Imperial College of London
Department of Computing

JErlang: Erlang with Joins

Final Report

Hubert Plociniczak

Supervisor
Professor Susan Eisenbach

Second marker
Professor Sophia Drossopoulou

Abstract

For the past few years the computing world has been affected by a massive increase in the popularity of multi-core machines. In order to fully utilise this opportunity, programmers have to switch their mindset to concurrent thinking. The human nature allows for quick adaptation to new environment, yet ironically, the tools that we use to program the concurrent programs are still in the late 80's where sequential programming was rampant.

Writing massive concurrent applications is an inherently error prone process due to the way humans think. Nevertheless popular languages like Java or C# still use locking as a basic concurrent abstraction. Erlang, a precursor in the world of concurrent languages, offers message passing instead of shared memory, as a way of avoiding this issue. Yet this idea itself is also very primitive and the whole burden of managing large amounts of processes that use the possibilities of multi-core machines, is again left on the shoulders of programmers.

This report investigates the development of JErlang, an extension of Erlang, that introduces simple yet powerful joins constructs. The idea is adapted from the Join-calculus, a process calculus that has recently been developed by Fournet and Gonthier. The aim of JErlang is to allow the programmer for easier synchronisation between different processes in an intuitive way. This will enable Erlang programmers to write concurrent applications faster, reliably and increase the overall clarity of the programs.

Acknowledgements

This project would not reach this stage without the help of many people, a few of which I will name below. I would like to thank Professor Susan Eisenbach, for her continuous support and encouragement not only during this project but during the whole duration of my study.

Secondly, I would like to thank Natalia, for her patience during the nervous times and smile that helped me to survive each day.

Many thanks also go to Professor Sophia Drossopoulou who helped me in thinking formally about languages and her persistance to make things succint. Dr Matthias Radestock who taught me how to write to proper programs and provided invaluable knowledge about Erlang. Thanks to Anton Stefanek for inspiring debates about programming languages, computing in general and hints for LaTeX.

Finally, I would like to thank my parents who always approved my, sometimes irresponsible and strange, decisions and allowed me to pursue my dreams. Special thanks go to my brother, Łukasz, his wife, Gosia and little Emilka for they never stopped believing in my abilities.

To Natalia

Contents

Chapter 1

Introduction

Writing concurrent applications in today's world has become not only the domain of a few programmers who produce superb, massively used systems. It has become the very nature of any kind of software development since it directly corresponds to the way our world is functioning on its own. The widespread popularity of efficient multi-threaded and multi-core architectures has further increased the interest in the subject of concurrency. Desktops with dual- or quad- cores at homes are no longer a domain of geeks or professionals. Laptops offering CPU-power once reserved for massive mainframes are easily accessible in every computer shop. This constant decrease in costs and increase in popularity and efficiency is a consequence of the continuous development of technology over the past 20 years.

Yet, ironically, the way most of the software is currently developed doesn't reflect this significant change. The industry still uses the good old synchronization patterns that are prone to errors, popular programming languages like *Java*, C# or *C++* which don't present any efficient way to correctly handle concurrency problems. Modern languages do try to catch up by adding new libraries, extending the languages, but their main fault is their history - they were designed with different ideas in mind, in a world of different technology. Their basic level of communication remains shared memory. At that time this did seem like a perfectly reasonable idea - speed of memory wasn't so dramatically different from the speed of CPU's, also multiple processors weren't so popular. But as we said at the beginning, the world is different now and "the free lunch is over"[38], i.e. simple, sequential programming is not enough anymore. Hardware designers try to overcome the gap between memory and CPU speed by introducing caches, then caches of caches but then they also need a way of synchronizing the state of the memory, like the nowadays popular *Cache Coherency Protocol*, just to ensure that cores accessing the same address of memory get consistent results. This proves to be a much harder process to overcome in comparison to "just" increasing the clock speed of CPU's. With constantly increasing performance of the hardware and comparably slow increase in the efficiency of the software systems it becomes apparent that maybe the former is not the culprit in this relationship.

Reasoning about all aspects of concurrent systems is tremendously difficult, and the

efficiency losses which result from not doing this thoroughly are often too easily accepted. For accessing a shared piece of memory the most popular way still is to lock the region so that all operations are made atomic. This, in consequence, often leads to the creation of sequential processes which brings down all the advantage of having multi-core architectures. Although locking is the right way of programming in *C*, it shouldn't be right in a high-level language like C# or *Java*. We all known that it sometimes causes (very dramatic) deadlocks or race conditions. Greater experience always reduces the frequency of such errors, the question is to what extent and why the programming languages aren't so much of a help in this situation?

Concurrency has been the subject of theoretical research for many years. There are concepts of process calculi, like CCS or *π-calculus* which again at the very low, elementary level state the idea of a process. These were extensively analysed in terms of concurrent computation, communication and have raised many interesting concepts into the world of concurrency. Yet again, there is a kind of gap between process calculi and programming languages for distributed programming. There have been number of attempts to implement original process calculi directly into languages, like *Acute*[1] or *Pict*[2], but their awkward semantics seemed against the natural instinct of the programmers. Also, a low number of synchronisation concepts used in those process calculi means that their usefulness is underestimated. It is disappointing that the amount of research that goes into this field, rarely is introduced and accepted by the wider audience.

So why not to use a language which was designed to meet the reality of the present time? One that is distributed, fault-tolerant, with support for concurrent constructs and better memory concepts. Additionally we could move to a more a natural programming paradigm rather than another typical Object Oriented language which seems increasingly awkward as it doesn't reflect the reasoning about multiple threads or, more accurately, processes. An increasingly popular language, *Erlang*, fits in this description perfectly. Although it is typically categorized as a functional language, the concept of Actors, the life-cycle of processes plays a crucial role in its programming model and reasoning. It also throws away the shared memory idea and uses a message-passing concept.

Recently, a bridge between process calculi and programming languages for distributed, concurrent environments was introduced, called the *Join-calculus* [15]. *Join-calculus* shares many concepts with its predecessor, *π-calculus*, yet it was immediately defined in a way that is compatible with the current programming languages. This new process calculi shares with *Erlang* the concept of process as a main concurrency and computation unit. Its ideas also fill up the hole in the *Erlang* design, which misses the multiple addressing space for communication channels of a single process, and also synchronisation on multiple messages retrieval. Both of these can currently can only be done using very awkward, non-natural constructs. *Erlang* was designed to strictly mimic the current, asynchronous world, yet it feels that it lacks tools to express the synchronous nature that is also common. The aim of this project is to define an efficient extension of the original *Erlang* that implements the *Join-calculus* constructs. This way we would be able to produce better concurrent applications in a less error prone environment.

[1] see http://www.cl.cam.ac.uk/~pes20/acute/

[2] see http://www.cis.upenn.edu/~bcpierce/papers/pict/Html/Pict.html

1.1 Contributions

The main contribution of the project is an extension of the existing *Erlang* language that I called *JErlang*. The end result is available in a form of a library as well as a backward-compatible patched *Erlang* compiler and Virtual Machine. I decided to support both of the approaches since each has its own advantages and disadvantages and I did not want to be limited by any choice. The new language provides joins constructs borrowed from *Join-calculus* that efficiently use *Erlang*'s pattern-matching mechanism in order to synchronise on multiple messages. Therefore the original language does not loose in its functionality and gains powerful, yet simple construct, that fits into the overall *Erlang* architecture model. I decided to provide greater expressiveness at the expense of performance. This was necessary in order to create a language which a typical *Erlang* programmer would find acceptable. I believe that having a language with more powerful constructs already at the beginning is much better for comparison with the standard *Erlang*. This way it is easier to decide to drop some features in order improve the efficiency instead of adding some and observing how they affect the performance.

In chapter 2 I give a brief overview of the inspiration behind *Join-calculus* and present most popular implementations. Although the languages support the same idea taken from *Join-calculus*, the range of expressiveness differs in each implementation. I continue with a short definition of the *Erlang*'s programming model and how it fits in the world of concurrent programming. I also examine efficient pattern matching algorithms as well as techniques for static analysis of the code that were crucial for implementing correct language extension.

Chapter 3 presents a formal syntax and semantics of the mini *JErlang* language. This gives a firm base for the establishment of concrete language features that I explain in chapter 4. The latter can serve as *JErlang*'s reference guide. Chapter 4 also considers two possible join definitions and their applicability in the context of *Erlang* language.

In order to provide *JErlang*'s support in the run-time I needed to modify the existing *Erlang*'s Virtual Machine and compiler what is discussed in chapter 5. Similarly I present the challenges involved during the development of concise *JErlang*'s library without the need for changing the original compiler. I examine possible algorithms used for the efficient implementation of a non-trivial joins solver as well as optimisations for improving the performance in selected scenarios.

Chapter 6 evaluates the design decisions made in the previous chapter in the context of language expressiveness and performance. I present any trade-offs between those two using numerous applications and analyse to what extent *JErlang* is a better language than the original *Erlang* and other *Join-calculus* implementations.

Chapter 7 concludes the result of the project and suggests numerous research areas which could be further expanded or were missing in the current *JErlang*'s implementation.

Figure 1.1 presents a simiplified diagram of the *JErlang* architecture. In case VM-supported *JErlang* I use modified *Erlang*'s compiler and VM, but also different version of the *JErlang* module.

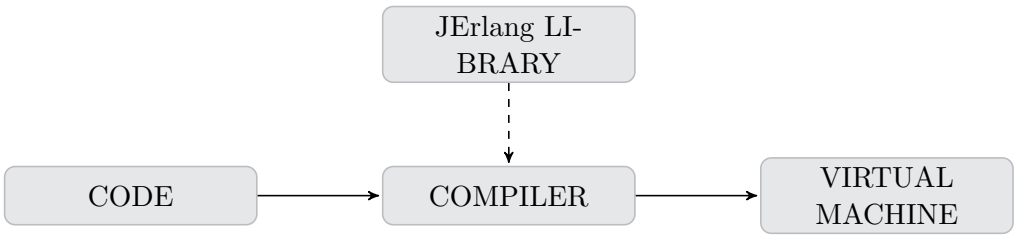

Figure 1.1: Simplified architecture used in *JErlang* language. The VM-supported *JErlang* uses modified compiler and VM instead of the standard *Erlang* one.

Chapter 2

Background

In this chapter we investigate the origins of *Join-calculus*, compare it with other calculi and describe in detail various existing implementations in modern programming languages to investigate their usefulness in programming concurrent systems by typical developers.We present the *Erlang* programming language that promotes novel, and often more natural, reasoning about concurrency issues and analyse its effectiveness in building fault-tolerant, distributed and highly concurrent systems. Finally we outline the efficient pattern-matching algorithms for solving large *Production Rule Systems* and variants of data flow analysis that statically analyse the code.

2.1 Join-calculus

Join-calculus came to the existence as a result of work done on extending a Chemical Machine (CHAM) of Berry and Boundol. The original CHAM raised an important fact - modelling chemical reactions could be very similar to modelling concurrent execution of processes. Unfortunately CHAM had a few design flaws, like the inability of adding new chemical rules to the solution and hence enforcing constraints on chemical reactions. The reflexive CHAM (RCHAM) developed in INRIA addresses these points by introducing two fundamental concepts: locality and reflexivity. *Join-calculus* was developed alongside as a way of formalizing the concurrency model and introduces an interesting construct called *multi-way join patterns* that enables synchronisation of multiple message patterns on different communication channels. This powerful, yet simple process calculus, was shown to be computationally equivalent to the π-*calculus* that addresses the question of mobility and communication between processes. This simplicity attracted many language architects as in theory it offers new concurrency constructs at a low cost of implementation.

2.1.1 Historical overview

A process calculus relates to the family of concepts for modelling concurrent systems. The aim of process calculi (also known as process algebras) was to provide frameworks for constructs of interaction, synchronisation and communication between sets of independent processes.

Tony Hoare, the inventor of the Quicksort algorithm and Hoare logic, started a new chapter in the analysis of mathematical theories of concurrency by publishing his pioneer book *Communicating Sequential Processes*[17]. *Communicating Sequential Processes* (CSP) presented a fresh look on interaction between processes and highly influenced the popular language Caml[1]. CSP as presented in the original paper wasn't fully a process algebra as was we understand it now, it was more of a language for writing concurrent models. Nor did it posses any proper mathematical syntax definition. Instead, CSP defined how processes which exist independently should communicate using only message-passing. In *Calculus of Communicating Systems* [28] (CCS), which based on CSP, Robert Milner presented a formal language for modelling concurrent interactions. His definitions of parallel composition, replication and scope restriction were widely studied and serve as concurrency primitives for building more complex interactions between processes nowadays.

The result of further work on CCS lead to the creation of *π-calculus*[29] by Milner, Parrow and Walker. The fundamental difference between the two is the introduction of the concept of mobility of processes in the latter, i.e. we take into consideration the situation when the execution location of the code might have changed.

The *π-calculus* relies on the concept of *names* which serve as identifiers for variables as well as for communication channels. Hence we gain the possibility of sending the channel names through other channels. This feature also enabled the achievement of computational completeness of the *π-calculus* because it introduces the possibility of recursive definitions. The constructs in the calculus are so primitive that *π-calculus* is often used as a basis for extension for other, more specialized ones and rarely exists in its pure form[2]. Implementation of *π-calculus* is complicated and constructs presented in the programming languages (Pict[3], Acute[4] or occam-π[5]), although powerful and perfectly valid, are hardly usable for day-to-day software development.

The radically different approach to the formalisation of the concurrency model was approached by the creators of the Γ-calculus. They claimed that "parallel programming with control threads is more difficult to manage than sequential programming, a fact that contrasts with the common expectation that parallelism should ease program design" [6]. Instead of having complete control over how concurrency was established between each of the entities, they allowed entities to "move freely". This way of thinking is similar to reactions that happens between molecules in biological terms. The biological solution in which the molecules lived enabled free interaction between molecules (processes, entities, values). The biological

[1] http://caml.inria.fr/
[2] see [1] and http://research.microsoft.com/en-us/um/people/aphillip/spim/
[3] http://www.cis.upenn.edu/~bcpierce/papers/pict/Html/Pict.html
[4] http://www.cl.cam.ac.uk/~pes20/acute/
[5] http://occam-pi.org/

solution was typically represented as a mathematical multiset, the rules that defined how molecules reacted with each other on contact were defined by tuples of conditions and actions which were later used during the execution step. The result was considered to be achieved when the solution is in a well-balanced state, i.e. no more conditions can be satisfied. Parallel programming represented in biological terms more naturally expressed the way concurrent execution happens since there can be numerous reactions happening at the same time. One drawback of the Γ was the much harder representation of sequential processes.

To simplify the implementation of the Γ-*calculus* concepts, G. Berry and G. Boudol extended the language using the Abstract Chemical Machine (CHAM). It provided structural meaning to molecules of the solution, so that they can be given meaningful syntax and semantics. CHAM can be therefore considered as a proper formalisation of the Γ-*calculus*. The reaction site can then be represented using normal operational semantics and typically molecules "move" to the proper reaction where they are sorted and catalysed to form actions defined in rules. The concept of a *membrane* is crucial for understanding CHAM. With *membrane* it is possible to create local solutions, containing one or more molecules, that only interact with the global solution whenever a restriction is available and matches with other molecules or *membranes*. This way the authors present some equivalence to the standard CCS and π-*calculus*, which uses the concept of channels. Unfortunately CHAM, had two significant drawbacks - recursion wasn't available due to the fact that no new solutions could be added to the solution and a single molecule needed to interact (in strict reaction locations) with all the other ones which makes its implementation unfeasible.

In 1995 C. Fournet and G. Gonthier came up with the so called reflexive Abstract Chemical Machine (RCHAM)[15] that fixed two of the original problems and raised a bridge between the two main concurrency models, process calculi represented by π-*calculus* and *Chemical Abstract Machine*. They designed *Join-calculus* which was a direct correspondence of the reflexive CHAM in the process algebra world. Reflexive CHAM enforces locality, meaning that molecules can be associated with a single reaction site instead of being mixed at many locations with all other molecules. The reflexivity of CHAM ensures that new reduction rules can be introduced by combining single molecules into more complex ones.

2.1.2 Reflexive CHAM

Similarly as in π-*calculus*, crucial for the relfexive CHAM is the concept of names that are used as values. We write $x\langle y\rangle$ for sending name y on the name x. The minimal grammar of the RCHAM as defined in [15] is:

$$\begin{array}{lll} P = x\langle\nu\rangle & J = x\langle\nu\rangle & D = J \triangleright P \\ def\, D in P & J \mid J & D \wedge D \\ P \mid P & & \end{array}$$

This way *process* `P` can be either an asynchronous message sending, a definition of new names (channels) or parallel composition of two processes. The join operation is either a retrieval of asynchronous name or a composition of two joins. Finally, the definition specifies the actions to be taken when all required join molecules are satisfied, i.e. whenever the reaction rule

matched molecules of which it is composed of, the process transforms into another process, as defined in the configuration.

For specification of the scoping rules it is helpful to clearly define the three types of variables that occur in the RCHAM as well as in *Join-calculus*: received, defined and free ones. When given an asynchronous polyadic message $x\langle\nu\rangle$ the only received variable would be the formal parameter ν since the value of ν will depend on the message name. Additionally this variable gets bounded only in the scope of the defined process (*process* `Q` in listing 2.1). Name `x` would be categorized as a defined variable and bound in the whole defining process - in listing 2.1 it would be bound in P.

```
def x⟨ν⟩ ▷ Q in P
```

Listing 2.1: Example of join definition in reflexive CHAM

The operational semantics of the reflexive CHAM as defined in [15] uses higher-order solutions $R \vdash M$, where the left hand side represents the rules of the reactions and the right side the molecules of the solution:

$$\begin{array}{llllll} (str-join) & & \vdash P \mid Q & \leftrightharpoons & & \vdash P, Q \\ (str-and) & D \wedge E & \vdash & \leftrightharpoons & D, E & \vdash \\ (str-def) & & \vdash def\ \mathrm{D}\ in\ \mathrm{P} & \leftrightharpoons & \mathrm{D}\sigma_{dv} & \vdash\ \mathrm{P}\sigma_{dv} \\ (red) & J \triangleright P & \vdash J\sigma_{rv} & \longrightarrow & J \triangleright P & \vdash P\sigma_{rv} \end{array} \tag{2.1}$$

The *dv* and *rv* refer to defined and received variables, respectively. Additionally we tend to define free variables with the usual meaning[6]. The first two rules express the commutativity and associativity of the molecules and definitions, respectively.

