<< {: take:id, name:n:};
67
          receive{
69          {: left:L, right:R :} => {
              /* eating */
71            waiter <<< {: drop:id, name:n:};
    } } } } }
73
    w = spawn Waiter(5);
75  phils = [
      spawn Phil("Kant", 0, w),
77    spawn Phil("Hume", 1, w),
      spawn Phil("Marx", 2, w),
79    spawn Phil("Plato",3, w),
      spawn Phil("Nietzsche",4, w) ];
81 } }
```

Listing A.1: Solution to the Dining Philosophers problem in Thorn

A.2 Dining Philosophers - Chord solution

The solution to the Dining Philosophers problem that uses chords is very similar to that using joins. The fragment below shows how the waiter is written in the solution using chords.

```
component Waiter(guests){

  sync phil(id, name) and
  async fork($(id)) and
  async fork($((id+1) mod guests)) {
    return {: left:id, right:((id+1) mod guests) :};
  }

  body{
    while(true){
      serve;
} } }
```

Listing A.2: Solution to the Dining Philosophers problem using chords

A.3 Santa Claus - Thorn's solution

This is Thorn's solution to the Santa Claus problem. Santa Claus gets help from two secretaries, which are responsible of marshalling the groups of elves and reindeer respectively.

```
spawn SantaClaus {
  body{

    component worker(secretary){
      body{
        while(true){
          secretary <<< "ready";
          receive{
            "ok" from santa => {
              /* work */
              santa <<< "done";
            }
    } } } }

    component secretary(santa, species, total){
      var count := 1;
      var workers := [];
      body{
        while(true){
          receive {
            "ready" from W =>{
              workers := [W, workers...];
              if(count == total){
                santa <<< {: type:species, ids:workers :};
```

```
                count := 1;
26              workers := [];
              } else {
28              count := count + 1;
    } } } } } }
30
    component santa(){
32    body{
        var group;
34      while(true){
          /* sleep */
36        receive{

38          {: type: reindeer, ids:Ids :} prio 10 => {
              group := Ids;
40          }

42          | {: type: elfes, ids:Ids :} =>{
              group := Ids;
44          }
          }
46        /* wake up */

48        for (e <- group){
            e <<< "ok";
50        }

52        /* help workers */

54        for (e <- group){
            receive{ "done" => {}};
56  } } } }

58  s = spawn santa();
    robin = spawn secretary(s, "reindeer", 9);
60  edna = spawn secretary(s, "elfes", 3);
    rs = %[spawn worker(robin) | for j <- 1 .. 9];
62  es = %[spawn worker(edna) | for j <- 1 .. 10];
} }
```

Listing A.3: Solution to the Santa Claus in Thorn

A.4 Single-Lane Bridge - Chord solution

A bridge that solves the Singe-Lane Bridge problem can be programmed using chords as shown below.

```
module BRIDGE{
  component Bridge(){

    sync enterSouth() and
    async south(n){
      m = n + 1;
      self <-- south(m);
      return;
    }

    async exitSouth() and
    async south(n){
      m = n - 1;
      if(m == 0){
        self <-- empty();
      } else {
        self <-- south(m);
    } }

    sync enterNorth() and
    async north(n){
      m = n + 1;
      self <-- north(m);
      return;
    }

    async exitNorth() and
    async north(n){
      m = n - 1;
      if(m == 0){
        self <-- empty();
      } else {
        self <-- north(m);
    } }

    sync enterNorth() and
    async empty(){
      self <-- north(1);
      return;
    }

    sync enterSouth() and
    async empty(){
      self <-- south(1);
      return;
    }

    body{
      self <-- empty();
      while(true){
        serve;
} } } }
```

Listing A.4: Solution to the Single-Lane Bridge problem in Thorn

Appendix B

Microbenchmarks