The σ_{rv} in (red) rule substitutes any received variables in the signature of the join operation with the ones in the body of the join. In the *Join-calculus* we will give more detailed explanation of the working of the reflexive CHAM.

2.1.3 Formal definition

Join-calculus was invented along with the reflexive CHAM as its complementary representation in process calculi. For implementation purposes it is much simpler to reason about. Additionally using this representation it is possible to show the structural equivalence with the π*-calculus* [15]. The grammar of *Join-calculus* is almost identical to the one represented by reflexive CHAM in definition 2.1. Apart from the different meanings of the constructs, we add an inert processes denoted by 0 and also a void definition:

[6] see [15] for detailed information

$$
\begin{array}{lll}
P, Q, R & ::= & \text{(processes)} \\
 & x\langle u \rangle & \text{(asynchronous message)} \\
 & \mid \quad def\ D\ in\ P & \text{(local definition)} \\
 & \mid \quad P \mid Q & \text{(parallel composition)} \\
 & \mid \quad 0 & \text{(inert process)} \\
D & ::= & \text{(definitions)} \\
 & J \ \triangleright P & \text{(reaction rule)} \\
 & \mid \quad D \wedge D' & \text{(composition)} \\
 & \mid \quad V & \text{(void definition)} \\
J & ::= & \text{(join patterns)} \\
 & x\langle u \rangle & \text{(message pattern)} \\
 & \mid \quad J \wedge J' & \text{(synchronisation on two patterns)}
\end{array}
$$

We take as an example a simple definition in *Join-calculus* terms:

$$\text{def D in P}$$

$$
\begin{array}{ll}
where & D \equiv J \wedge J', \\
 & J \equiv x\langle \mu \rangle \mid y\langle \nu \rangle \triangleright Q, \\
 & J' \equiv s\langle \sigma \rangle \triangleright Q', \\
 & P \equiv ... \mid x\langle s \rangle \mid y\langle t \rangle
\end{array}
$$

Exactly as in reflexive CHAM, at least one of the join patterns needs to be satisfied in order for the whole join definition to become satisfied and eventually be replaced by the appropriate process. In our example, we can be sure that join pattern `J` will become satisfied, since two of the required messages are actually sent in parallel with the process `P`. Obviously there is the possibility that join pattern `J`' would also have become satisfied in the underspecified part of the process `P`. When more than one join pattern is satisfied, we have a classical example of non-determinism of execution, i.e.

$$\text{a process } P \ \equiv ... \mid \ Q \text{ or } P \ \equiv ... \mid \ Q' \ \mid ....$$

Whenever a join pattern is satisfied the substitution for received variables takes place in the process defined in the reaction rule. Hence our example would actually resolve into $P \equiv ... \mid Q\{s, t/\mu, \nu\}$.

We provide a couple of examples of a *Join-calculus* programs representing typical process calculi and concurrency problems:

- Channel forwarding - $def\ x\langle \nu \rangle \triangleright y\langle \nu \rangle$ in P. Any message sent to x is transferred to y.
- Non-deterministic choice - $def\ c\langle\rangle \triangleright Q \wedge c\langle\rangle \triangleright Q'$ in $P \mid s\langle\rangle$. As we explained in the *Join-calculus* definition, whenever more than one join pattern is satisfied, the actual choice on which one will be fired is actually non-deterministic. We make use of that property and the end result might be either $P \mid Q$ or $P \mid Q'$.

- Continuation - $def\ x\langle\nu,\mu\rangle \triangleright \mu\langle\tau\rangle$ in P. Send the result of computation (here for simplicity arbitrary τ) to the explicit continuation. Here we assume the existence of polyadic channels. This can be easily encoded by having for example a join definition consisting of at least two patterns: $def\ x\langle\nu\rangle \mid y\langle\mu\rangle \triangleright \mu\langle\tau\rangle$ in P.

- Single-cell queue with initial state - $def\ get\langle\mu\rangle \mid s\langle\nu\rangle \triangleright empty\langle\rangle \mid \mu\langle\nu\rangle \wedge set\langle t\rangle \mid empty\langle\rangle \triangleright s\langle t\rangle$ in $P \mid empty\langle\rangle$. Channel (port name) `s` stores the internal value of the queue. It is not possible to overwrite the cell, while it is not empty, since `empty` message needs to be sent first and this happens only at the initialization point and after consuming the message.

2.2 Join-calculus implementations

Join-calculus, due to its simplicity and power, has recently become a very popular calculus to implement in existing languages (either as a language extension or a separate library). We used some of the ideas implemented in the presented languages to come up with a consistent view of what features *JErlang* has to posses.

2.2.1 JoCaml

JoCaml[7] is an extension of a well-known general-purpose language, *Objective Caml*[8] (OCaml) defined for easiness of expression. Similarly as *Join-calculus* and reflexive CHAM it was invented at INRIA, the French national research institute for computer science. OCaml is a mature object-oriented language with a powerful type system (based on ML). The reason for the development of *JoCaml* was to add new primitive constructs for the efficient development of concurrent and distributed systems. The original language was written by F. Le Fessant and later re-implemented by C. Fournet to provide a more suitable syntax for *Join-calculus* parts as well as to provide better compatibility with the standard OCaml language. For better understanding of the language we present numerous examples, basing on the official *JoCaml* reference manual [20].

JoCaml has two main entities in which programs are written: *processes* and *expressions*. The former are executed asynchronously and produce no end result, whereas the expressions are executed synchronously and have return values. The main new construct of *JoCaml* (in comparison to OCaml) is the port name used for inter-process communication, in *JoCaml* called *Channel*. The creation of the *Channel* entity naturally refers to the *Join-calculus* syntax.

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

Listing 2.2: Creation of simple *Channel* in *JoCaml*

[7] http://jocaml.inria.fr/
[8] http://caml.inria.fr/ocaml/

Listing 2.2 defines a new channel foo of type `int Join.chan`, i.e. of type channel that accepts 'int' values (carries the 'int; value). The `0` process is the empty process, and is necessary in this case since our channel is asynchronous. Hence the above definition leads to the creation of the non-blocking channel, since whenever a message is sent on it, it gets executed as all the join patterns, in this case only `foo(x)`, get satisfied immediately (more about join patterns will be explained later).

Another crucial entity of *JoCaml* are *processes*. Since only definitions and expressions are normally allowed to exist in the top-level systems code, we transform the processes into expressions through the usage of the **spawn** construct. This leads to the concurrent execution and the exact sequence of steps is implementation dependent.

```
1 # spawn foo(1)
2 # ;;
3 - : unit = ()
4 # spawn foo(2)
5 # ;;
6 - : unit = ()
7 12
```

Listing 2.3: Spawning *processes* in *JoCaml*

Listing 2.3 will, as shown, print "12", but a perfectly valid expectation is that it could also print "21". Synchronous channels can be represented in *JoCaml* by means of a continuation, as given in listing 2.4.

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

Listing 2.4: Synchronous channels using continuation in *JoCaml*

In this case the parameters for the `succ` channel are represented in the implementation as a tuple consisting of an integer and another channel. When all the necessary operations are finished, we simply send the message on the continuation channel. Writing channels with continuations is a perfectly valid operation in *JoCaml* but in terms of productivity, clarity and effectiveness it can become a very tedious and error prone process. Therefore, the language provides a syntactic sugar, which allows treating synchronous channels as functions. The exact equivalence of the example 2.4 would be presented as in 2.5.

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

Listing 2.5: Synchronous channels using build-in continuation-reply in *JoCaml*

The result of `succ` can be then used in other expression and one can easily see that the type of the expression is no longer a *Channel* but a function.

Writing join synchronisation patterns is done in *JoCaml* by declaring definitions with the names of the ports (channels) and they are all blocked until they become satisfied, i.e. receive an appropriate message (listing 2.6). Patterns of the join are combined using the

`ampersand` sign, so both channels in listing 2.6 need to receive at least one message on each of them to spawn the process defined in the body of the join definition. *JoCaml*allows also for composition of the processes in the expressions using, again, the `ampersand` sign, as in example 2.7

```
# def fruit(x) & cake(y) = print_endline(x ^ " " ^ y); 0
# ;;
val fruit : string Join.chan = <abstr>
val cake : string Join.chan = <abstr>
```

Listing 2.6: Simple join patterns in *JoCaml*

```
# spawn fruit("apple") & fruit("raspberry")
#          & cake("pie") & cake("crumble")
# ;;
- : unit = ()
apple pie
raspberry crumble
```

Listing 2.7: Join pattern composining of many channels in *JoCaml*

Depending on the implementation, a perfectly valid solution would again be "apple crumble", followed by "raspberry pie". This way we can see how the action defined in the join definition is only fired when all the patterns are synchronised. One restriction on the channels of the *JoCaml* join pattern is that the definition cannot contain channels of the same name. This resolves a potential non-determinism in synchronised channels where one would have to decide to which channel the value should be returned.

It is also possible to declare multiple join patterns in a single definition construct. This is done using a special keyword construct named `or` and creates possibilities for more interesting synchronisation patterns, as presented in listing 2.8.

```
# def apple() & pie() = print_string("apple pie"); 0
#   or strawberry() & pie() = print_string("strawberry pie"); 0
# ;;
val apple : unit Join.chan = <abstr>
val strawberry : unit Join.chan = <abstr>
val pie : unit Join.chan = <abstr>
```

Listing 2.8: Multiple join definitions in *JoCaml*

```
# spawn apple() & spawn strawberry & spawn pie
- : unit = ()
```

Listing 2.9: Spawning multiple processes in *JoCaml*

This way we define the "pie" channel only once, also in this definition making the pie, a synchronous channel would be a valid *JoCaml* syntax. Sending messages on all port names creates a likely non-determinism in what process will actually be spawned, because listing 2.9 will either produce an output "apple pie" or "strawberry pie" and is implementation dependent.

JoCaml contains an efficient pattern matching typical for ML[9]. It is possible to define join patterns that await on synchronisation on channels of specific pattern. This way one can create many interesting concurrency models like the one presented in 2.10.

```
# type fruit = Apple | Raspberry | Cheese
# and desert = Pie | Cake
# ;;
type fruit = Apple | Raspberry | Cheese
and desert = Pie | Cake
# def  f(Apple) & d(Pie) = echo_string("apple pie")
# or   f(Raspberry) & d(Pie) = echo_string("raspberry pie")
# or   f(Raspberry) & d(Cake) = echo_string("raspberry cake")
# or   f(Cheese) & d(Cake) = echo_string("cheese cake")
# ;;
val f : fruit Join.chan = <abstr>
val d : desert Join.chan = <abstr>
```

Listing 2.10: Usage of pattern matching for identification of join patterns in *JoCaml*

```
# spawn f(Raspberry) & d(Pie) & d(Cake)
# ;;
- : unit = ()
raspberry pie
```

Listing 2.11: Sending messages to channels defined in previous example in *JoCaml*

The result of sending the messages in listing 2.11 could be either “raspberry pie” or “raspberry cake”, depending on the implementation and timing issues.

Apart from the concurrency constructs as defined before, *JoCaml* has support for distributed programming. The execution of processes in *JoCaml* is location-independent, i.e. running two processes P and Q on two separate machines will be equivalent to running a compound (P | Q) process on a single machine. The are some limitations to this model, because the output/error port is different when run on different machines and the latency of the network has to be taken into account in the execution. However, the scope of the defined values and names will remain the same - port name might serve now as a channel address (possibly to another machine) and the locality issues will still remain transparent. Since the local instances of *JoCaml* aren't aware of any other ones, *JoCaml* needs the name-server for initiation of the exchanges of channels. This is primarily needed to establish the connection between the machines that do not know each other. Obviously, after the initial shake-up one can use normal channels since the instances are aware of each other.

[9]Pattern Matching is common in functional languages like *Haskell* or *Erlang*

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

# spawn begin
#     let sqr = (Join.Ns.lookup Join.Ns.here "square" : int -> int) in
#     print_int (sqr 2);
#     0
# end
# ;;
- : unit = ()
4
```

Listing 2.12: Distributed channels support in *JoCaml*

Listing 2.12 presents an example of the usage of the name server for the initial setup for the two machines running two different process, in which one is dependent on the other. The first process uses the `register` function to store its location and available function in the central name-server, whereas the latter process uses the function `lookup` to search for the `square` function (synchronous channel) in the *JoCaml* world and assigns it a local name. Hence, after setting the configuration, the execution can be easily continued as if all the processes were local. Depending on the type of the messages content, its value will be either copied or referenced. In case of a function, all code and values are replicated on the destination machine, however in case of synchronous channels only the name of the port is given and sending on the port name results in a typical remote procedure call, where all the execution takes place where the channel was defined and the result is returned to the caller.

2.2.2 Polyphonic C#

C# is a modern, type-safe, object-oriented programming language running on the *.Net* platform developed by Microsoft®. The language itself belongs to the closed-source category, but there are attempts to create an open-source version of C# on a platform called *Mono*. The language in its nature is very similar to the popular Java programming language. Both compile to their respective bytecodes, which then, in the case of C# , is run on the *.Net* runtime. Standard C# has quite a few concurrency constructs, starting from primitive (and most common) `lock` mechanisms where each object can be set as a target, through mutexes to complicated semaphores. Since C# 2.0 it is also possible to construct asynchronous-like methods called delegates, but they are not exactly equivalent to *Join-calculus* asynchronous channels. Even though C# has a few concurrency constructs, the complexity of writing concurrent and distributed programs is often high.

Therefore in [5] N. Benton, L. Cardelli and C. Fournet define the extension of C# with *Join-calculus* which they call *Polyphonic C#*. In comparison to *JoCaml* *Polyphonic C#* doesn't provide *Join-calculus* primitives like channels or processes, nor any way for parallelising these. Instead the authors took two useful elements and implemented them in an

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

Listing 2.13: Unlimited buffer in *Polyphonic C#*

extension of C# : asynchronous methods and chords.

Asynchronous methods

In a standard C# expressions are executed sequentially. Any called methods execute until completion, before other instruction in the code can be executed (even when they do not have to have a return value). There is a way to introduce the asynchronous methods through the complex use of threads or delegates but they are hard to understand and reason about, since it builds unnecessary code around a single problem (especially problematic for complex problems) and is an error prone activity. Running asynchronous methods doesn't wait for their finish, nor returned value or exception. The methods are run in a separate thread in parallel with the currently executed thread. Asynchronous methods in *Polyphonic C#* are introduced with the keyword `async` instead of method return type (which internally type system recognizes as `void`):

Definition 2.1 (Chords in *Polyphonic C#*)**.**
`async` *method name* (*method parameters*) { *method body* }

From the implementation point of view, whenever an asynchronous method is called, the method body is executed in a separate thread (by an expensive creation of a fresh thread or through usage of the possible thread pools). The method has to be packed in a new thread, queued and then dispatched by the appropriate process responsible for scheduling of the threads.

Chords

Chord is an object-oriented adaptation of the join patterns concept. Chords consists of a method header and a body. A method signature in *Polyphonic C#* consists of a list of method headers joined by the `ampersand` sign. The body of the chord is executed once all of the methods defined in the header were called. By default, any chord contains at most one synchronous method and all the others need to be asynchronous. This simplifies the general thread scheduling mechanism as well as the overall structure of the return instructions. A typical example of the chord in the *Polyphonic C#* is the buffer class presented in 2.13.

`Buffer` class from 2.13 defines a chord consisting of a synchronous method `Get` and an asynchronous `Put`. Any calls on a method `Get` of an object of class `Buffer` will get blocked, unless there were corresponding calls of a `Put` method. Whenever a call to the `Get` has a corresponding call to the method `Put` a chord body gets executed in a thread in which `Get`

was called. On the other hand, any `Put` calls with unmatched `Get`s get queued up until any call to `Get` is made. The decision upon which `Put` should be used is strictly implementation dependent and one should not depend on the order of the calls. Similarly one can have a chord consisting only of asynchronous methods(listing 2.14).

```
1 public class FooBar {
2   public async Foo(String x) & public async Bar(String y) {
3     Console.Write(x + y);
4   }
5 }
```

Listing 2.14: Chords definition in *Polyphonic C#*

In 2.14 example whenever the methods `Foo` and `Bar` are called, a chord is satisfied and the body gets executed. Since in the asynchronous-only chord there is no indication in which thread the body of the chord should get executed, a scheduler will either create a new separate thread or take a free thread from *thread pool*. The former is a quite an expensive process in C# .

As we mentioned before, the chords are very powerful constructs but it doesn't mean that they don't have to be used with care. It is quite easy to create a situation of non-determinism using chords.

```
1 public class NondetereministicBuffer {
2   public Object Get() & public async Put(Object s) {
3     return s;
4   }
5
6   public Object Get() & public async Signal() {
7     return null;
8   }
9 }
```

Listing 2.15: Non-deterministic buffer in *Polyphonic C#*

In an example 2.15 whenever a method `Get` is called one cannot make any assumptions on which chord is actually going to be called, if we have a common code where both of the (a)synchronous methods can get called.

Polymorphism

Polyphonic C# is a fully-featured Object-oriented Programming language with the inheritance, overloading and overriding constructs. However, as it was already noted in [27] and [10] that inheritance and synchronisation constructs are often in conflict. *Polyphonic C#* enforces serious restrictions and an example of it is a situation when one wants to override a method that is part of a chord, one has to re-define the whole chord. Otherwise one would be left with an inconsistent state of the program.

```
1 class A {
2   virtual void f () & virtual async g () { ... }
3   virtual void f () & virtual async h() { ... }
4 }
5
6 class B : A {
7   override async g () { ... }
8 }
```

Listing 2.16: Invalid override of chords in *Polyphonic C#*

Hence the code presented in 2.16 is an invalid sample of *Polyphonic C#*. If this wasn't the case, then the VM, given the object of class *B*, would always relate the calls to method `g` with the object of class `B` (with no connection to the chord defined in class `A`), and calls to method `f` would be deadlocked in the first chord. *Polyphonic C#* allows for simple inheritance, where the subclasses inherit any chords defined in the super class. In that situation however, the *.Net* VM has to consider not only the static type, but also runtime for finding necessary chords that can be signaled:

```
1 class A {
2   public int foo() & async bar() { ... }
3 }
4
5 class B : A {
6   public int foo() & async bar() & async test() { ... }
7 }
```

Listing 2.17: Valid overloading pattern in *Polyphonic C#*

In this example we overloaded the chord defined in class `A` by adding `test` asynchronous method to the chord. Again, the execution of the chords defined in this hierarchy is non-deterministic when on an object of class `B`, methods `test`, `bar` and finally `foo` were called. Adding chords revolutionises quite significantly the polymorphism method, and can sometimes minimize the possibilities of Object Oriented mechanisms, reducing this way the freedom of typical constructs to which programmers got used to in standard C# . [5] presents more examples of the possible Object-oriented constructs and restrictions in *Polyphonic C#*.

Implementation

The authors of *Polyphonic C#* encountered numerous issues when defining the scheduling mechanism for the chords. Internally each chord is represented as a bitmap - a set bit means that there was at least a single call to the respective method. This efficient data structure enables to quickly determine the state of the chord by comparing their state bitmaps with corresponding fixed bitmaps that represent chords ready to execute. As it was already mentioned, the cost of creating fresh threads is unacceptably high, especially for high performance systems using *Polyphonic C#*. The authors attempted to skew the scheduling of chords towards more synchronous chords. In other words, whenever an asynchronous method was called a check was done first on a synchronous one because then we are simply re-using the already existing, blocked thread, instead of creating a new one in case of an

asynchronous-only chord. Additionally an analysis of the paths in the code was made to minimize the chance of the synchronous method being blocked. The scheduler makes sure that asynchronous methods of the chord are run before the synchronous one. An effort was made to improve the performance of the chords, but it also complicated the implementation as well as made the behaviour of the chords unintuitive and unexpected for programmers. Because of that, the decision was made to remove priority scheduling from the original version of the language.

An important aspect of *Polyphonic C#* is that it is translated directly into valid C# code. The transformation also ensures the necessary synchronisation around the chords. To ensure thread-safe access, the implementation uses global locks on objects, but still the locks are different from the regular ones (on objects) that programmers can use - deadlocks are therefore avoided. The usage of a global lock creates obvious races for updating the state of the chord, the time-complexity of locking is small enough to be ignored.

Ignoring the aspect of previous chords performance optimizations the order in which chords are checked is sequential. Whenever a method on an object is called the scheduler scans for possible (satisfied) chords. However, as explained in [5] it is quite important to start the *scan* process even after the chord is satisfied.

```
1 class Foo {
2   void m1() & async s() & async t() { ... }
3   void m2() & async s() { ... }
4   void m3() & async t() { ... }
5 }
```

Listing 2.18: Possible deadlock in chords definition for naitve implementation of scheduler in *Polyphonic C#*

The possible sequence of executions in an example 2.18 might involve:

```
Thread 1. calls m1() and blocks.
Thread 2. calls m2() and blocks.
Thread 0. calls t() then s(), awaking Thread 1.
Thread 3. calls m3() and succeeds, consuming t().
Thread 1. retries m1() and blocks again.
```

In this particular case Thread 3 is quicker in consuming the remaining `t` method call (indeed no atomicity of actions is required) than thread 1. This process of execution however leaves Thread 2 deadlocked even though the method `s` was called at least once. The scheduler takes into account this scenario and runs the necessary *scan* process that eventually awakens Thread 2.

The advantage of *Polyphonic C#* is its compatibility with the original C# language. To enforce this connection the actual type of the asynchronous methods is `void`. This allows for convenient overloading and overriding of the standard methods that have the void return type, but the actual behaviour is still quite different.

2.2.3 SCHOOL and ^{f}SCHOOL

SCHOOL[8], the Small Chorded Object Oriented Programming language, is a feather light model of a properly built chorded language with a syntax similar to Java. It was developed by A. Buckley, S. Drossopoulou, S. Eisenbach and A. Petrounias at Imperial College London. Its role is essentially to model languages rather than providing the programmers with fundamental libraries or conditional control constructs. It focuses on the design of an Object-oriented language that captures only the essential features of concurrency constructs based on the *Join-calculus*. ^{f}SCHOOL, an extension of SCHOOL, that adds the fields to the objects and focuses on the study of their interaction with the chords. SCHOOL has a well studied syntax, semantics and formalisation part. It is normally described using the structural operational semantics defined in the appendix of [8]. The evaluation rules of SCHOOL determine its usage [32]:

- New: creates a new object of a given class and allocates a previously undefined address in the heap to this new object.
- Seq: executes the first element of the sequence, discarding its result and later executes the second one, finally returning its result.
- Async: starts the invocation of an asynchronous method. Method is queued up in the list of waiting methods, later dispatched and executed. The return value is of type void.
- Join: selects a chord in which there is a single synchronous method and joins it with the other asynchronous methods defined with it. The body of the chord gets executed, using the parameters of the methods defined in the chord.
- Strung: similarly as `Join` selects a chord, with the only difference that it consists of only asynchronous methods. The body of the chord is executed concurrently with the other expressions.
- Run: runs the expression with a given heap state.
- Perm: enables the non-deterministic selection of expressions to evaluate and the reordering of expressions in the execution

SCHOOL shares many features with the *Polyphonic C#*(2.2.2) and tries to formalize properly the notion of chords. The authors of the language thoroughly study the well-formedness of the language, its equivalence to the ^{f}SCHOOL as well as the soundness of the type systems, but at the current state, the language is more suitable for reasoning, rather than for the development of real systems. Along with the development of the language a Virtual Machine was created (similar to JVM), called Harp[10], that runs the previously compiled interpretable bytecodes.

[10] http://slurp.doc.ic.ac.uk/chords/

2.2.4 Join Java

Surprisingly, for *Java*, which is a very mature and popular language among developers, one doesn't have much choice over the possible concurrency primitives. Typically programmers use `synchronised` keyword for locking the regions in the code that will be only available to a single thread. This method tends to be used too extensively and in more complex and badly designed systems leads to deadlocks or starvation problems.

In [14] the authors present an experimental extension of the *Java* language, *Join Java*, that provides features similar to those given in *Polyphonic C#*. *Join Java* provides asynchronous methods of which the return value is of a special `signal` type. In reality it is implemented similarly as in 2.2.4 for `async`, and `signal` essentially represents `void` type and doesn't return any value. Calling an asynchronous method immediately returns to the caller and the execution continuous in a separate thread (after being queued and dispatched).

Join Java provides the way to synchronise on method calls through the usage of chords (authors also tend to wrongly call them join patterns). However in comparison to *Polyphonic C#*, *Join Java* has several differences:

- *Join Java* also allows for synchronous methods, but whenever one occurs in a chord it has to be the first method. Authors claim that it contributes to better coding style, but we believe that is just an unnecessary burden on programmers who are being constrained by the language.

- *Join Java* doesn't allow for inheritance of classes that contain chords (classes need to be final). Again the authors claim that this solution is far better then providing a subset of polymorphism in *Polyphonic C#* which is restricted at some points. *Join Java* does allow (as well as *Polyphonic C#*) for method overloading in chords.

- *Join Java* is the first to possess the capability of introducing determinism into the scheduling of chords. Typically, as it was noted already in *Polyphonic C#*, when two chords are satisfied the choice of which one will be run is non-deterministic. In the *Java* extension, two new keywords are introduced: `ordered` and `unordered`. They allow the programmer to influence the way chords are scanned and adds control over how the the language is functioning. The decision to introduce such a feature to the language (without properly fixing previous ones) is very controversial since it allows the programmer to change the execution process. Potentially it can also create code that is much more vulnerable to mistakes and harder to analyze.

Listing 2.19 presents typical code of a `Buffer` class written in *Join Java*. It contains two chords that are evaluated in the order in which they are defined. Hence whenever the methods `put` and `empty` were called on an object, and finally a synchronous method `get` was also called, then the first chord will *always* be fired. A lack of the `ordered` keyword would introduce non-determinism as in all other programming languages implementing *Join-calculus*.

Join Java maintains an internal data structure that stores all the required method definitions, as well as all the method calls made on the object. The compiler adds special code

```
1 class ordered Buffer {
2   Object get() & put(Object s) {
3     return s;
4   }
5
6   Object get() & empty() {
7     return true;
8   }
9 }
```

Listing 2.19: Buffer class with chords scanning sequence in the order of definition in *Join Java*

that initializes the chords structure of type `join.system.joinPatterns`. This way for every method definition in the chords we have `patterns.addPattern(TYPES, synchroniseD)` where `TYPES` represents argument identities and `synchroniseD` determines whether the method is synchronised or not. Similarly the bodies of the method calls call either the `addCall` or `addSynchCall` methods with appropriate parameters to notify about the calls made on the object. Each call on those methods initialises the scan for satisfied chords.

In [13] authors analyze different types of the pattern matching algorithm. This is to avoid the status state explosion problem as defined in the first version of the *Join-calculus* language. The authors propose an efficient tree data structure for storing the references for the pattern methods. For example *B() & C()* and *D() & B()* chords would share the reference to the method *B* leaf. This reduces the time of scanning when a new method call is made. Calling *D* provokes the scan only on the second chord, whereas call on *B* forces check on both of the chords (see [13][11] for details).

2.2.5 JoinHs and HaskellJoinRules

Haskell is a popular lazy functional language with a powerful type system. The standard library doesn't have much support for concurrent programming, but there exists an extension, *Concurrent Haskell* [24], which provides the notion of a process and inter-process communication. The former is introduced with the use of a new `forkIO` primitive. Hence whenever `forkIO` is called it creates another thread (i.e. a thread inside the *Haskell* VM, which is different from a UNIX OS thread). The interprocess communication is introduced by the new type primitive `MVar a` which is a mutable location with an empty state or value of type `a`. This allows for building simple semaphores, monitors etc, but writing concurrent programs is still unnecessarily hard.

There were two attempts at implementing the *Join-calculus* mechanism in *Haskell*. *JoinHs*[12] is an experimental approach to allow creation of asynchronous and synchronous channels, which can be distributed in a transparent way. It also allows for creation of join patterns using those channels. Joins are separated by the '|' sign (see example 2.20).

[11] page 11
[12] http://www.cs.helsinki.fi/u/ekarttun/JoinHs/

```
join ch1 | ch2 | .. | chn = P1
  ch1 | ... | chm = P2
```

Listing 2.20: Multiple Join definition in *JoinHs*

The synchronous channels are created in a similar fashion to the *JoCaml* implementation - through continuation. *JoinHs* provides a syntactic sugar (`sync`) for automatic creation of the continuation-reply argument, thus simplifying the process for the programmers. Since *Haskell* doesn't allow for sending multiple return values, *JoinHs* automatically handles conversion of the continuation-reply into single tuples. A simple counter example with joins (adopted from *JoinHs*) is given in listing 2.21.

```
createCounter = do
  join count n | inc reply = count (n+1) >> reply
  count n | get reply = count n >> reply n
  count 0
  return (inc,get)

main = do
  (i1,g1) <- createCounter
  let is1 = sync i1
  gs1 = sync g1
  is1 >> is1 >> is1 >> is1
  print =<< gs1
```

Listing 2.21: Joins used for definition of simple counter and its usage in *JoinHs*

The `createCounter` function initialises the join patterns definition for the channels `count`, `inc` and `get`. At the end of the function we send message 0 on channel `count` and return the names for the other channels. This way the caller binds the names to local variables `i1` and `g1` and can use them in the scope of the `main` function. The first join returns a null reply, since we only introduce synchronisation to `inc` to provide proper sequencing.

JoinHs re-uses a significant amount of ideas from *JoCaml*, including the concept of a name-server to support distributive executions. Nodes register any channels they want to expose to other nodes through the existence of a central name service. *JoinHs* implements basic fault-tolerance by unregistering channels which become inaccessible. However, in comparison to *JoCaml*, the asynchronous and synchronous channels are implemented using the same mechanism - through channel proxies. The Glasgow Haskell Compiler(GHC) creates two proxies, one is sent to the remote node and handles necessary initialization, serialization and communication whereas the local proxy of the channel performs reverse operations and at the end sends the result to the local channel. Such implementation allows for clear transparency of the distributed programming. The *JoinHs* is implemented on top of the *Concurrent Haskell* [24] and is converted into valid *Haskell* syntax through the preprocessor.

In a language *HaskellJoinRules*[13] the authors experiment with a composure of a language extension and external library to allow for Join-calculus-like chords in *Haskell*. *HaskellJoin-*

[13] http://taichi.ddns.comp.nus.edu.sg/taichiwiki/HaskellJoinRules/

Rules is based on the effort made to define *Constraint Handling Rules*(CHR), which is a declarative programming language that was defined formally in [12] (first sketches of the constraint solving mechanism were already presented in [11]). CHR allows for efficient constraint reasoning like simplification and propagation. The former uses common techniques to create simple constraints that are equivalent to the original ones. The latter refers to introduction of redundant constraints that eventually lead to simplification. An example of simplification rule on constraints is: $X \geq Y, X \neq Y$ which results in $X > Y$ Whereas propagation is used in: $X > Y, Y > Z$ that results in adding new constraint $X > Z$.

CHR rules are defined using multi-sets and incremental solving techniques (of which two main rules were described) to find non-trivial solutions to concurrent logical systems' problems. The language is especially popular for definition of multi-agent systems - it is commonly used in *Prolog* implementations (SICStus and SWI-Prolog), or verification and type systems. Example[14] presented in listing 2.22 focuses on using simple logical laws for defining constraint solver on $A \leq B$, $C \leq A$, $B \leq C$.

```
   A ≤ B,C ≤ A,B ≤ C.
1. C ≤ A, A ≤ B propagates C ≤ B (transitivity).
2. C ≤ B, B ≤ C simplifies to B = C (antisymmetry)
3. A ≤ B, C ≤ A simplifies to A = B (antisymmetry, B = C).
4. A = B, B = C.
```

Listing 2.22: Simplification and propagation used for finding equivalances in CHR

Because CHR is a concurrent constraint programming language it is interesting to notice the similarity between CHR and CHAM. The CHR constraints, defined in multiset store, are a direct parallel to molecules (a chemical soup) in the CHAM world and to *Channels* in *Join-calculus* as well. The presented examples also show similarity between the *Join-calculus* join definitions, CHAM reductions and CHR multi-headed guarded rules - that idea was analysed formally in [21]. The authors also present an extension of *Join-calculus* join patterns with guards, which significantly changes the complexity of the implementation. Guarded conditions allow for setting additional boolean constraints of the join patterns, which at the same time enables the building of more complex concurrency constructs.

Finally, another extension to the original join definition is proposed - propagation. It happens very often in *Join-calculus* constructs that one has to define a single channel that provides mutual exclusive execution to processes, similar to a `lock` construct. Since the programmers ensure that only a single message on a state's channel exists, only a single process is executed at a time in the join. The authors propose a construct, that automatically ensures that message on such a channel is not removed on successful joins scan (see listing 2.23).

```
auth(Cid) \ ready(P, Pcr), job(J, Cid, Jcr) ⟺ Pcr ≥ Jcr | send(P, J)
```

Listing 2.23: The effect of the propagation rule in CHR

Lam and Sulzmann defined *HaskellJoinRules* in [37], which is a different approach to *Join-calculus* implementation than what we have seen before. A typical Buffer example,

[14] http://www2.cs.kuleuven.be/ dtai/projects/CHR/

presented already in other implementations, is given in 2.24.

```
1  Channel Buffer where
2    sync  get :: Join Int
3    async put :: Int -> Join ()
4
5  Chord x@get & put(y) where
6    x = return y
```

Listing 2.24: Unbounded Buffer definition using chords in *HaskellJoinRules*

Such syntax and programming style is clearly intuitive with regards to *Join-calculus*, but *HaskellJoinRules* is the first implementation offering conditional guards for joins as well as propagation rules as presented in listing 2.25.

```
1  Channels CondBuffer where
2    sync  cget :: (Int -> STM Bool) -> Join Int
3    async cput :: Int -> Join ()
4
5  Chord x@get(f) & put(y) | f y where
6    x = return y
7
8  Channels AuthBuffer where
9    async auth :: String -> Join ()
10   sync  aget :: String -> Join Int
11   async aput :: Int -> Join ()
12
13 Chord auth(id) / x@aget(id) & put(y) where
14   x = return y
```

Listing 2.25: Conditional and primitively authenticated Buffer in *HaskellJoinRules*

The `CondBuffer` channel underlines the fact that conditional join guards can be specified using first-class *Haskell* functions, i.e. at run-time, which creates numerous possibilities for filtering the behaviour. Creating such concurrency constructs without guards is often much harder. Example 2.26 presents the propagation construct where `auth` asynchronous message is never removed.

```
1 { auth(id1), aget(id1), put(m2), aget(id2), aget(id1), put(m1) } →
2 { auth(id1), aget(id2), aget(id1), put(m1) } (simplification)→
3 { auth(id1), aget(id2) } (simplification)
```

Listing 2.26: Reduction of messages of join channels in a schematic representation using CHR semantics

2.2.6 Joins Concurrency Library

Lately, the *Polyphonic C#*(2.2.2) was included in the *Cϖ* (COmega) research programming language, which is an official C# extension[15]. This however enforces from developers the

[15] http://research.microsoft.com/en-us/um/cambridge/projects/comega/

usage of a concrete language, something that may not necessarily be compatible with the previous systems written in other languages. The introduction of generics to the C# and, more generally, the *.Net* framework, made writing libraries that extend language functionality more accessible. In [34] Russo presents a *Join Concurrency Library* which is a direct translation of the *Polyphonic C#* language into a fixed API, that programmers can use. Because the *Joins Concurrency Library* is partially language neutral it can be used by any language written on top of the *.Net* framework.

Class `Joins` serves crucial role in the *Joins Concurrency Library* - it is used for initialising the *Join-calculus* environment, defining asynchronous and synchronous Channels using the *.Net 2.0 delegates* feature that allows for first-class methods and joining respective channels into well defined chords. Similarly the body of the chords is defined using delegates. The library provides thread-safe synchronisation and the delegates of the body allow for accessing the scope of the class in which they are defined.