```
spawn microCCMO {
  body{
    import FromJava.*;
    ITER = 60;
    outer = thisComp();
    start = currentTime();

    component client(s){
      body{
        /* Note, mailbox will overflow, one extra 4 every round */
        for(i <- 1 .. ITER){
          s <<< 1; s <<< 1;
          s <<< 4; s <<< 4;
          s <<< 4; s <<< 2;
          s <<< 2; s <<< 1;
    } } }

    component server(){
      body{
        for(i <- 1 .. ITER){
          receive{
            1 and 1 and 2 and 4 and 4 and 1 => { /* do nothing */ }
        } }
        outer <<< "done";
    } }

    s = spawn server();
    c = spawn client(s);
    recv{"done"  => {}};
    end = currentTime();
    execTime = (end - start) / 1000;
    println("$execTime");
} }
```

Listing B.1: Microbenchmark to test CCMO optimisation

```
spawn microCC {
  body{
    import FromJava.*;
    ITER = 1000;
    outer = thisComp();
    start = currentTime();

    component client(s){
      body{
        for(i <- 1 .. ITER){
          s <<< {: book: "Brave New World" :};
          s <<< [29,03,1986];
          s <<< 7;
          s <<< "La Dolce Vita";
          s <<< {: author: "Albert Camus":};
          s <<< ["key", 432];
          s <<< {: name: "Peter J.", address: "London" :};
          s <<< [0,1,2,3];
          s <<< "Imperial College";
          s <<< {: Morroco:"Mirleft" :};
    } } }

    component server(){
      body{
        for(i <- 1 .. ITER){
          receive{
            {: book: "Brave New World" :} and [x,y,z]
            and 7 and "La Dolce Vita" and {: author: "Albert Camus":}
            and ["key", 432] and {: name: "Peter J.", address: "London" :}
            and [0,1,2,3] and "Imperial College"
            and {: Morroco:"Mirleft" :} => {
              /* do nothing */
        } } }
        outer <<< "done";
    } }

    s = spawn server();
    c = spawn client(s);
    recv{"done"  => {}};
    end = currentTime();
    execTime = (end - start) / 1000;
    println("$execTime");
} }
```

Listing B.2: Microbenchmark to test CC optimisation

```
spawn microUSP {
  body{
    import FromJava.*;
    ITER = 1000;
    outer = thisComp();
    start = currentTime();

    component client(s){
      body{
        for(i <- 1 .. ITER){
          s <<< {: data: "one" :};
          s <<< "one";
          s <<< "two";
          s <<< [1];
          s <<< {: data: "two" :};
          s <<< {: data: "three" :};
          s <<< {: data: "four" :};
          s <<< {: data: "five" :};
          s <<< [2,3];
          s <<< [4,5];
          s <<< [6,7];
          s <<< [8,9];
    } } }

    component server(){
      body{
        for(i <- 1 .. ITER){
          receive{
            x and [a,b...] and [c,d...] and [e,f...] and [g,h...]
            and {: data: n :} and {: data: j :} and {: data: k :}
            and {: data: l :} and [m] and {: data:"one" :} and "one" => {
              /* do nothing */
        } } }
        outer <<< "done";
    } }

    s = spawn server();
    c = spawn client(s);
    recv{"done"  => {}};
    end = currentTime();
    execTime = (end - start) / 1000;
    println("$execTime");
} }
```

Listing B.3: Microbenchmark to test USP optimisation

In this section we have listed the code from the microbenchmarks to test the following optimisations: CCMO, CC and USP. To test the RRCI optimisation the Single-Lane Bridge problem was used. The microbenchmark for the SC optimisation was the Santa Claus problem.

Appendix C

Benchmark

C.1 More results

Figure C.1 shows the result of running the benchmark with 40 synchronous clients. The random delayed between consecutive sends in the synchronous clients has been removed. This fact explains the large number of requests that the server is able to handle. The peak is 30000 when there are 128 asynchronous clients in the 120 second run.

Figure C.1 shows the same experiment, but with the Rank Reordering optimisation disabled. We observe that the performance is slightly worse. For the 10, 20 and 40 second runs, the number of requests that the server is able to handle start to drop when the number of asynchronous clients reaches 128 instead of 256. Moreover, the maximum number of synchronisation for this runs is also considerably lower.

Figure C.3 shows the effect of assigning higher numeric priority to reindeer. In the similar experiment presented in section 6.4.3 of chapter 6, the reindeer only had positional priority. The explicit numeric priority does not have a dramatic effect on performance, although the mailbox floods slightly earlier.

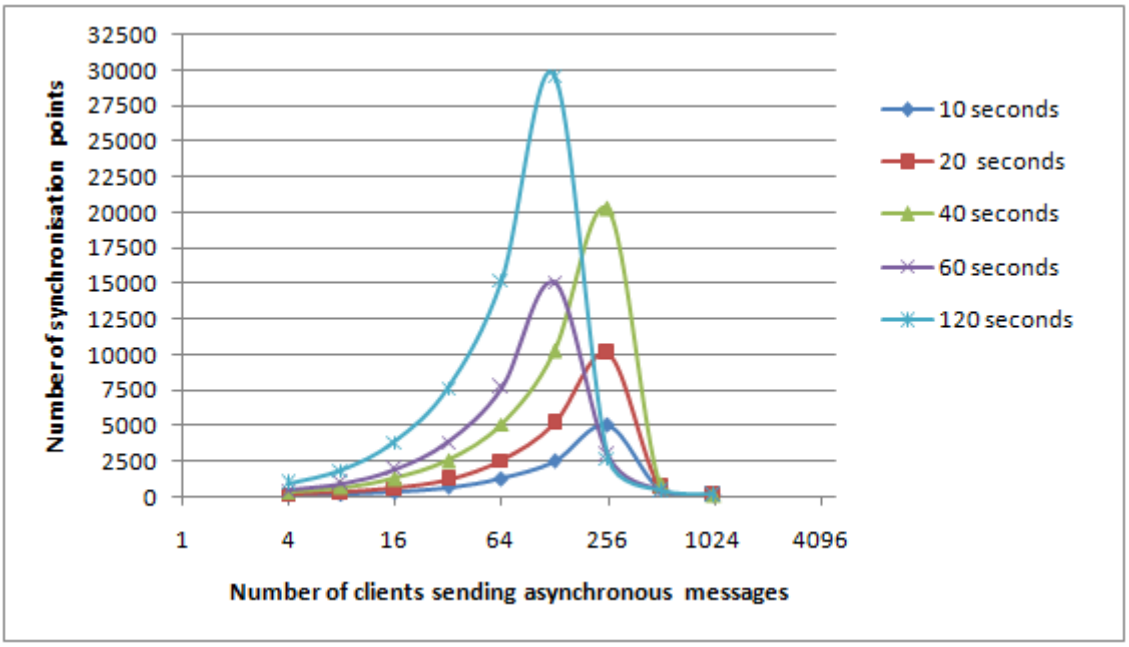


Figure C.1: Effect of increasing the number of asynchronous clients on the amount of synchronisation points that the server is able to analyse in a give amount of time. The number of synchronous clients is 40. Synchronous clients are not delayed after sending a message in this experiment

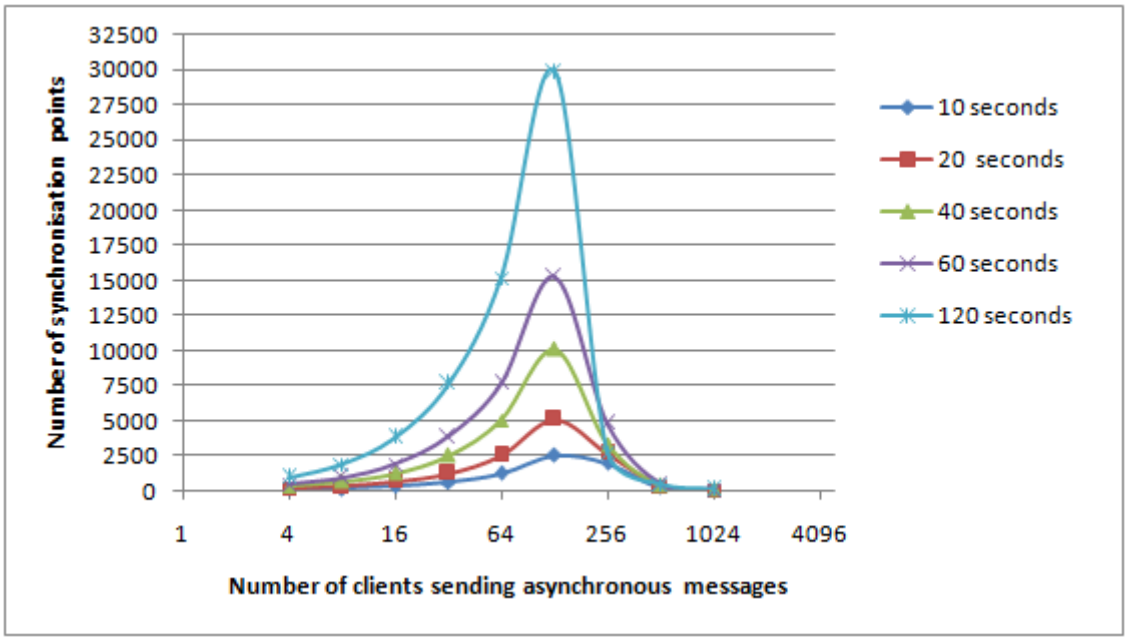


Figure C.2: Effect of increasing the number of asynchronous clients on the amount of synchronisation points that the server is able to analyse in a give amount of time. The number of synchronous clients is 40. Synchronous clients are not delayed after sending a message in this experiment. Rank Reordering optimisation disabled

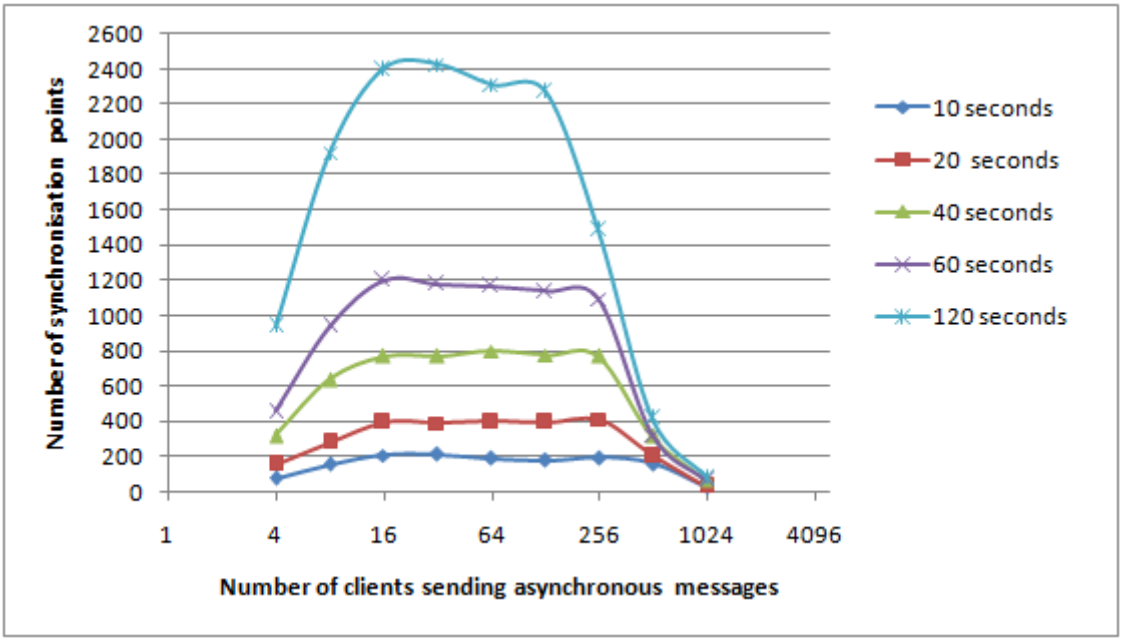


Figure C.3: Effect of increasing the number of asynchronous clients on the amount of synchronisation points that the server is able to analyse in a give amount of time. The number of synchronous clients is 10. Explicit numeric priority to reindeer

C.2 Code