```
1  public class OnePlaceBuffer<S> {
2    private readonly Asynchronous.Channel Empty;
3    private readonly Asynchronous.Channel<S> Contains;
4    public readonly Synchronous.Channel<S> Put;
5    public readonly Synchronous<S>.Channel Get;
6    public OnePlaceBuffer() {
7      Join j = Join.Create();
8      j.Initialize(out Empty); j.Initialize(out Contains);
9      j.Initialize(out Put); j.Initialize(out Get);
10     j.When(Put).And(Empty).Do(delegate(S s) { Contains(s);});
11     j.When(Get).And(Contains).Do(delegate(S s) { Empty(); return s;});
12     Empty();
13   }}
```

Listing 2.27: Single-cell Buffer using the *Joins Concurrency Library*

Listing 2.27 provides single cell buffer, similar to the one presented in section 2.2.2. The `Joins` class needs to be explicitly informed about the number of channels provided, in order for the channels to be associated with this particular instance of the `Join` (lines 8 and 9). Those are easily defined using the classes from the *Joins* library, and generics allow for comfortable specification of the return values (synchronous channels) as well as parameters (synchronous and asynchronous channels). The usage of the encapsulation mechanism allows for the specification of internal locking mechanism, that ensures mutual exclusion on a single cell buffer, since outside of the class, channels `Empty` and `Contains` cannot be called, but `Put` and `Get` can. Calling `Put` when a buffer is empty, sets the `Contains` channels with the value representing the internal state. Any calls to `Put` will block, until the value is consumed, since there is no corresponding join pattern for `Put` and `Contains`.

2.2.7 Conclusion

Join-calculus is an interesting approach which raises a bridge between chemical abstractions, which are very similar to the massively concurrent world, process calculi, like π-*calculus*, and programming languages. In comparison to languages that strictly implement π-*calculus*, like

Pict or Acute[16], and have only a small number of users, since the concurrency primitives were too low level and too abstract to use, *Join-calculus* implementations seem like a natural step for the existing languages to adapt to the ongoing research in the world of concurrency. Typically, Object-oriented languages are extended with Chords, a function equivalent to the original join definitions idea. Implementing message passing between channels and the concept of process was often avoided since it required a different approach to programming. Fortunately, in functional languages like *OCaml* and *Haskell* the architects implemented the ideas from the original *Join-calculus* and programmers gained more synchronisation constructs without loosing expressiveness.

It is important to understand how the features of the language affected the programmers. *Join Java*(section 2.2.6) was never really accepted because we believe that their approach was often simplistic and claimed, without proper background, that their approach was much better than in for example *Polyphonic C#*. Still, it is the latter, which had proper evaluation, is more widely used among programmers. Hence, the importance of the solution to be conforming with the existing language design.

In terms of functionality, *Constraint Handling Rules* and *HaskellJoinRules* offers even more powerful and still reasonable constructs that have the potential of being widely used by the programmers when correctly implemented[17]. Even though *Constraint Handling Rules* omit the concept of processes and channels, it is interesting when it comes to defining and implementing logic behind join patterns. As described in section 2.3 guarded conditionals are a common construct in *Erlang* and it would be positive to have it included also for join constructs in *Erlang*.

Most of the presented implementations naturally fit into the original programming languages. Though simple, *Join-calculus* allows to lower the number of cases when primitive locking for mutually exclusive access to internal state or any other error-prone and complex construct, needs to be used. Writing concurrent applications should focus on analysing concurrent behaviour rather than thinking on how to provide safe shared variable. Join patterns seem very natural, partially due to their correspondence to chemical reactions, in expressing synchronisation on message based languages as well as equally expressive as π*-calculus*.

2.3 Erlang

Erlang is general-purpose functional language for concurrent programming. It came into existence in 1986 at the Ericsson Computer Science Laboratory and was designed for solving typical telecom problems like building zero-downtime systems, managing millions of concurrent operations or transactions. Although the gap between technology of today and that time is enormous, it is still (or even more) important to provide systems that are easily extensible, highly concurrent and distributed, fault-tolerant, non-stop-running and yet, simple in implementation. It is, obviously, possible to design such systems with enough time and budget, but the result often ends up being so complex that they are close to being unmain-

[16] http://www.cl.cam.ac.uk/ pes20/acute/

[17] *HaskellJoinRules*is still in the development phase as explained in http://taichi.ddns.comp.nus.edu.sg/taichiwiki/HaskellJoinRules/

tainable. *Erlang* went open-source in 1998 and since then is gaining attention for the features mentioned before.

Erlang programs are normally compiled into interpretable bytecode that can later be run on the virtual machine. Although running interpreted code may seem as a wrong thing for systems that are supposed to be fast and concurrent, the VM is quite efficient. In 2001, an experimental native compiler, High Performance Erlang Project (HiPE), was merged into stable *Erlang* branch and it is often possible to run the execution even faster.

Although *Erlang* was designed for creating massively concurrent telecom applications, in today's world it is a remedy to many problems that current systems are struggling with. The developers of other programming languages are doing whatever they can in order to extend their languages with new constructs and libraries. We believe that such an approach doesn't bring as many benefits as having a language that is from the start designed for developing concurrent and distributed systems. *Erlang* in comparison to other modern programming languages has support in the standard release for managing, analysing and detecting other nodes(machines) and proper utilisation of multiple cores available in massive applications.

2.3.1 Process

Joe Armstrong, the original creator of *Erlang*, coined a term Concurrency Oriented Programming (COP)[18], which aim is to set general rules for systems which main concerns are the concurrency design patterns. In a sense *Erlang* is precursor of the path that other languages for COP would need to follow. In OPP `Object` is a basic entity around which all the design rules are built. In *Erlang*, the *process* is the main concept of which systems are built of. It serves as a container for the execution of the expressions.

In an early stage of the *Erlang* development, a decision was made to make the variables in the expressions single assignment, and to use message-passing for inter-process communication. This was partially influenced by the *Concurrent Prolog* on top of which the language was originally built on. Such decision helped in further isolation of the *process* and removed any concurrency problems related to shared memory. Therefore there is no notion of locks in *Erlang*, and a lot of effort was put in the runtime in implementing efficient message-passing mechanism, so that the language does not suffer from poor efficiency typical for such systems.

Normally the programming languages allow for the creation of threads and processes, which involve kernel calls, memory initialisation and other memory- and CPU-intensive resources. *Erlang* allows for explicit declaration of its own *process*, inside its Virtual Machine, provides scheduling appropriate for the *Erlang* environment rather than being directly dependent upon the operating system.

```
1 spawn(fun() -> io:format("Foo bar", []), ok end).
```

Listing 2.28: Spawning a process from first-class function definition in *Erlang*

Listing 2.28 presents how simple (and fast) the actual creation of the *process* is. `spawn` function, uses the builtin method for the creation of the *process*. *Erlang* allows for much more fine-grained creation of the processes with many optional arguments, like the name of

[18] see `http://www.sics.se/~joe/talks/ll2_2002.pdf` and [2]

the module, parameters, node on which the process is to run and indication whether the *process* is to be monitored by the parent. Processes get identified through the unique *PID* value, which is always returned on successful spawning.

Erlang doesn't just provide the notion of a *process*, but also allows for connecting different processes and eventually creating hierarchies of processes. Processes' linking sets the exception propagation path between two or more processes. This notion will be described more in sections 2.3.4 and 2.3.5.

2.3.2 Inter-process communication

To enable inter-process communication, *Erlang* has a single mailbox for each of the existing processes which internally are to be accessed in a FIFO queue manner. Sending messages to other processes is fast and easy as presented in listing 2.29.

```
1 Pid ! {ok, foo, bar}
```

Listing 2.29: Sending message to the process represented by *Pid* variable in *Erlang*

The example sends the tuple to the *process* represented by variable `Pid` that stores valid unique identification value. Behind the scenes this expression results in putting the message directly in the mailbox of the receiving *process*. This construct is asynchronous and conforms to the *Erlang* thinking that until the sender receives the confirmation, it cannot be stated whether the message was actually received or processed, since the other side might have crashed. After all our world mostly works in an asynchronous manner. Obviously *Erlang* ensures mutually exclusive access to the queue, but it is much quicker than traditional 'lock and process' way of accessing shared memory.

The only possible way for the *process* to analyse the contents of the mailbox is to use the *Selective Receive* construct as presented in listing 2.30.

```
1  receive
2      {ok, Val1, Val2} when (Val1 = ok)->
3          Expr1;
4      {ok, Result} ->
5          Expr2;
6      {error, Error} ->
7          Error_Expr
8  after
9      Timeout ->
10         Timeout_Expr
11 end
```

Listing 2.30: *Selective Receive* in *Erlang*

The existence of an internal unmatched queue is meant for improving the efficiency of pattern matching - there is no need to match the message again against the fixed set of heads when we know that in the past it wasn't successful. *Selective Receive* will only be working on the mailbox of the *process* in which it is currently running and is not possible to analyse the mailbox of the other *process*. The `receive` clause allows only for matching a single head

of the pattern at a time, therefore definition of synchronisation constructs requires from the programmers quite a lot of innovation.

```
1  receive
2      {ok, res1, Result1} ->
3          receive
4              {ok, res2, Result2} ->
5                  io:format("synchronisation on res1 and res2", []),
6                  {ok, Result1, Result2};
7              {error, Error} -> Error_expr
8          end;
9      {ok, res3, Result3} ->
10         io:format("synchronisation on res3", []),
11         {ok, Result3}
12     {error, Error} -> Error_Expr
13 end
```

Listing 2.31: Join-like synchronisation using *Selective Receive* in *Erlang*

Listing 2.31 presents an example of a typical construct that currently *Erlang* programmers would use to wait for completion of computations. In line 2 we make an assumption that the first computation to finish is `res1`, and only later we wait for the second one (when saying computation is finished, we mean that the process received appropriate message). This normally wouldn't be a valid concurrent program since we shouldn't depend on the ordering of the messages as during waiting for `res2` we might get `{ok, res3, Result3}` and there is a chance for a deadlock. This can be of course fixed by duplicating the action on `res3` while waiting on `res2`. Similarly lines 7 and 9 perform the same operation and are redundant. This is a significant drawback in the powerful *Selective Receive* construct.

Writing systems that rely on priority messaging is also problematic in *Erlang*. The usual pattern is to wrap the true message in a tuple, where first element represents the priority, as presented in listing 2.32.

```
1  Pid ! {error, 'Unexpected action'},
2
3  receive
4      {critical, Message} -> ... ; %% process critical value
5      {error, Message} -> ... ; %% process error
6      {info, Message} -> ...  %% processs info
7  end.
```

Listing 2.32: Sending and receiving priority messages

This is the result of the language design, where each process has only a single mailbox. Ideally, the programmers should be able to dynamically create message queues for the *process*. This would also improve the efficiency of scanning the mailbox, because currently when a single match is found the unmatched and unprocessed queues are merged and the scanning repeats from the beginning. This problem gains in urgency whenever the size of the mailbox increases (the size of the mailboxes is only limited by the amount of memory available), which is not something uncommon and eventually becomes the bottleneck of the system. The only solution the architects of *Erlang* have for this problem is to avoid such situations and scan

```
1 try
2     Expr, ...
3 catch
4     exit:Reason -> ; %% handle exit
5     error:Reason ->  %% handle error
6 end
```

Listing 2.34: Catching exceptions in *Erlang*

the mailbox frequently using a single variable that matches all messages (see listing 2.33).

```
1 Pid ! {error, 'Unexpected action'},
2
3 receive
4     {foo, Message} -> ... ; %% process the message
5     {bar, Message} -> ... ; %% process the message
6     Other -> ... %% process some other message
7 end.
```

Listing 2.33: Pattern to keep the size of the mailbox small

Example 2.33 still it isn't much of a help as it is more of a recommendation rather than an appropriate solution.

2.3.3 Expression and Functions

In *Erlang* almost every clause is an expression and returns an expression. The language provides arithmetic and boolean operations typical for high-level programming language as well as conditional statements that can lead to building for example loops. *Erlang* only allows for single-assignment variables within function scopes. In reality this is neither a limitation nor it leads to variable count explosion. On the contrary, this enforces more appropriate usage of variables. The compiler and runtime ensure the appropriate usage of variables and for example sending unbounded variables in messages is forbidden.

Functions are fundamental constructs in the language and *Erlang* allows for writing short and efficient applications using its pattern matching abilities on them. Functions can have the same names but different arguments and hence define different actions for specific function calls. All function calls are synchronous and the callers end up waiting for the body of the function to finish before proceeding.

2.3.4 Exception handling

As most of the modern programming languages, *Erlang* has a well defined syntax and semantics for handling exceptions and errors. Similarly as in *Java* it is possible to surround the region of code using `try ... catch` semantics (listing 2.34).

Therefore the exception handling code is able to cope with different types of unexpected behaviour. Since exceptions are represented using tuples it is also possible to add appropriate

`case` statement or perform variable assignment/matching to the expression that can throw an exception.

The real power of *Erlang* exception handling is shown when combined with managing sets of processes. It is possible to set monitors or link different processes. The main purpose would be to determine an action to be done on the status change of the *process* we are interested in. Whenever a *process* `B` is linked to *process* `A`, and `B` dies for some reason, a message would be sent to the `A`'s mailbox explaining the reasons for crashing. Depending on the contents of the result, parent *process* `A` may decide to do some action or fail as well. This corresponds directly to the situation when a critical part of the application crashes and the rest of the system will not work correctly. Because it is hard to predict how the system will behave, to prevent this uncertainty it is better to fail quicker, which is one of the core principles of *Erlang*.

A reasonable exception hierarchy is critical when it comes to building complex systems. Since *Erlang* allows for easy monitoring of processes on different nodes, as well as catching *exit messages* from them. Additionally we are able to monitor other nodes' crashes and hence getting exit messages of processes of other nodes running on different machines.

Whenever a non-normal exit signal is received, the parent will also die, unless it belongs to the category of *system processes*. This is ensured by setting an appropriate `trap_exit` flag on the builtin `process_flag` method, as shown in listing 2.35.

```
1  process_flag(trap_exit, true),
2  Pid = spawn_link(fun() ->
3                       receive
4                           Other -> exit({error, {invalid, Other}})
5                       end
6                   end),
7
8  Pid ! {die, now},
9  receive
10     {'EXIT', Pid, Reason} ->
11         %%process the signal
12         ok
13 end.
```

Listing 2.35: Exit signal processing in *Erlang*

The fundamental (and often forgotten) feature of links is that they are symmetric, meaning that in a set of linked processes, whenever any of them dies and the other side doesn't belong to system processes, then all of them die. This is not always expected, therefore *Erlang* allows for creation of monitors which are basically asynchronous links. [3] defines numerous scenarios of dependent processes failing and the parent processes crashing as well and trapping the system messages.

When designing any system or an extension in *Erlang* is therefore crucial to remember about exception handling since it often defines the way any proper *Erlang* system is built. It can be an extension, library or simple application but in *Erlang* we always have to be to answer fundamental question: "What happens when it fails?" Exceptions are crucial in defining design patterns in *Erlang*, as it is shown in section 2.3.5.

2.3.5 Open Telecom Platform

Erlang wouldn't be much of a help for programmers if they would have to define the fault-tolerant systems from scratch themselves. After all, any programming language without additional generic libraries is just a nice addition not suitable for complex systems or rapid development, since maintaining *every* aspect of the system is unbearable. Ericsson developers hit into this problem in the early days of *Erlang*[2] and came up with an Open Telecom Platform - a set of design patterns, libraries, documentation and how to's to increase the productivity of the developers and provide sufficient guidance on how the real *Erlang* applications should be built.

One of the fundamental OTP design principles are *supervisor trees*. As mentioned in section 2.3.4, proper handling of exceptions allows for creation of fault-tolerant systems. The *Supervisor tree* is therefore a tree of processes where it is easy to define an action on processes' non-normal or normal exit. It is usually described in terms of a *supervisor* and *workers*, which the former has control over. There are two types of supervisor trees:

- one-for-one - whenever one of the workers fails, only this particular *process* is restarted.
- all-for-one - failing of a single worker provokes the restart of all the workers of the supervisor.

`Behaviour` is a common way in *Erlang* for introducing general frameworks - including supervisor trees but also typical client-server models. Typically *behaviours* are defined in modules through callbacks. It is natural to define an API for the modules, which is made public to others, as well as the implementation of the calls (callbacks) in a single module, as presented in listing 2.36, a simple example of a server-client model implementing `gen_server` *behaviour*.

In example 2.36 we present how simple it is to write complex applications using well-defined `behaviours` from OTP. In line 2 we define the type of the `behaviour` that our system has to implement. In a sense behaviours are similar to abstract classes known from *Java*, where only specific methods need to be defined in order to correctly create the implementation of the behaviour. In `gen_server` we have three types of messages that can be sent to the server (more about them later): *call*, *cast* and *info*. The only place where we specify the starting parameters of the server's *process* is in `start_link` function (lines 13-14). We can see how the complexity of setting up appropriate *process*, registering its name or simply maintaining the state is hidden from the programmers.

Synchronous Call

`gen_server:call/2` implements all the complexity related to sending the message and waiting for the reply. In order to be correctly handled we have to implement respective `handle_call` functions with appropriately matching headers. For instance the call in line 26 matches function in lines 32-33. The synchronous call is accomplished by returning an appropriate final value in the handling function as in line 33 for `state` call. `gen_server:call/2` for sending the message requires the name of the destination *process* as defined in `start_link`.

```
-module(simple_counter).
-behaviour(gen_server).
%% API
-export([start_link/0]).
-export([inc/1, dec/1, state/0, last/0]).

%% gen_server callbacks
-export([init/1, handle_call/3, handle_cast/2, handle_info/2,
         terminate/2, code_change/3]).
-define(SERVER, ?MODULE).
-record(state, {amount=0, last_action=none}).

start_link() ->
    gen_server:start_link({local, ?SERVER}, ?MODULE, [], []).

init([]) ->
    {ok, #state{}}.

inc(Num) ->
    gen_server:cast(?SERVER, {inc, Num}).

dec(Num) ->
    gen_server:cast(?SERVER, {dec, Num}).

state() ->
    gen_server:call(?SERVER, state).

last() ->
    gen_server:call(?SERVER, last).

%% ------------------- Callbacks
handle_call(state, _From, #state{amount=Num} = State) ->
    {reply, {counter, Num}, State};
handle_call(last, _From, #state{last_action=Action} = State) ->
    {reply, {action, Action}, State};
handle_call(_Request, _From, State) ->
    Reply = ok,
    {reply, Reply, State}.

handle_cast({inc, Num}, #state{amount=A} = State) ->
    {noreply, State#state{amount = (A + Num)}};
handle_cast({dec, Num}, #state{amount=A} = State) ->
    {noreply, State#state{amount = (A - Num)}};
handle_cast(_Msg, State) ->
    {noreply, State}.

handle_info(_Info, State) ->
    {noreply, State}.

terminate(_Reason, _State) ->
    ok.

code_change(_OldVsn, State, _Extra) ->
    {ok, State}.
```

Listing 2.36: Client-server model of the counter in *Erlang*

Asynchronous call

`gen_server:cast/2` performs necessary message wrapping and sending to the server to reduce the amount of code necessary to be written. It would be perfectly valid, if we used `cast` to send the message to some arbitrary name - `gen_server:cast/2` will catch an exception in this case and report the error to the caller. Similarly as in a synchronous call, the parameters of the `cast` should match at least a single `handle_cast` function (lines 20 and 40) to be successful.

All of the callback functions are made public but in reality the only *process* making calls to them would be of type `gen_server`. For instance, `terminate` would be called when we or the system decides that the server is to be shutdown, for cleanup purposes. Behind the scenes, the target of the (a)synchronous messages is is the server *process* started in line 14. It maintains a inner loop that receives messages, decodes them, dispatches to appropriate callbacks, returns replies when necessary and, most of all, ensures the appropriate design of the fault-tolerant system all the way. The beauty of that solution relies on the fact that the callers only rely on the interface provided by `gen_server` and the implementation can be freely changed.

Having a synchronisation pattern when using behaviours in *Erlang* is quite hard as it has to be encoded explicitly in the logic of the application, hence making the implementation more complex. For instance in `gen_server` firing an action whenever two functions were called requires maintaining some field in the internal state of the process. This state then has to be checked each time we handle the function call and fire the required action when necessary conditions are satisfied, as presented in listing 2.37.

In example 2.37 it can be easily noted that in order to correctly call the `synchr_action` one would also have to store the arguments of each function call. Another option of implementing synchronisation is to use functions' conditional guards but again the code gets complicated. Having more than two methods on which we have to synchronise increases the maintainability factor significantly. The best practice for the programmers would be to focus on the implementation of the applications' models rather than getting stuck in implementing correct control flow.

2.3.6 Conclusion

Surprisingly *Erlang*, a mature Concurrent Oriented Programming language, lacks better concurrency constructs that would enable it to become a much more expressive language. In comparison to other modern programming languages it is advantageous because it already uses processes, messages, mailboxes etc. *Erlang* currently only allows for having a single mailbox for each process, but implementing processes that have multiple addressing spaces through multiple mailboxes (a.k.a. channels) would increase the capabilities of the language. A lot of discussion on the *Erlang*'s mailing list was devoted to this issue increasing the importance of having something similar to first-class channel entities. It is easy to see a lot of overhead that needs to be done for synchronisation of only two processes, not saying anything about more.

We believe that following OTP design rules for any invented extension is crucial for the

```
1  [...]
2  handle_call({first, Args}, _From, #state{second=Num} = State) ->
3      Val = case Num of
4                0 -> %% continue normally
5                _ -> synchr_action(Args), Num - 1
6            end,
7      %% handle call
8      {reply, Reply, State#state{second=Val}};
9  handle_call({second, Args}, _From,  #state{second=Num} = State) ->
10     Val = case Num of
11               0 -> %% continue normally
12               _ -> synchr_action(Args), Num - 1
13           end,
14     %% handle call
15     {reply, Reply, State#state{first=Vall}};
16 handle_call(_Request, _From, State) ->
17     Reply = ok,
18     {reply, Reply, State}.
19
20 synchr_action(Args) ->
21     % details of the function
22 [...]
```

Listing 2.37: Pattern for implementing synchronisation on two function calls in `gen_server` in *Erlang*

successful acceptance of *JErlang*. For instance we would need to choose an appropriate `supervisor` pattern for handling multiple channels as well as appropriate fail-over strategy for exceptions encountered. *Erlang* supports asynchronous message-passing, but with some syntactic sugar it is quite easy to have synchronous version. Again, appropriate OTP strategy would enable for the programs to recover from waiting on a synchronous message on a systems crash, something that other extensions do not support.

2.4 Solving pattern-matching problems

The process of pattern-matching is important in both *Join-calculus* and *Erlang*. We give brief a description of of a typical system, which aim is to process patterns and present efficient algorithm for solving them.

2.4.1 Production Rule System

Production Rule System[19] is a term often used in the branch of Computer Science, A.I. Such system is used for the inference of knowledge expressed in terms of goals. We name the process of creating new knowledge terms simply an action. We use well-defined rules in order to determine which actions can actually be fired. Testing of the rules is performed through the process of pattern-matching. The set of states available in the current time is often known as *Working Memory*, whereas the possible inference rules are named *Production Memory*. The role of the system is to perform efficient pattern matching to achieve new facts about the current state. Figure 2.1 represents the schematic diagram of the whole process[20].

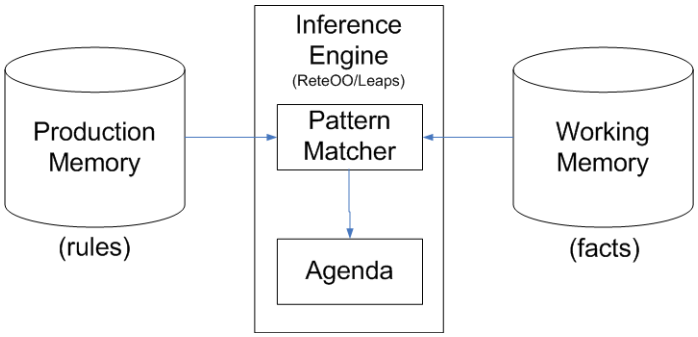

Figure 2.1: Production Rule System

2.4.2 RETE algorithm

RETE[21] is a fundamental algorithm used for efficient solving of *Production Rule Systems*. Although it was published in 1982 it is still a fundamental approach for all the possible derivative algorithms.
With RETE, we are able to build efficient structures represented by multiple nodes that correspond to the patterns used on the left hand side of the rule. In the following example **mom** and **dad** are basic patterns and **parents** is a possible new term that is created in the action:

```
mom(gosia, Child), dad(lukasz, Child)  =>  parents(lukasz, gosia, Child)
```

[19] see http://en.wikipedia.org/wiki/Production_system
[20] see http://www.jbug.jp/trans/jboss-rules3.0.2/ja/html/ch01.html for the source of the figure
[21] see http://en.wikipedia.org/wiki/Rete_algorithm

Algorithm allows for direct linking between the chains of nodes, thus creating dependencies between them. The increased speed efficiency however also increases the memory usage, which in some unrealistic situations can lead to crash of the system. Some of the other advantages of using RETE involve[22]:

- Reduces the amount of redundant operations done on the patterns.
- Partial evaluation of the sets of patterns allows for the steady building of knowledge necessary to trigger actions. This on the other hand helps in avoidance of complete re-evaluation of the existing patterns during each iteration of the algorithm.
- Rule and state's removal does not destroy the integrity of the whole search network, therefore promoting knowledge distribution among all of the nodes.

Alpha reduction

Alpha reduction is a name given to running simple test functions on the current facts from the *Working Memory*. Alpha-functions only check the consistency against independent patterns in a separate manner. This fast processing allows for filtering states that do not contribute to the production rules. Tests performed in this stage usually use only single inputs. If the individual state is succesful in matching the pattern, it is enqueued along other successful terms to this *alpha-memory* (pattern). It is latter used as input to the beta reductions. Whenever rules can be removed or added to the system, it is very easy to identify which elements of the alpha memory has to be updated.

Beta reduction

Beta reduction is an optional fragment of the rules resolution system, where we attempt to identify any conflicts existing between the corresponding alpha results. The natural intuition would typically suggest to test all the combinations for the sequences of alpha-patterns from the rules, but this approach ends up being relatively inefficient. Instead we perform tests on an increasing number of patterns, thus steadily filtering sequences of terms from the alpha reductions that cannot create consistent partial results for the rules.

Each beta reduction takes two inputs for processing - the first is the result of the previous beta memory that includes some part of the LHS of the rule and the second is the next alpha memory associated with the rule. This way we no longer work on raw states and we increase the efficiency of the processing. The first beta reduction is shipped with the dummy beta memory in order to provide consistency among the algorithm. The result of the action is then fetched (if necessary) into another beta reduction. The single action is fired whenever there are no more alpha memories to be joined against thus meaning that we have found a sequence of terms satisfying the LHS of the rule.

Beta reduction is performed sequentially for all of the possible patterns. When there are no more rules to be checked the production system arranges the rules that can be fired according to some key, and the actions are finally fired (commit step). This final part is

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

often named conflict resolution.
Typically the new facts that are produced in the last step are pushed onto the separate *Working Memory*. We do not allow for mixing them with the old facts, because any alpha reductions done previously are already valid. This way, the system does not have re-evaluate existsing solutions, a characteristic that is typical for inefficient pattern-matching solving. The RETE algorithm stops whenever there are no further facts that be inferred given the specified rules.

2.4.3 Optimisations

Numerous extensions of the original RETE algorithm were proposed with varying success. The main focus was directed onto improving the sequential nature of the algorithm. A typical examples of such improvements include TREAT[30] and LEAPS[4] algorithms. The former does not store the partial results of the computations and thus is better when the memory overhead becomes the bottleneck of the computation. The latter relies on the optimisations that comes with lazy evalutation of the patterns. With the rise of multicore machines, there is an increasing attempt to create parallelizable versions[25] of all of the above algorithms. Typically, the (hardware and software) implementation involves splitting the *Working Memory* into reasonable chunks that can be analysed by separate processes independently.

2.5 Data Flow Analysis

One of the main responsibilities of the compiler is to create efficient, optimised code. Data flow analysis is a technique for gathering information about possible values for the specific points in the program, like assignments for variables or arguments to the function calls. The analysis is run on the graph representation of the program, where each program clause is a node. This allows for creation of clear flow paths within the program and define for instance propagation of the value among the variables. Typically in order to formally define some property for each program node, we have to define a set of equations. Solving those, using the the stabilising property of the programs, allows to reach an equilibrium point, fixpoint, when we are able to state the minimal/maximum value for the property, depending on its type.

There are four, so called, classical types of analysis of the program: *Available Expression Analysis*, *Very Busy Expression*, *Reaching Definitions Analysis*, and *Live Variable Expression*. For the purpose of our extension to *Erlang* we will only adapt two last types, and the other two follow similar pattern and are described in detail in [31].

2.5.1 Reaching Definitions Analysis

The aim of the *Reaching Definitions Analysis* is to determine for each program point, which assignments may have been made and not overwritten, when program execution reaches this point along some path.
For the purpose of *JErlang* language, where variables cannot be overwritten, we only determine which variables were defined along some path when program execution reaches a given expression.

2.5.2 Live Variable Analysis

The aim of the *Live Variable Analysis* is to determine for each program point, which variables may be *live* at the exit from the point.
For the purpose of *JErlang* language we define the variable to be *live* at the exit from the point (expression) whenever there is another expression among the sequence or in the guards that uses the variable. We will show that this limited definition is enough for the purpose of our analysis in *JErlang*.

2.6 Summary

Process and expression are basic entities in *Erlang*, so any *Join-calculus* extension would probably be treated by programmers as less experimental. Having *Join-calculus*-like synchronisation on messages or function calls is sensible when writing highly concurrent, distributed and complex applications. Therefore Erlang with Joins could increase the programmer base of Erlang.

We believe that an introduction of new basic entity (apart from process and expression) might increase the complexity of the language and eventually lead to re-inventing the existing constructs (like *Selective Receive*) for multiple versions. *Selective Receive*, as a main message passing construct could therefore inherently allow for better synchronisation on messages with the use of join patterns. Any solution to the problem, should be able to explicitly create or destroy its channels as well as define join patterns on messages of the same or different channels.

It seems useful to extend the functionality of `gen_server` behaviour to enable synchronisation on functions level (like in Chords extensions) rather than maintaining an internal state which is an error prone process. This way *JErlang* would not only serve as a bridge between *Erlang* and *Join-calculus*, but also between functional and Object-oriented programming.

Chapter 3

JErlang: Formal Definition

In this section we give an overview of *JErlang*, an extension of the standard *Erlang* language definition. Our work is based on [41], which we consider to be the most mature research on the subject of formal definitions of this language. That thesis focuses on the formal definition in order to develop a Proof System for the language. It contains a set of syntax and operational semantics, but note that its style differs from the one presented below. This is because the aim of our research was to focus on the semantics of pattern matching in the context of joins, and as a result we omit details about distributed systems as well as *JErlang*'s exception hierarchy and failure propagation, to maintain clarity.

We start by defining the syntax of *JErlang*, detail the available transitions in the semantics and connect it with the possible join resolution strategies, which influence the way our implementation is done.

3.1 Syntax

The grammar given in 3.1 is presented in the *Backus-Naur Form (BNF)* with additional extensions to make it more elegant and avoid unnecessary repetitions. We use [e] to imply an optional construct e but also have a similar terminal symbol [e] for the list construct (to agree with the original *Erlang* syntax). For clarity reasons we include *overbar* notation for expressions ($\overline{e}$) and values ($\overline{v}$) to denote the (possibly empty) sequence of elements separated by comma. We also describe the sequences of values or expressions by using optional indices (v_i) to underline the order of occurrence.

Definition 3.1 (Value)**.** *JErlang* values represent a subtype of the available *JErlang* expressions. We distinguish between the basic and complex values, where the latter compose of themselves of the former ones. To the first group belong: *Value*

- *atom* - name without any whitespace signs, has to start with the lower-case letter.
- *number* - sequences of integers.

function	::=	$\overline{functiondef}$
functiondef	|	$(\overline{p})$ **when** *guard* $\rightarrow$ *e*
expr(*e*)	::=	e_1 op_e e_2
	|	$e(\overline{e})$
	|	**case** *e* **of** $\overline{match}$ **end**
	|	**receive** $\overline{join}$ **end**
	|	e_1 ! e_2
	|	e_1 , e
	|	$p =_{\mathbf{m}} e_2$
	|	*basicvalue* | *varId* | **{** $\overline{e}$ **}** | [$\overline{e}$]
basicvalue	::=	atom | number | pid | funcId
value(*v*)	::=	*basicvalue* | **{** $\overline{v}$ **}** | [$\overline{v}$]
pattern(*p*)	::=	*varId* | *basicvalue* | **{** $\overline{p}$ **}** | [$\overline{p}$]
match	::=	*p* **when** *g* $\rightarrow$ *e*
join	::=	*jpattern* [*join_aux*] [**when** *g*] $\rightarrow$ e
join_aux	::=	**and** *jpattern* [*join_aux*]
jpattern	::=	*propagation p*
propagation	::=	**true** | **false**
guard(*g*)	::=	g_1 op_g g_2 | *basicvalue*
	|	*varId* | $g(\overline{g})$ | **{** $\overline{g}$ **}** | [$\overline{g}$]

Figure 3.1: *JErlang* syntax in a BNF-like form. It introduces joins to *Erlang* by extending the definition of **receive**

- *pid* - feature of *JErlang* that represents the unique identifier of the process.
- *funcId* - unique identifier to the definition of the function.

Compounded variables are represented through lists [v_1, ... v_2] and tuples, **{** v_1, ... v_2 **}**. The main two differences between the representations are access of the elements and pattern matching. There is no easy way to expand a tuple, whereas lists are easily expandable. In the case of lists, the standard notation is the shorthand from the one used in for example *Haskell*, where lists are built using *nil* and *cons* constructs and drives the way elements can be accessed.
For instance [**1,2**] is represented internally as [**1** | [**2** | []]].

Although *JErlang* doesn't have the separate type for *boolean* values as in other languages, atoms `true` and `false` are commonly used, instead. This doesn't limit the language's abilities, as pattern matching is used on the representation of the atoms. It is important to distinguish the syntactic *propagation* boolean values from the semantic ones.

Variable

Definition 3.2 (Variable)**.** A variable is a sequence of characters that start with an uppercase letter. We distinguish between two types of variables: *unbounded* and *bounded*. The latter represents a unique mapping from the identifier (name of the variable) to the value.

Variables in *Erlang* can only be assigned once through its lifetime. Variables are used for pattern matching to find the correct sequence of expressions that match. We provide detailed definition of free variables and substitution in 3.17.

Definition 3.3 (Function)**.** *JErlang* functions define a frame in which their expressions are executed. We allow for having multiple function definitions associated with the identifier, and finding the right one is done by comparing the headers of the functions. To sum up *JErlang* has the possibility of having: *Function*

- (1) functions with the same name and number of arguments but different patterns in the headers
- (2) functions with the same name and different number of arguments

Case (1) uses efficient pattern matching that analyzes the existing function headers, as in listing 3.1, whereas the latter case treats the function definitions separately and first finds the identifier of the function with the required number of arguments (see the different declaration in listing 3.2).

In further sections, for simplicity, we assume that functions have a fixed arity of one and arguments are represented in a tuple. This doesn't decrease the applicability of the language and allows us to easier present and understand the semantics of the language (see example 3.4).

Example 3.4 (Representation of the function definitions)**.** Function with multiple headers
$f(\overline{p})$ **when** $g \rightarrow e$
is internally represented as
$f(\{\overline{p}\})$ **when** $g \rightarrow e$.