```
spawn Benchmark {
  body{
    INTERVAL = 10000;
    ASYNC_CLIENTS = 64;
    SYNC_CLIENTS = 10;
    outer = thisComp();

    component syncClient(s){
      body{
        import FromJava.*;
        start = currentTime();
        end = start + INTERVAL;
        var curr := 0;
        while(curr < end){
          curr := currentTime();
          s <<< "notify";
          receive{
            {: ok:"buy", id:Id :} =>{}
            | {: ok:"sell", id:Id :} => {}
            timeout(10000) {}
          }
          sleepRandom();
    } } }

    component packetGen(s){
      body{
        import FromJava.*;
        start = currentTime();
        end = start + INTERVAL;
        var curr := 0;
        while(curr < end){
          curr := currentTime();
          s <<< {: packetValue: 45, id:3 :};
          sleepRandom();
    } } }

    component buyer(s){
      body{
        import FromJava.*;
        start = currentTime();
        end = start + INTERVAL;
        var curr := 0;
        while(curr < end){
          curr := currentTime();
          s <<< {: buy:3 :};
          sleepRandom();
    } } }

    component deposit(s){
      body{
        import FromJava.*;
        start = currentTime();
```

```
53          end = start + INTERVAL;
            var curr := 0;
55          while(curr < end){
              curr := currentTime();
57            s <<< {: deposit: 56 :};
              sleepRandom();
59      } } }

61      component secure(s){
          body{
63          import FromJava.*;
            start = currentTime();
65          end = start + INTERVAL;
            var curr := 0;
67          while(curr < end){
              curr := currentTime();
69            s <<< {: secure: 3 :};
              sleepRandom();
71      } } }

73      component server(){
          body{
75          import FromJava.*;
            start = currentTime();
77          end = start + INTERVAL;
            var curr := 0;
79          var counter := 0;
            while(curr < end){
81            curr := currentTime();
              receive{
83              "notify" from S and {: packetValue:V, id:Id :}
                and {: buy:$(Id) :} => {
85                S <<< {: ok:"buy", id:Id :};
                  counter := counter + 1;
87              }
                | "notify" from S and {: deposit:V1 :}
89              and {: packetValue:_, id:Id :} and {: secure:$(Id) :} => {
                  S <<< {: ok:"sell", id:Id :};
91                counter := counter + 1;
                }
93              timeout(2000) {}
              }
95          }
            println("$counter"); outer <<< "done";
97      } }

99      s = spawn server();
        %[spawn syncClient(s) | for j <- 1 .. SYNC_CLIENTS];
101     %[spawn packetGen(s) | for j <- 1 .. ASYNC_CLIENTS];
        %[spawn buyer(s) | for j <- 1 .. (ASYNC_CLIENTS/2)];
103     %[spawn deposit(s) | for j <- 1 .. (ASYNC_CLIENTS/2)];
        %[spawn secure(s) | for j <- 1 .. (ASYNC_CLIENTS/2)];
105 } }
```

Listing C.1: Benchmark that generates large mailboxes

Appendix D

Armstrong Challege

D.1 With joins

```
spawn Amstrong {
  body{
    ROUNDS = 4;
    PROCESSES = 1000;
    OUTER = thisComp();

    component startProcess(id){
      body{
        import FromJava.*;
        start = currentTime();
        var next;
        var next2;
        var next3;
        OUTER <<< "ready";
        receive{ "go" => {}};
        receive{ N => { next := N; }}
        receive{ N2 => { next2 := N2;}}
        receive{ N3 => { next3 := N3;}}

        next <<< {: type: 1 :};
        next2 <<< {: type: 2 :};
        next3 <<< {: type: 3 :};
        yo = thisComp();

        for(i <- 1 .. ROUNDS){
          receive{
            {: type: 1 :} and {: type: 2 :} and {: type: 3 :} => {
              if  (i < ROUNDS){
                next <<< {: type: 1 :};
                next2 <<< {: type: 2 :};
                next3 <<< {: type: 3 :};
        } } } }

        end = currentTime();
          execTime = (end - start) / 1000;
        println("$execTime");
    } }

    component endProcess(id, next){
      body{
        var next2;
        var next3;
        OUTER <<< "ready";
        receive{ "go" => {}};

        receive{ N2 => { next2 := N2;}}
        receive{ N3 => { next3 := N3;}};

        next2 <<< {: type: 2 :};
        next3 <<< {: type: 3 :};
        yo = thisComp();

        for(i <- 1 .. ROUNDS){
```

```
            receive{
              {: type: 1 :} and {: type: 2 :} and {: type: 3 :} => {
                if  (i < ROUNDS){
                  next <<< {: type: 1 :};
                  next2 <<< {: type: 2 :};
                  next3 <<< {: type: 3 :};
                } else {
                  next <<< {: type: 1 :};
    } } } } } }

    component beforeEndProcess(id, next, next2){
      body{
        var next3;
        OUTER <<< "ready";
        receive{ "go" => {}};
        receive{ N3 => { next3 := N3;}};
        next3 <<< {: type: 3 :};
        yo = thisComp();

        for(i <- 1 .. ROUNDS){
          receive{
            {: type: 1 :} and {: type: 2 :} and {: type: 3 :} => {
              if  (i < ROUNDS){
                next <<< {: type: 1 :};
                next2 <<< {: type: 2 :};
                next3 <<< {: type: 3 :};
              } else {
                next <<< {: type: 1 :};
                next2 <<< {: type: 2 :};
    } } } } } }

    component process(id, next, next2, next3){
      body{
        OUTER <<< "ready";
        yo = thisComp();

        for(i <- 1 .. ROUNDS){
          receive{
            {: type: 1 :} and {: type: 2 :} and {: type: 3 :} => {
                println("iter: $i id: $id");
                next <<< {: type: 1 :};
                next2 <<< {: type: 2 :};
                next3 <<< {: type: 3 :};
    } } } } }

    startP = spawn startProcess(PROCESSES);
    endP = spawn endProcess((PROCESSES - 1), startP);
    beforeEndP = spawn beforeEndProcess((PROCESSES - 2), endP, startP);

    var secondP := beforeEndP;
    var thirdP := endP;
    var fourthP := startP;
    for (j <- 1 .. (PROCESSES - 3)){
      temp = spawn process((PROCESSES - (2 + j)), secondP, thirdP, fourthP);
      fourthP := thirdP;
      thirdP := secondP;
      secondP := temp;
    }

    for(i <- 1 .. PROCESSES){
      receive{ "ready" => {}};
    }

    beforeEndP <<< "go";
    endP <<< "go";
    startP <<< "go";

    beforeEndP <<< secondP;

    endP <<< secondP;
    endP <<< thirdP;

    startP <<< secondP;
    startP <<< thirdP;
    startP <<< fourthP;
  }body
}Amstrong;
```

Listing D.1: Armstrong Challenge with joins

D.2 Without joins

This version of the Armstrong Challenge does not use joins. Instead, three nested receive statements are used in every node. We only give the fragment of code belonging to one of the nodes, since the rest of the code is very similar to the version using joins.