```
1  foo(bar, Value2) ->
2    \%\% function body... ;
3  foo(Value1, bar) ->
4    \%\% function body... ;
5  foo(Value1, Value2) ->
6    \%\% function body... .
```

Listing 3.1: Functions with the same number of arguments

```
1  foo(Value1) ->
2    \%\% function body... .
3
4  foo(Value1, Value2) ->
5    \%\% function body... .
```

Listing 3.2: Functions with different number of arguments

Definition 3.5 (Guard)**.** Guards allow for additional filtering of messages (in **receive**) and appropriate statement executions (in **case** statements and function definitions). Guards allow for calls to the built-in functions and standard relational, arithmetic and boolean *Guard*

operations, but without side effects. Hence, the range of functions available in guards is very limited and one is for example not allowed to bind a value to the fresh variable in this section. The role of guards is only to assist you in getting more expressiveness, and not in performing complicated execution.
As a result, we can imagine that the guards section always exists, but is equivalent to "**when** `true`" whenever it is not provided explicitly. In *JErlang* we only allow for the existence of bounded variables in guards in the given context. This makes the explanation of further semantics easier.

3.1.1 Differences between Erlang and JErlang

The main difference between the syntax of minimal *Erlang* and *JErlang* is in the definition of **receive** statement. In the *Erlang* we are only allowed to have a single *pattern* and in the latter we explore the power of joins by having the possibility of synchronization on more than a single message. Additionally we have the possibility of propagation on the message (as explained in 3.2). This change enforces us to use new semantics for receiving the messages as we explain in the next section.

3.2 Semantics

$\rightarrow$ **Definition 3.6** (Partial functions)**.** Notation A $\rightarrow$ B denotes the set of partial functions such that for f $\in$ A $\rightarrow$, and i $\in$ A, f(i) $\in$ B or f(i) = *Udf* otherwise.

F **Definition 3.7** (Frame)**.** *Frame* denotes the mapping of variables into values, in the given local context:
F : *VardId* $\rightarrow$ *value*
For obvious reasons, each function has its own environment and inherits only values of arguments from the caller's frame.

Dom(F) We also define the notion of the domain of the frame F, such that it is a set of all the valid variables in the frame mapping F:
$Dom(F) = \{k \mid F(k) \neq Udf\}$

Q **Definition 3.8** (Queue)**.** Each process in *JErlang* is associated with a single *Queue* defined as:
Q : $value^*$
Messages in this mailbox are stored consecutively from the oldest to the newest. To allow accessing of the messages not only from the head of the queue we add indices for each message,
Range(Q) and $Range(Q)$ represents the set of indices in the increasing order.. We also introduce the
$|Q|$ notion of length on the queue, denoted as $|Q|$.

We define the existence of the following operation on the *queues* as follows:

$$Q(\mathrm{i}) \quad = \quad \begin{cases} value_i & \text{if } i \in Range)\,Q) \\ \textsc{Udf} & \text{otherwise} \end{cases}$$

Example 3.9 (Queue length)**.** Consider the queue:

$$\begin{aligned} Q &= (value_1 \cdot value_2 \cdot value_3) \\ Q' &= (value_1 \cdot value_2) \end{aligned}$$

The lengths of the queues are: $|Q| = 3$ and $|Q'| = 2$

Definition 3.10 (Sublist operation)**.** Relations $<_q$ and $\leq_q$ for the queues define that the former queue is smaller in size or smaller and equal than the latter, respectively. Additionally all the elements in the first queue are consecutively taken from latter, starting from the first element, i.e. $<_q$ $\leq_q$
$Q <_q Q' \Rightarrow |Q| < |Q'| \wedge \forall (i \in Range(Q))\ (Q(i) = Q'(i))$

Definition 3.11 (Module)**.** The module function maps identifiers to function bodies. The latter are represented through the **case** choice statements for simplicity, as it helps in understanding the operational semantics: *Module*
Module : *FuncId* $\rightarrow$ funtionBody

Example 3.12 (*Module* access)**.** Given the *JErlang* program as in listing 3.3:

```
1  foo(bar, Value2) ->
2    io:format("Testing", [Value2]),
3    ok;
4  foo(Value1, bar) ->
5    io:format("Second choice", [Value1]),
6    error.
```

Listing 3.3: Sample function in *JErlang*

Module(foo) would be represented as:

```
case e of
  {bar, Value2} ->
    io:format("Testing", [Value2]),
    ok;
  {Value2, bar} ->
    io:format("Second choice", [Value2]),
    error
end
```

Definition 3.13 (*JErlang* execution)**.** Executions rewrites tuples of *expressions*, *frames* and *queues* into tuples of *expressions*, *frames* and *queues*. Execution takes place in the context of the module (program) where all the user-defined functions exist. $\rightsquigarrow_E$
Thus, the signature of the rewriting relation is:

$$\rightsquigarrow_E : \text{module} \rightarrow e \times F \times Q \rightarrow e \times F \times Q$$

Definition 3.14 (Structural equivalence)**.** Symbol $=_s$ is used to denote the syntactic structural equivalence between two expressions as defined in the original BNF syntax in 3.1. $=_s$

$\widehat{op}$ **Definition 3.15** (Operations on values)**.** To allow the evaluation of the expressions that contain different operations on values we introduce the evaluation function $\widehat{op}$ that performs execution of the given operation *op*. For instance $2 + 3$, would return the result of $\widehat{+}(2,3)$, similarly $13 > 21$ would call function $\widehat{>}(13, 21)$. We assume the intuitive existence of those operations, with the extension that they return **error** whenever the execution of the operation is stuck. We assume that operations do not have side-effects.

self() **Definition 3.16** (Process identifier)**.** The helper function *self()* returns the identifier of the process in which we are currently executing. Standard *Erlang* provides complex strategies of linking between processes to provide stable, fault-tolerant systems but we omit those details and just assume that whenever processes want to communicate between each other they know their respective *pid*s.

free **Definition 3.17** (Free variables in the frame context)**.** The set of free variables in *JErlang* expressions are detected using *free* function, where $\triangleq$ defines the result of running the function on the expression:

$$
\begin{array}{rcl}
free(e_1\ op_e\ e_2,\ F) & \triangleq & free(\mathrm{e}_1,\ F) \cup free(\mathrm{e}_1,\ F) \\
free(e_1\ \mathbf{!}\ e_2,\ F) & \triangleq & free(\mathrm{e}_1,\ F) \cup free(\mathrm{e}_1,\ F) \\
free(p_1\ =_{\mathbf{m}}\ e_2,\ F) & \triangleq & free(\mathrm{p}_1,\ F) \cup free(\mathrm{e}_1,\ F) \\
free((e_1,\ e_2),\ F) & \triangleq & free(\mathrm{e}_1,\ F) \cup free(\mathrm{e}_1,\ F) \\
free(\{\ \overline{e}\ \},\ F) & \triangleq & \cup_i\ free(\mathrm{e}_i,\ F) \\
free([\ \overline{e}\ ],\ F) & \triangleq & \cup_i\ free(\mathrm{e}_i,\ F) \\
free(basicvalue,\ F) & \triangleq & \{\} \\
free(\textbf{case}\ e\ \textbf{of}\ \overline{match}\ \textbf{end},\ F) & \triangleq & \cup\ free(\mathrm{e},\ F) \cup_i\ free(\mathrm{match}_i,\ F) \\
free(\textbf{receive}\ \overline{join}\ \textbf{end},\ F) & \triangleq & \cup_i\ free(\mathrm{join}_i,\ F) \\
free(p\ \textbf{when}\ g \to e,\ F) & \triangleq & free(p,\ F) \\
free(jpattern_1\ \textbf{and}\ ...\ \textbf{and}\ jpattern_n\ \textbf{when}\ g \to e,\ F) & \triangleq & \cup_i\ free(jpattern,\ F) \\
free(varId,\ F) & \triangleq & \begin{cases} \{varId\} & if F(varId) = \textsc{Udf} \\ \{\} & \text{otherwise} \end{cases} \\
free(g_1\ op_g\ g_2,\ F) & \triangleq & free(\mathrm{g}_1,\ F) \cup free(\mathrm{g}_1,\ F) \\
free(\{\overline{g}\ \},\ F) & \triangleq & \cup_i\ free(\mathrm{g}_i,\ F) \\
free([\overline{g}\ ],\ F) & \triangleq & \cup_i\ free(\mathrm{g}_i,\ F)
\end{array}
$$

apply **Definition 3.18** (Substitution)**.** Substitution is performed using the function *apply* and whenever there is a value for the variable in the given context, the value is inserted in that place.

$$
\begin{array}{rcl}
apply(\{\overline{e}\ \},\ F) & \triangleq & \forall_i apply(\mathrm{e}_i,\ F) \\
apply([\overline{e}\ ],\ F) & \triangleq & \forall_i [apply(\mathrm{e}_i,\ F)] \\
apply(varId,\ F) & \triangleq & \begin{cases} v & if F(varId) = v \neq \textsc{Udf} \\ varId & \text{otherwise} \end{cases} \\
apply(basicvalue,\ F) & \triangleq & basicvalue
\end{array}
$$

Definition 3.19 (*JErlang* guards transition)**.** The guards transition relation $\rightsquigarrow_G$ translates a pair of a guard and a frame into a guard result. Hence, it can be perceived as a very limited transition on expressions. The signature of rewriting relations is: $\rightsquigarrow_G$

$$\rightsquigarrow_G : guard \times F \rightarrow guard$$

Definition 3.20 (Operations on values in guards)**.** To allow the evaluation of the guards that contain different operations on values we introduce the evaluation function $\widehat{op_g}$ that performs execution of the given operation op_g on guards. This operation is defined similarly as operations on values in 3.15. $\widehat{op_g}$

3.2.1 Operational Semantics

To provide the succinct version of the Structural Operational Semantics of *JErlang* we will base on the Wright/Felleisen style of semantics [40]. This involves the definition of the Evaluation Contexts.

Definition 3.21 (Evaluation Context)**.** A reduction context E is an expression with a *hole* in it. Any expression e placed in such *hole* is evaluated irrespective of the context and the whole expression with the context can be evaluated iff e has be correctly evaluated. Evaluation contexts are based on the original syntax (as in 3.1): E

$$\begin{array}{rcl} \mathrm{E} & ::= & \mathrm{E}\ op\ e \mid v\ op\ \mathrm{E} \mid [\ ...,\ v_{i-1},\ \mathrm{E}_i,\ e_{i+1},\ ...\ ] \mid \{\ ...,\ v_{i-1},\ \mathrm{E}_i,\ e_{i+1},\ ...\ \} \mid \\ & & \textbf{case}\ \mathrm{E}\ \textbf{of}\ \overline{match}\ \textbf{end} \mid p =_{\mathbf{m}} \mathrm{E} \mid \mathrm{E}(\overline{e}) \mid v(...,\ v_{i-1},\ \mathrm{E}_i,\ e_{i+1},\ ...) \end{array}$$

Example 3.22 (Execution in the context)**.** Execution of E[*Test*] might look as follows:

$$\textsc{Context}: \frac{Frame(Test) = \{ok, 12\}}{\mathrm{E}[Test],\ F,\ Q \rightsquigarrow_E \mathrm{E}[\{\mathrm{ok},\ 12\}],\ F,\ Q}$$

Evaluation in contexts allows for more succinct representation. For instance **Seq**, for completeness, would typically involve writing rules:

$$\textsc{Seq}_1: \frac{e, F, Q \rightsquigarrow_E e'', F', Q'}{(e,\ e'),\ F,\ Q \rightsquigarrow_E (e'',\ e'),\ F',\ Q'} \qquad \textsc{Seq}_2: \frac{e, F, Q \rightsquigarrow_E \textbf{error}, F', Q'}{(e,\ e'),\ F,\ Q \rightsquigarrow_E \textbf{error},\ F',\ Q'}$$

Definition 3.23 (Evaluation Context for guards)**.** A reduction context $\mathrm{E}_g$is a guard with a *hole* in it. Any guard can be given for the evaluation context and is resolved irrespective of the context. This definition follows in a similar fashion to the definition of evaluation context for expressions in 3.21. E_g

$$\begin{array}{rcl} \mathrm{E}_g & ::= & \mathrm{E}_g\ op_g\ g \mid v\ op_g\ \mathrm{E} \mid [\ ...,\ v_{i-1},\ \mathrm{E}_{g_i},\ g_{i+1},\ ...\ ] \mid \{\ ...,\ v_{i-1},\ \mathrm{E}_{g_i},\ g_{i+1},\ ...\ \} \mid \\ & & \mid \mathrm{E}(\overline{e}) \mid v(...,\ v_{i-1},\ \mathrm{E}_{g_i},\ g_{i+1},\ ...) \end{array}$$

$$\textsc{Context}: \frac{e, F, Q \rightsquigarrow_E e', F', Q'}{\mathrm{E}[e],\ F,\ Q \rightsquigarrow_E \mathrm{E}[e'],\ F',\ Q'}$$

$$\textsc{Var}_0: \frac{\begin{array}{c} F(\mathbf{varId}) = v \\ v \neq \textsc{Udf} \end{array}}{\mathbf{varId},\ F,\ Q \rightsquigarrow_E v,\ F,\ Q}$$

$$\textsc{Var}_1: \frac{F(\mathbf{varId}) = \textsc{Udf}}{\mathbf{varId},\ F,\ Q \rightsquigarrow_E \mathbf{error},\ F,\ Q}$$

$$\textsc{Seq}: \frac{}{(v,\ e),\ F,\ Q \rightsquigarrow_E e,\ F,\ Q}$$

$$\textsc{Match}_1: \frac{matches(v, p, \textsc{true}, F, F')}{p =_{\mathbf{m}} v,\ F,\ Q \rightsquigarrow_E v,\ F',\ Q}$$

$$\textsc{Match}_2: \frac{\neg\exists F'\ matches(v, p, \textsc{true}, F, F')}{p =_{\mathbf{m}} v,\ F,\ Q \rightsquigarrow_E \mathbf{error},\ F,\ Q}$$

$$\text{OP}: \frac{\widehat{op}(v, v') = v''}{v\ op\ v',\ F,\ Q \rightsquigarrow_E v'',\ F,\ Q}$$

$$\text{SEND}: \frac{pid \neq self()}{pid\ \mathbf{!}\ v,\ F,\ Q \rightsquigarrow_E v,\ F,\ Q}$$

$$\text{SEND}: \frac{\begin{array}{c} pid = self() \\ Q' = Q \mathbf{.} v \end{array}}{pid\ \mathbf{!}\ v,\ F,\ Q \rightsquigarrow_E v,\ F,\ Q'}$$

$$\textsc{Case}_1: \frac{\begin{array}{l} \forall(i \in 1..n)(match_i\ =_s\ p_i\ \mathbf{when}\ g_i \rightarrow e_i) \\ j \in 1..n,\ matches\ (v, p_j, g_j, F, F') \\ \forall(1 \leq k < j).\neg\exists F''(\ matches\ (v, p_k, g_k, F, F'')\ ) \end{array}}{\mathbf{case}\ v\ \mathbf{of}\ match_1\ ...\ match_n\ \mathbf{end},\ F,\ Q \rightsquigarrow_E e_j,\ F',\ Q}$$

$$\textsc{Case}_2: \frac{\begin{array}{l} \forall(i \in 1..n)\ (match_i =_s p_i\ \mathbf{when}\ g_i \rightarrow e_i) \\ \forall(j \in 1..n)\ \neg\exists F''\ (matches\ (v, p_j, g_j, F, F')) \end{array}}{\mathbf{case}\ v\ \mathbf{of}\ match_1\ ...\ match_n\ \mathbf{end},\ F,\ Q \rightsquigarrow_E \mathbf{error},\ F,\ Q}$$

$$\textsc{Fun}_1: \frac{Module(f) = \mathbf{case}\ e\ \mathbf{of}\ \overline{match}\ \mathbf{end}}{f(\overline{v}),\ F,\ Q \rightsquigarrow_E \mathbf{case}\ \{\overline{v}\}\ \mathbf{of}\ \overline{match}\ \mathbf{end},\ F,\ Q}$$

$$\textsc{Fun}_2: \frac{Module(f) = Udf}{f(\overline{v}),\ F,\ Q \rightsquigarrow_E \mathbf{error},\ F,\ Q}$$

Figure 3.2: Structural Operational Semantics for *JErlang*

matches

Definition 3.24 (Predicate *matches*)**.** Predicate *matches* checks whether there is a frame that is consistent with the original one, in which the given *pattern* and *value* are equal. This also ensures that *pattern* contains no free variables in the context of the new frame, and the guard is satisfied:

$$matches \subseteq (value \times pattern \times guard \times F \times F)$$

$matches(value,\ pattern,\ guard,\ F, F')$ **iff**
$(Dom(F') \setminus Dom(F)) = free(pattern, F)$
$\wedge\ F \subseteq F'$
$\wedge\ value\ ==\ apply(pattern, F')$
$\wedge\ guard,\ F' \rightsquigarrow_G \text{TRUE})$

Lemma 3.25. If *matches*(*value*, *pattern*, *guard*, F, F') and *matches*(*value*, *pattern*, *guard*, F, F'') then $F' = F''$.
The proof follows from the definition of *matches*. Function *free* identifies the set of free variables for the pattern with respect to the *frame* F, Hence F' and F'' to remain consistent agree on values contained in F.
Let's assume, that F' and F'' differ on the mapping of the variables not contained in F, i.e. $\exists k, (F'(k) \neq F''(k) \wedge k \in (Dom(F') \setminus Dom(F)))$. From the definition of the function *apply*, which is deterministic, we know that it will return different values for different frames in the same pattern, i.e.
apply(*pattern*, F') = v_1, *apply*(*pattern*, F'') = v_2 and $v_1 \neq v_2$.
Therefore either $v_1 \neq v$ or $v_2 \neq v$ but this contradicts the assumption, hence $v_1 = v_2$ and F' = F''.

Definition 3.26 (Function *remove*)**.** Function *remove* when given any queue and a set of indices returns a new queue with all the elements from the original one apart from those specified in the set. *remove*

$remove\ :\ (\mathbb{NS} \times Q) \rightarrow Q$:
$remove\ ((i_1, ..., i_n),\ Q) =$
$\forall k\ (Q(k)' = Q(k) \longleftrightarrow k \notin (i_1, ..., i_n))$
$\wedge\ \forall\ (k \in i_1, ..., i_n)\ (Q'(k) = \text{UDF})$
$return\ Q'$

Definition 3.27 (Predicate *joinMatches*)**.** The role of the predicate *joinMatches* is to determine whether the given *join pattern* can be satisfied by the messages in the queue. To do this *joinMatches* checks if initial queue and frame are consistent with the ones resulting from the pattern matching. As there can be different strategies related to finding a correct pattern match, we assume that the algorithm used in *joinMatches* is independent, i.e. we check only for the correctness of the solution, not the least solution. In section 3.2.2 we further explain the different pattern-matching algorithms that our semantics could enforce. Hence for the general purpose the predicate *checkSolution* would always be true. *joinMatches*

$$joinMatches \subseteq (join \times F \times Q \times Q \times F \times Q)$$

$$\textsc{Receive}_1: \frac{\begin{array}{l} Q^a \leq_q Q \\ \forall(i \in 1..n)\ (join_i =_s jpattern_{i,1}\ \textbf{and}\ ...\ jpattern_{i,n}\ \textbf{when}\ g_i \rightarrow e_i) \\ k \in 1..n,\ joinMatches\ (join_k,\ F,\ Q,\ Q^a,\ F',\ Q') \end{array}}{\textbf{receive}\ join_1\ ...\ join_n\ \textbf{end},\ F,\ Q \rightsquigarrow_E e_k,\ F',\ Q'}$$

$$\textsc{Receive}_2: \frac{\forall(Q' \leq_q Q).\forall(1 \leq l \leq n).\neg\exists F', Q'(joinMatches(join_l,\ F,\ Q'',\ F',\ Q'))}{\textbf{receive}\ join_1\ ...\ join_n\ \textbf{end},\ F,\ Q \rightsquigarrow_E \textbf{error},\ F,\ Q}$$

Figure 3.3: Structural Operational Semantics for **receive** statement. You should note that this definition doesn't enforce any specific semantics of pattern matching. The RECEIVE$_1$ case ensures that the valid execution is based on a satisfying queue and frame. In *JErlang* we assume that whenever we reach the end of the queue in our joins solver, and we are not successful, then we have an error case.

$$\begin{array}{l} joinMatches\ ((prop_1\ p_1\ \textbf{and}\ ...\ \textbf{and}\ prop_n\ p_n\ \textbf{when}\ g\ \rightarrow\ e),\ F,\ Q_{init},\ Q,\ F',\ Q')\ \textbf{iff} \\ \quad \exists Set\ (joinMatchesAuxiliary((p_1,\ ...,\ p_n),\ F,\ Q,\ \emptyset,\ F',\ Set) \\ \quad\quad \wedge\ g, F' \rightsquigarrow_G \ \text{TRUE} \\ \quad\quad \wedge\ Q' = remove(Set,\ Q_{init}) \\ \quad\quad \wedge\ checkSolution(Set, (p_1, ..., p_n),\ g,\ F, Q)) \end{array}$$

$$joinMatchesAuxiliary \subseteq (pattern^* \times F \times Q \times \mathbb{NS} \times F \times \mathbb{NS})$$

$$\begin{array}{l} joinMatchesAuxiliary\ ((p_1,\ ...,\ p_n),\ F,\ Q,\ Set,\ F',\ Set')\ \textbf{iff} \\ \quad \exists Set'', F''\ (\\ \quad\quad \{i\} = (Set'' \setminus Set)\ \wedge\ 1 \leq i \leq |Q|\ \wedge\ i \notin Set \\ \quad\quad \wedge\ matches(Q(i),\ p_1,\ F,\ F'') \\ \quad\quad \wedge\ joinMatchesAuxiliary\ ((p_2,\ ...,p_n),\ F'',\ Q,\ Set'',\ F',\ Set') \\ joinMatchesAuxiliary\ ((p),\ F,\ Q,\ Set,\ F',\ Set')\ \textbf{iff} \\ \quad \{i\} = (Set' \setminus Set)\ \wedge 1 \leq i \leq |Q|\ \wedge\ i \notin Set \\ \quad \wedge\ matches(Q(a),\ p,\ F,\ F') \\ joinMatchesAuxiliary\ ((),\ F,\ Q,\ \emptyset,\ F, \emptyset)\ \textbf{always} \end{array}$$

The final predicate case ensures that **receive** statement without any joins is always satisfied irrespective of the queue, since we allow for such situations in our syntax.

The last fragment of the operational semantics for **receive** statement is defined in figure 3.2.1.

3.2.2 Pattern-matching algorithms

The standard transformation for the **receive** operation doesn't ensure any order in which pattern matching is resolved. We will give definitions for two intuitive strategies: *First-*

Match and *Join Priority-Match*.

Definition 3.28 (First-Match)**.** The *First-Match* strategy ensures that whenever we find a Join of patterns, say J, that is satisfied using the subset of the original mailbox then this subset is the smallest one which is able to satisfy **any** pattern in the set of joins (from the syntax in 3.1 we know that there can be many of them). Additionally with *First-Match* we ensure that J is the first join from the set that can be satisfied. *First-Match*

In other words, we perform sequential check of the original mailbox and with each extension we check for possible joins, starting from the first one. The advantage of this approach is that the implementation of the algorithm gives quite a lot of freedom to the programmers and can be reasonably efficient for most cases.

The drawback of this approach is that it can sometimes create counter-intuitive situations (see example 3.32) and we often have to bear in mind that we are dealing with mailboxes, not just the joins that we have defined. The modified operational semantics that ensures this strategy is shown in figure 3.2.2.

$$\textsc{Receive}_{First-Match}: \frac{\begin{array}{l} Q^a \leq_q Q \\ \forall(i \in 1..n)\ (join_i =_s jpattern_{i,1}\ \textbf{and}\ ...\ jpattern_{i,n}\ \textbf{when}\ g_i \rightarrow e_i) \\ k \in 1..n,\ joinMatches\ (join_k,\ F,\ Q,\ Q^a,\ F',\ Q') \\ \forall(Q^b <_q Q^a).\forall(1 \leq l \leq n).\neg\exists F'', Q'' \\ \qquad (joinMatches(join_l,\ F,\ Q,\ Q^b,\ F'',\ Q'')) \\ \forall(1 \leq l < k).\neg\exists F'', Q'' \\ \qquad (joinMatches(join_l,\ F,\ Q,\ Q^a, F'',\ Q'')) \end{array}}{\textbf{receive}\ join_1\ ...\ join_n\ \textbf{end},\ F,\ Q \rightsquigarrow_E e_k,\ F\text{'},\ Q\text{'}}$$

Figure 3.4: The transition rule for **receive** in *First-Match* semantics for joins solver.

Definition 3.29 (Ordering on the sets of indices)**.** We define ordering between the sets of indices in a style similar to an alphabetical ordering for words, i.e. we compare indices starting from the left-most element of the sets and the first element that differs determines the ordering: $<_{idx}$

$Set <_{idx} Set' \Rightarrow Set \neq Set' \ \wedge$
$(i_1, ..., i_n) <_{idx} (i_1', ..., i_n') \ \longleftrightarrow \forall\ (k \in 1..n)\ (i_k \leq i_k' \ \vee\ \exists l\ (1 \leq l < k \ \wedge\ i_l < i_l'))$

Example 3.30 (Ordering $<_{idx}$ on the sets of indices)**.** $\{1,3,2\} <_{idx} \{1,4,1\}$ and $\{2,3,2\} \not<_{idx} \{1,4,1\}$

Definition 3.31 (Predicate *checkSolution*)**.** Given a set of indices *Set* and a join, predicate *checkSolution* succeeds if the given set of indices is the smallest one (in terms of $<_{idx}$ ordering), such that corresponding messages in the queue satisfy the join in the context of frame F. *checkSolution*

$$\textsc{Receive}_{Priority-Match}: \frac{\begin{array}{l}\forall(i \in 1..n) \\ \quad (join_i =_s jpattern_{i,1} \textbf{ and } ... \ jpattern_{i,n} \textbf{ when } g_i \rightarrow e_i) \\ k \in 1..n,\ joinMatches\ (join_k,\ F,\ Q,\ Q,\ F',\ Q') \\ \forall(1 \leq l < k).\neg\exists F'', Q'' \\ \quad (joinMatches(join_l,\ F,\ Q,\ Q, F'',\ Q''))\end{array}}{\textbf{receive}\ join_1\ ...\ join_n,\ F,\ Q \rightsquigarrow_E e_k,\ F',\ Q'}$$

Figure 3.5: The transition rule for **receive** in *Join Priority-Match* semantics for the joins solver. The subset of the original queue no longer determines the order of execution.

$$checkSolution \subseteq (NS \times pattern^* \times guard \times F \times Q)$$

$$\begin{array}{l} checkSolution\ (Set, (p_1,\ ...,\ p_n),\ guard,\ F,\ Q)\ \textbf{iff} \\ \forall Set'(Set' \neq Set \wedge\ \exists F'(joinMatchesAuxiliary((p_1,\ ...,\ p_n),\ F,\ Q,\ \emptyset,\ F',\ Set') \\ \qquad \wedge\ g, F' \rightsquigarrow_G\ \text{TRUE})\ \rightarrow Set <_{seq} Set') \end{array}$$

Example 3.32 (First-Match joins matching)**.** For the listing 3.4 and mailbox $Q = (\text{foo} \cdot \{\text{error, function\_clause}\} \cdot \{\text{ok, bar}\})$
the *First-Match* strategy will execute the second join since the queue of size 2 already satisfies the **receive** statement. Such a case would be undesirable if we have a high-priority message sent after the low-priority ones, since it is not possible to get the former first.