```
1   component process(id, next, next2, next3){
      body{
3       OUTER <<< "ready";
        yo = thisComp();
5
        for(i <- 1 .. ROUNDS){
7         receive{
            {: type: 1 :} => {
9             receive{
                {: type: 2 :} => {
11                receive{
                    {: type: 3 :} => {
13                    next <<< {: type: 1 :};
                      next2 <<< {: type: 2 :};
15                    next3 <<< {: type: 3 :};
} } } } } } } } }
```

Listing D.2: Armstrong Challenge without joins

Appendix E

Log of changes

We list all the changes made to the grammar file and the interpreter. The `+` prefix means lines of code added, and the `!` prefix lines of code modified. `M` stands for modified and `A` for added. In the interpreter section, the total number of lines changed in each package are listed.

Grammar

`grammar/grammar-fisher.jj +112 !118`

Interpreter

- `src.fisher.desugar +155 !487`
  - `M: AbstractDesugarer.java`
  - `M: DistDesugarer.java`
- `src.fisher.eval.interfaces +1184 !20`
  - `A: NonBindingPatternMatcher.java`
  - `M: Computer.java`
  - `M: Evaller.java`
  - `M: EvalUtil.java`
  - `M: Frame.java`
  - `A: JoinMatcher.java`
  - `Rename:  Matchiste.java to BindingPatternMatcher.java`
- `src.fisher.ingest +7 !19`
  - `M: Ingester.java`

- src.fisher.run +9 !5
  - M: Thorn.java
- src.fisher.runtime +5 !8
  - M: ClassDynamicTh.java
  - M: ObjectTh.java
- src.fisher.runtime.dist +162 !256
  - A: ComponentJoinState.java
  - M: ComponentThread.java
  - M: Letter.java
  - A: LetterLinkedList.java
  - M: LetterWithSerializedContents.java
  - A: MailboxEntry.java
  - A: RecvMatchStateHandler.java
- src.fisher.runtime.lib.http +3
  - M: MscComp2HTTPSocket.java
- src.fisher.runtime.lib.socketeer +3
  - M: MsgComp2StringSocket.java
- src.fisher.statics +443 !2
  - M: ExtractSealsFromModuleMember.java
  - M: Sealant.java
  - M: SealKind.java
  - M: SealMaker.java
  - A: JoinStaticAnalysis.java
- src.fisher.statics.purity +18
  - M: StaticPurityChecker.java
- src.fisher.syn +1220 !286
  - A: ChannelDecl.java
  - A: ChordBody.java
  - A: ChordDecl.java
  - M: HighLevelCommunication.java
  - A: JoinDecl.java

  - A: JoinPat.java
  - A: JoinSignature.java
  - M: Pat.java
  - M: Recv.java
  - M: Spawn.java
- src.fisher.syn.visitor +40
  - M: PureticVisitor.java
  - M: PureticWalker.java
  - M: VanillaVisitor.java
  - M: VanillaWalker.java
  - M: Visitor.java
- src.fisher.test +331 !47
  - M: AllTests.java
  - A: DistTestJoins.java
- src.fisher.util +97
  - A: Ignacio.java
- TOTAL: +3677 !1130 , both: 4807

Testcase

Added folder `/testcase/joins`, that contains 26 new testcases.

Applications

Added folder `apps/` with all the applications covered in this report. For example, solutions to the Dining Philosopher problem, the benchmark, etc.

Bibliography

[1] Process calculus. `http://en.wikipedia.org/wiki/Process_calculus`. [Accessed on December 2009].

[2] J. Armstrong. Concurrency oriented programming in Erlang. *Invited talk, FFG*, 2003.

[3] JCM Baeten. A brief history of process algebra. *Theoretical Computer Science*, 335(2-3):131–146, 2005.

[4] J.P. Banatre and D. Le Metayer. The GAMMA model and its discipline of programming. *SCI. COMP. PROGRAM.*, 15(1):55–77, 1990.

[5] N. Bansal and M. Sviridenko. The santa claus problem. In *Proceedings of the thirty-eighth annual ACM symposium on Theory of computing*, page 40. ACM, 2006.

[6] M. Ben. Principles of concurrent and distributed programming. 2006.

[7] N. Benton. Jingle bells: Solving the santa claus problem in polyphonic c#, 2003.

[8] N. Benton, L. Cardelli, and C. Fournet. Modern concurrency abstractions for C#. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 26(5):769–804, 2004.