```
1 receive
2     {ok, Test} ->
3         %% ...
4     Value1 and {error, Reason} ->
5         %% ...
6 end
```

Listing 3.4: Example of First-Match joins matching

Standard *Erlang* semantics uses algorithms consistent with the *First-Match* and since only single-pattern joins are allowed in the **receive** statements, the situation presented in example 3.32 does not occur.

Join Priority-Match

Definition 3.33 (Join Priority-Match)**.** *Join Priority-Match* is focused on the syntactical definition of the joins and therefore the context of the mailbox doesn't influence the order in which pattern-matching is resolved. The main disadvantage of this approach is the efficient implementation in the concurrent world, where messages can be received independently of the join-matching procedure.
The difference in semantics for the *Join Priority-Match* algorithm is presented by the semantics in figure 3.2.2 and example 3.34.

Example 3.34 (Priority-Match joins matching)**.** For the same example 3.32 and mailbox Q = (foo · {error, function_clause} · {ok, bar}) the *Join Priority-Match* strategy will successfully match the first join. Even though the message {ok, bar} is the last one, it satisfies the pattern in the first join and fires the execution.
We can see how typical priority receiving of messages can be implemented with this semantics, therefore being more intuitive for real examples.

The *Join Priority-Match* semantics in *JErlang* raises the question about which queue we should pattern-match against. The introduction of *snapshots* of the queue invalidates the semantics presented in figure 3.2.2, whereas for example locking the queue while running the pattern-match solver defeats the purpose of *JErlang* as a highly concurrent language.

Definition 3.35 (Structural Operational Semantics for Guards)**.** Semantic rules for the guards, as presented in 3.35, underline the fact that guards are a limited derivative of expressions. Nevertheless their existence improves the expressiveness of the language.

$$\textsc{ContextGuard}: \frac{g, F \rightsquigarrow_G g', F'}{\mathrm{E}[g],\ F \rightsquigarrow_G \mathrm{E}[g']}$$

$$\textsc{VarGuard}_0: \frac{F(\mathbf{varId}) = v \quad v \neq \textsc{Udf}}{\mathbf{varId},\ F \rightsquigarrow_G v}$$

$$\textsc{VarGuard}_1: \frac{F(\mathbf{varId}) = \textsc{Udf}}{\mathbf{varId},\ F \rightsquigarrow_G \mathbf{error}}$$

$$\textsc{OPGuard}: \frac{\widehat{op_g}(v, v') = v''}{v\ \ op_g\ \ v',\ F \rightsquigarrow_G\ v"}$$

Figure 3.6: The semantic rules for guards are very similar to the rules defined for expressions. The main difference is the introduction of the new transition relation.

3.3 Conclusion

The contents of this chapter is devoted to the formalisation of the minimal *JErlang* language. In our definition we focus on the semantics of pattern-matching in the context of joins, therefore omit the details realted to the existencce of processes or fault tolerant systems. The behaviour of the language is given in terms of *Small Step Semantics*, due to the concurrent nature of *JErlang*. Our work is concluded by the definition of a general joins solving algorithm, which is then extended by the concrete semantics of two popular algorithms that exhibit different guarantees towards order of the message sequences.

Chapter 4

The language

Given the semantics from chapter 3 we will now present the features that can be used by *JErlang* developers. The chapter can be used as a reference guide with many examples illustrating the joins constructs and their variations. We will try to show that the change from *Erlang* to *JErlang* comes very naturally as we were very conservative when it comes to altering the standard behaviour of the language. The first section explains the possible advantages of having different types of synchronisation constructs, followed by an enumeration of the language features. We conclude by describing the important new *Open Telecom Platform* behaviour.

4.1 Joins for Mailboxes

Depending on the architecture of the original language, we distinguish between two main streams of *Join-calculus*, namely concurrent and distributed synchronisation on join patterns. In the following section we present the differences between the two approaches and the decision that convinced us to follow the former path.

4.1.1 Joins for Multiple Mailboxes

Erlang is a highly concurrent language which supports communication through the simple concept of mailboxes. All messages are sent asynchronously and enqueued directly in the process' mailbox, so the communication is not dependent upon the current state of the receiver. Unfortunately each process contains only a single mailbox and it is desireable to have the possibility of sending different types of messages to the process through different channels.

Popular Join implementations like *Polyphonic C#* or *JoCaml* use the concept of channels, which in reality work similarly to *Erlang*'s mailboxes. In the latter however they server as independent entities and it is possible to synchronise between the values. This possibility

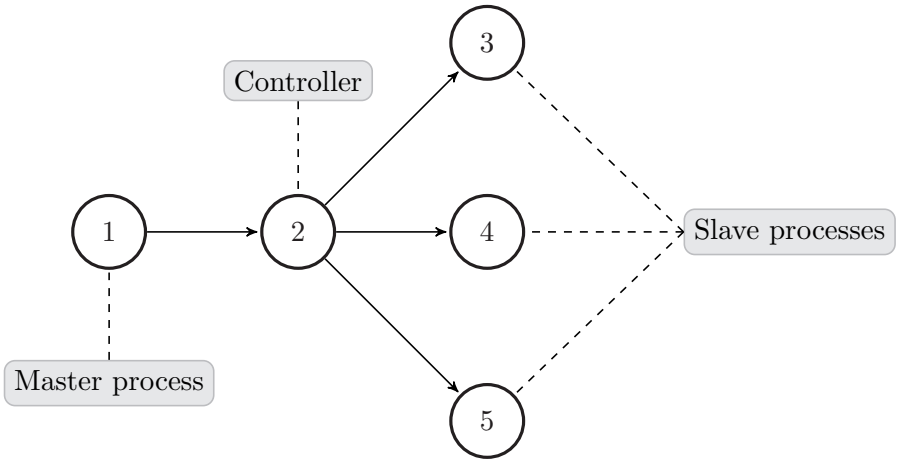

Figure 4.1: Architecture for joins on processes having multiple mailboxes

is clearly prohibited in the current *Erlang*implementation, so our initial attempt at *JErlang* involved developing support for multiple name-space addressing for the processes and distributed synchronisation on messages. A typical architecture for a single *JErlang*'s process is presented on figure 4.1.

It is easy to see that support for even a small number of channels involves a large number of new processes, the sole role of which is to serve as buffers (assuming that we do not want to fundamentally change the internals of *Erlang*'s VM). That itself is not a problem because *JErlang* processes are very lightweight and the built-in support for fault-tolerant systems enables to build a net of processes and dependencies between them through the supervision trees. As a result we ended up with a hierarchy of processes: *Master* (original process), *Controller* and multiple *workers*. The distinction between the first two allows for a clear interface to the whole interaction and separation of tasks.

In a typical schema a message will be sent either directly to the specific mailbox, or to the *Controller* process that dispatches the messages to the appropriate queue. In the latter case the *Controller* is likely to become the bottleneck of the system due to the possible number of external as well as internal messages. The main joins mechanism that lives in the *Controller* process is enabled on the **receive** construct by the *Master* process, and communicates with all the subprocesses in order to:

- Retrieve the messages necessary for pattern-matching.
- Notify the *worker* process whether pattern-matching is successful or not.
- Ensure the correct synchronization between messages, assuming that determinism is important.

The first attempt at implementing multiple-channels per process was successful in that we were able to run a few simple test-cases that synchronised on messages. Nevertheless building a system based on the above architecture has shown us some of the drawbacks:

- We were unable to use the built-in mailbox *Erlang* support, since to know the contents of the message, you have to first read it, and hence remove it from the queue. *Erlang* does not provide any way to re-insert the value without destroying the initial ordering, or even "peeking" at the contents of the queue[1].

- Lower network load can be achieved by providing the middleware buffers. Having a way to identify the messages doesn't suffice because retrieving multiple messages from the mailboxes involves multiple synchronous calls. Storing the result in multiple locations also doesn't help as we end up creating "God" processes having large number of copies. We developed an architecture in which the burden of a small number of messages was unreasonably high and hence would not scale-up to even hundreds of messages per queue, which is a typical scenario for *Erlang* systems.

- Since messages can be received independently by multiple *workers* we are unable to determine any kind of ordering between the independent channels. Given, for example three channels as presented in example 4.1, the only way to introduce ordering would be for the *Controller* to provide unique timestamps (assuming that messages are passed through it), and then perform pattern-matching that takes these into account. Given the complexity that is involved in finding the correct join's messages (see later in 5.4) we decided against the introduction of another constraint.

Based on diagram 4.1, a general scenario for the processing of **receive** construct with joins would be:

- Notification from the *Master* to the *Controller* about the requested joins, represented through partial patterns in an easy to analyse form.

- The *Controller* passes those pattern tests to the respective mailboxes (slave processes).

- Channels independently send the partial matching messages to perform global matching on the whole join (assuming that we want to preserve reasonable semantics as in 4.2).

- The *Controller* keeps receiving matching messages from channels and runs the joins solver to find the correct sequence of messages.

- The *Controller* notifies the required channels that matching is finished (or timeout was hit), though its message may take some time until it is processed (knowing that the mailbox may still contain standard messages) and hence introduces an unnecessary increase in the computation and the load on the network.

[1]There is in fact a technique for getting a rough look at the queue contents, but this technique becomes unusable for large systems

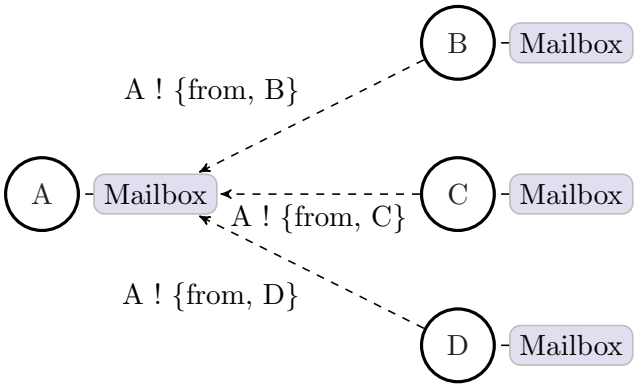

Figure 4.2: Sending messages and synchronisation architecture in a single mailbox in *JErlang*

- The *Controller* notifies which messages should be removed and returns the result to the *Master*

We found the architecture for multiple queues to be a reasonable choice due to its intuitive meaning, its range of possibilities (having separate messages for system, debug and normal messages) and similar constructs presented in *Polyphonic C#*, *HaskellJoinRules*or *JoCaml*.

Nevertheless we made a decision to abandon this idea not only because of the implementation problems. *Erlang*'s nature of a single mailbox is deeply rooted in it by the *Erlang*architects who, reasonably, proclaim that having multiple queues unnecessary complicates the system in its entirety, which already at the current stage is hard to manage. Some of the implementations support distributed joins semantics, yet they are not restricted by the *no memory sharing* principle of *Erlang* and can directly access the messages stored inside local channels. Further discussion is continued in the evaluation section 6.2.2.