[9] G. Berry and G.Boudol. The chemical abstract machine. *Theoretical computer science*, 96(1):217–248, 1992.

[10] B. Bloom, J. Field, N. Nystrom, J. Östlund, G. Richards, R. Strniša, J. Vitek, and T. Wrigstad. Thorn: robust concurrent scripting on the JVM. In *Proceeding of the 24th ACM SIGPLAN conference companion on Object oriented programming systems languages and applications*, pages 789–790. ACM, 2009.

[11] S. Conchon and F. Le Fessant. Jocaml: Mobile agents for objective-caml. *Migration*, 1000(1):n2.

[12] K.D. Cooper and L. Torczon. *Engineering a compiler*. Elsevier, 2004.

[13] C. Delporte-Gallet, H. Fauconnier, and R. Guerraoui. Shared memory vs message passing. *Technical Report2003*, 77.

[14] S. Drossopoulou, A. Petrounias, A. Buckley, and S. Eisenbach. School: A small chorded object-oriented language. *Electronic Notes in Theoretical Computer Science*, 135(3):37–47, 2006.

[15] C. Fournet and G. Gonthier. The reflexive CHAM and the join-calculus. In *Proceedings of the 23rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 372–385. ACM New York, NY, USA, 1996.

[16] C. Fournet, G. Gonthier, J.J. Lévy, L. Maranget, and D. Rémy. A calculus of mobile agents. *Lecture Notes in Computer Science*, 1119:406–421, 1996.

[17] M. Hennessy and R. Milner. *On observing nondeterminism and concurrency*. Springer.

[18] CAR Hoare. Communicating sequential processes. *Communications of the ACM*, 21(8):677, 1978.

[19] C. Holzbaur, M. de la Banda, D. Jeffery, and P. Stuckey. Optimizing compilation of constraint handling rules. *Logic Programming*, pages 74–89.

[20] D.C. Hyde. Introduction to the programming language Occam. *Department of Computer Science Bucknell University, Lewisburg*, 1995.

[21] G.S. Itzstein and M. Jasiunas. On implementing high level concurrency in Java. *Lecture Notes in Computer Science*, 2823:151–165, 2003.

[22] G.S. Itzstein and D. Kearney. Applications of join Java. *Australian Computer Science Communications*, 24(3):46, 2002.

[23] S. Krishnaprasad. Concurrent/Distributed programming illustrated using the dining philosophers problem. *Journal of Computing Sciences in Colleges*, 18(4):110, 2003.

[24] F. Le Fessant and L. Maranget. Compiling join-patterns. *Electronic Notes in Theoretical Computer Science*, 16(3):205–224, 1998.

[25] Q. Ma and L. Maranget. Compiling pattern matching in join-patterns. *Lecture notes in computer science*, pages 417–431, 2004.

[26] Paul Mackay. Why has the actor model not succeeded? Accessed on December 2009, from: `http://www.doc.ic.ac.uk/~nd/surprise_97/journal/vol2/pjm2/`. [Accessed on December 2009].

[27] J. Magee and J. Kramer. *State models and java programs*. Wiley, 1999.

[28] L. Maranger and L.Mandel. *JoCaml Documentation and Manual (Release 3.11)*. INRA, INRA, INRA, 2008.

[29] S. Matsuoka and A. Yonezawa. Analysis of inheritance anomaly in object-oriented concurrent programming languages. 1993.

[30] R. Milner. *A calculus of communicating systems*. Springer-Verlag New York, Inc. Secaucus, NJ, USA, 1982.

[31] R. Milner. *Communicating and mobile systems: the pi-calculus*. Cambridge Univ Pr, 1999.

[32] T.W.F.Z. Nardelli and S.L.J.O.J. Vitek. Integrating Typed and Untyped Code in a Scripting Language.

[33] D.A. Park. Concurrent programming in a nutshell. *Journal of Computing Sciences in Colleges*, 23(4):51–57, 2008.

[34] B.C. Pierce and D.N. Turner. Pict: A programming language based on the pi-calculus. *Proof, language and interaction: Essays in honour of Robin Milner*, 1997.

[35] H. Plociniczak. JErlang: Erlang with Joins.

[36] A.S. Tanenbaum. *Modern operating systems*. Prentice Hall Englewood Cliffs, NJ, 2001.

[37] J.A. Trono. A new exercise in concurrency. *ACM SIGCSE Bulletin*, 26(3):8–10, 1994.