```
1  receive
2      Q1:{message_01_q2, A} and Q2:{message_02, _B} ->
3          io:format("First join", []);
4      Q1:{message_01_q2, A} and Q3:{message_02, B} ->
5          io:format("Second join with values [~p", [])
6  end.
```

Listing 4.1: Receiving messages for multiple queues architecture in *JErlang*

4.1.2 Joins for a Single Mailbox

The second, theoretically more conservative approach for joins support involves synchronisation only on the messages within the single mailbox. Hence from the point of view of the architecture *JErlang* remains the same as in the original *Erlang* (see figure 4.2).

We present an example on why synchronisation on two messages within the same mailbox is often problematic.
Given the mailbox: (**{get, 1}** · **{set, 4}** · **{set, 2}** · **{get, 2}**)

we would like to synchronize on messages that match the patterns **{get, A}** and **{set, B}**, where A and B are equal. Clearly such a combination is possible in our message queue.

An intuitive implementation under *Erlang* is presented in listing 4.2. Since standard *Erlang* follows the *First-Match* strategy when retrieving messages from the mailbox, the code will match with the first rule in the first **receive** and it will get stuck since there is no value **{set, 1}**. Even when the order of patterns is reverted, we first match **{set, 4}** and fail when trying to find the match for **{get, 4}**. This example illustrates a common problem in *Erlang*, where we are not allowed to rely on the ordering of the messages inside the queue. This forces the programmers to consider the possible layouts of the mailbox and how the processes would need to interact. We believe that pressing such constraints on the programmer is undesirable.

Listing 4.3 presents a refined, working code, where we continuously fetch messages from the queue and at the stage when no synchronisation can be fulfilled with the first message, we resend it and run the script until successful. Although this code does perform the expected behaviour, we no longer can ensure that the ordering of the messages in the queue is preserved as **send**[2] operation enqueues the value to the end. As we mentioned in 2.3 messages sent from the same process are guaranteed to be received in the original order. With the re-send semantics we may have a scenario when an **abort** message is run only after the **commit** instruction, instead of the other way around. Not to mention the fact that solution represented in listing 4.3 is error prone and hard to understand. Similar ideas are discussed in [35], but no working *Erlang* prototype is presented there.

In order to abstract from the current state of the mailbox, *JErlang* provides synchronisation semantics with **receive**-like join construct. Using the guards in example 4.4 we end up with the most intuitive (and correct) solution and reduced number of lines from 19 to 3. Execution of the selective **receive** matches the necessary messages separately and then performs a full join test to check whether guards and variables as a complete set are actually satisfiable. The solution to the problem can be even shorter in *JErlang* using non-linear pattern matching (see section 4.2.6).

[2] In *Erlang* used by sign**!**

```
1  fun() ->
2      receive
3          {get, X} ->
4              receive
5                  {set, Y} when (X == Y) ->
6                      %% execute expression
7                      {found, X}
8              end;
9          {set, X} ->
10             receive
11                 {get, Y} when (X == Y) ->
12                     %% execute expression
13                     {found, X}
14
15     end
16 end.
```

Listing 4.2: Error prone synchronisation on two messages in *Erlang*

```
1  fun() ->
2      A = fun(ReceiveFunc) ->
3              receive
4                  {get, X} ->
5                      receive
6                          {set, Y} when (X == Y) ->
7                              {found, X}          %% execute expression
8                      after 0 ->
9                          self() ! {get, X},
10                         ReceiveFunc(ReceiveFunc)
11                     end
12                 {set, X} ->
13                     receive
14                         {get, Y} when (X == Y) ->
15                             {found, X}          %% execute expression
16                     after 0 ->
17                         self() ! {set, X},
18                         ReceiveFunc(ReceiveFunc)
19                     end
20             end
21         end,
22     A(A)
23 end.
```

Listing 4.3: Correct synchronisation for two messagesin *Erlang*

```
1  fun() ->
2      receive
3          {get, X} and {set, Y} when (X == Y) ->
4              {found, X}          %% execute expression
5      end
6  end
```

Listing 4.4: Synchronisation on two messages with guards in *JErlang*

4.2 Language features

4.2.1 Getting started

In order to minimise the amount of effort a programmer has to put in using *JErlang* we introduce a transformation module that provides different semantics to *Erlang* syntax. This way the programmer focuses on solving the problems rather than reading the complex API of the library and calling the functions in an awkward way. In order to notify the compiler about our transformation, we introduce a following single line at the top of the module definition:

```
1 -compile({parse_transform, jerlang_parse}).
```

Obviously the library has to be placed somewhere along the *Erlang* search path, so that the compiler and VM can find the necessary modules. We give a detailed description of the actions performed during transformation in implementation section 5.3. In order to use *JErlang* with changed VM, one obviously needs a patched R12B-5 version of *Erlang*'s VM and the transformation module is instead:

```
1 -compile({parse_transform, jerlang_vm_parse}).
```

Both version are binary incompatible, therefore it is important to stick one version or use *Erlang*'s macro definitions as we present in many provided examples[3].

4.2.2 Joins

The most visible feature of *JErlang* are obviously joins, i.e. the ability to match on more than a single message in a single simple construct. Listing 4.5 presents a function that retrieves from the mailbox two numbers, which are then either added, multiplied or subtracted. In an artificial situation three processes might cooperate in the production of a single value. This example follows a typical client-server architecture and due to the inherent concurrency, it is valid for the messages to arrive in any order.

```
1  operation() ->
2      receive
3          {ok, sum} and {val, X} and {val, Y} ->
4              {sum, X + Y};
5          {ok, mult} and {val, X} and {val, Y}  ->
6              {mult, X * Y};
7          {ok, sub} and {val, X} and {val, Y} ->
8              {sub, X - Y}
9      end
10 end.
```

Listing 4.5: Sychronisatin in *JErlang* to perform simple arithmetic operations on messages

[3]To avoid spurious errors we provide typical Makefile. This will automatically detect the current version of the run-time. In case of mysterious errors while switching between two modes use *make clean; make*

```
1  self() ! {foo, one},                    %% Message 1
2  self() ! {error, function_clause},      %% Message 2
3  receive
4      {foo, A} and {foo, B} ->
5          {error, invalid_join};
6      {error, Reason} ->
7          {ok, {error_expected, Reason}}
8  end.
```

Listing 4.6: Joins with pattern that can match on the same messages

Listing 4.6 presents the case where the first message matches both of the patterns in the first join, namely **{foo, A}** and **{foo, B}**. However the implementation of *JErlang* ensures that messages satisfying the join have to be unique, i.e. one message cannot satisfy more than a single pattern in a successful match. Hence, the end result of calling example 4.6 would be the second tuple. Most of the existing implementations forbid the the definition of channels that share the name in the same join. This is dictated by the implementation complexity as solving such joins is non-trivial (as we discuss in 5.4). Nevertheless, lack of this feature in *JErlang* would be a serious disadvantage since having the possibility of building complex pattern matches in a simple way in *Erlang* is one of the fundamental features of the language. It is common to build matches the variable of which span over more than a single pattern (for instance matching elements of a list **[A, A]**).

As expected, the *JErlang* implementation preserves the notion of bounded variables and the execution of the example 4.7 would get stuck, provided that only the first two messages are in the mailbox. This is because the variable **C** is already mapped to **none** and matching of **{bar, C}** fails when compared with **one**.

```
1  self() ! {foo, one},                    %% Message 1
2  self() ! {bar, one},                    %% Message 2
3  C = none,
4  receive
5      {foo, A} and {bar, C} ->
6          {error, invalid_join}
7  end.
```

Listing 4.7: Joins matching with bounded variables in *JErlang*

4.2.3 Order preservation in mailbox and First-Match execution

Implementations of the *Join-calculus*in languages like *JoCaml*(2.2.1) or *Polyphonic C#*(2.2.2) do not bind themselves to any strictly specified algorithm when it comes to solving the joins' patterns. In other words when there is more than one join that is satisfiable by the current state of the channels/queues then the result of the join operation is non-deterministic. The situation of *Polyphonic C#* is easier since we are not allowed to have a single channel name in more than one chord, whereas in *JErlang* we can easily have a situation (given appropriately liberal patterns) where a message could match all the patterns.

In the implementation of **receive** in *JErlang* we assume that the semantics of *First-Match*, as defined formally in 3.2.2, allow for more predictable behaviour when it comes to

the resolution of the joins. Implementing the semantics of *Join Priority-Match* is inherently impractical in *JErlang* due to the concurrent nature of the language.

Example 4.8 represents a confusing situation where the result of the joins depends on the ordering of the messages. *JErlang*'s priority is to preserve the original sequence of the messages during each pattern-matching attempt. This idea drives the execution of the join-solver mechanism that takes into account the order of the messages. Therefore by looking only at the first two messages of the mailbox presented in the example (assuming no other messages to the process apart from those specified) we can see that the second join is satisfied already by the second message, whereas the first join at this stage still misses one more successful pattern. Programmers do not know the contents of the mailbox (and shouldn't know or care about it in any implementation) but the successful firing of the action provides information that the join was the first to happen given the concurrent environment (affected by delays in sending the messages for example).
By ensuring this specific behaviour in our implementation we believe that developers have more control over what scenarios could actually happen. Non-deterministic behaviour hides all the details but also gives less control in a situation when one would like to have it. Listing 4.9 presents a typical *Erlang*-like situation where we want to make sure that the second join will fire if and only if the first join failed to match.

```
1  self() ! {foo, one},                    %% Message 1
2  self() ! {error, function_clause},      %% Message 2
3  self() ! {bar, two},                    %% Message 3
4  receive
5      {foo, A} and {bar, B} ->
6          {error, {A, B}};
7      {error, Reason}  ->
8          {ok, {error_expected, Reason}}
9  end.
```

Listing 4.8: Oddity Simple mathematical operations on messages

```
1  receive
2      {foo, A} and {bar, specific} ->
3          %% perform specialized action...
4      {foo, A} and {bar, B} ->
5          %% perform general action...
6  end.
```

Listing 4.9: Usage of deterministic behaviour for joins resolution in *JErlang*

4.2.4 Guards

A guard is a feature typical for functional languages, which allows the programmer to include additional constraints for popular constructs, like function definitions or case statements. Guards in *Erlang*'s **receive** allow to check the bounded value or compare variables defined in the message.

The original definition of Join-calculus in [15] does not mention guards at all, but we believe there are possible advantages to using them, which many other implementations

have not attempted. Lam and Sulzmann in [21] provide successful scenarios where guards are the most intuitive features to use.

In *JErlang* guards enable the programmers to filter the incoming messages depending on the success or failure of the computation. Since developers are familiar with it, it seemed reasonable to also have guards that would be able to operate on the variables that span over more than a single pattern. In listing 4.10 we use the construct to perform sanity checks for the withdrawal transaction that was requested by the external user.

```
1  withdraw(Transaction) ->
2      receive
3          {amount, Transaction, Money} and {limit, LowerLimit, UpperLimit}
4                      when (Money < UpperLimit and Money > LowerLimit) ->
5              commit_withdrawal(Money, Limit);
6          {abort, Trans} and {amount, Transaction, Money}
7                      when (Trans == Transaction) ->
8              abort_withdrawal(Transaction, Money)
9      end
10 end.
```

Listing 4.10: Usage of guards for withdrawal operation on the bank account in *JErlang*

As the mailbox can be operating on multiple accounts and transactions, guards in joins allow us to avoid unnecessary removal of the messages from the transaction's queue. Programmers do not have to focus on the way to express the problem that is a valid *JErlang*'s syntax and semantics, but rather focus on the problem itself.

4.2.5 Timeouts

In order to avoid potential deadlocks, similarly to example 4.2, where the execution of **receive** gets stuck while analysing the mailbox of the process, *JErlang* introduces timeouts. A timeout is measured as a period when the Join-solver doesn't perform any useful procedure, i.e. it stalls waiting for new messages, since all the previous combinations of pattern matching failed. Therefore time spent on performing the actual matching does not count towards the timeout value (as in *Erlang*). None of the existing implementations provide this feature.

```
1  withdraw_using_card(Timeout) ->
2      receive
3          {pin, Pin} and {card, Number} ->
4              authenticate(Pin, Number);
5          abort and {card, Number} ->
6              abort_authentication(Number)
7      after Timeout ->
8          return_card(Number, Timeout);
9      end
10 end.
```

Listing 4.11: Timeout execution no response in *JErlang*

Listing 4.11 presents an example when a person wants to withdraw money using debit card in a cash-machine and we want to make sure that the card does not in it stay forever. To specify a timeout, developers have to follow the syntax rules presented in 4.12.

```
1     receive
2         Joins
3     after Value ->
4         TimeoutAction
5     end
```

Listing 4.12: Standard schema for constructing timeouts in *JErlang*

If a new message is sent during the idle period, then new join matching process is started. The timeout counter is turned off during that time and if matching fails then the counter continues with its last value. Therefore the standard **receive** construct without timeout value is equivalent to the syntax presented in 4.14.

```
1     receive
2         Joins
3     after infinity ->
4         TimeoutAction    %% Never executed
5     end
```

Listing 4.13: Full receive syntax with timeouts

A typical usage of timeouts would be to ensure that by the time joins matching is performed, all the necessary components are available. In order to force this semantics, the programmer defines an action on timeout 0.
It is important to note that timeout should not be used to reliably determine the time passage, since measuring of the counter can be influenced by the external factors like process priority, operating system scheduling, internal *JErlang* scheduling, random execution etc. The only difference between the timeouts in *Erlang* and *JErlang* is the way the counter is defined - the former is managed inside the *Erlang*'s VM and the latter manages them by cooperating between the Joins library and *JErlang*'s VM. *JErlang*'s timeouts ensure that there is no loss of information and any message received at the point when the timeout alarm triggers would still be available in the mailbox.

4.2.6 Non-linear patterns

Unlike all of the *Join-calculus*'s implementations *JErlang* allows for non-linear patterns. In other words patterns in joins can contain the same unbounded variables and therefore synchronise on their values as well, since we know that a single variable can have only a single value associated with it. As a result, the example presented in 4.4 could be even shorter and more intuitive as we present in listing 4.14.

```
1 fun() ->
2     receive
3         {get, X} and {set, X} ->
4             %% execute expression
5     end
6 end
```

Listing 4.14: Synchronisation for two messages in *JErlang* where they agree on the value of

the unbounded variable

We believe that the ability to express this syntax increases the freedom of the programmers as they are less constrained by the awkward limitations of the language. The construct should gain the approval from the *Erlang*'s community as it makes the code more readable and easier to maintain. Example 4.15 presents a more complicated case when synchronisation on patterns and variables inside them happens more than once. We define a scenario where two independent entities running in different processes want to establish a mutual transaction with agreement that both continue if and only if both agree to the transaction. However if at least one of the entities forbids the transaction, then it is aborted (first join). The synchronization is made by the the third-party-process that has no knowledge of the transaction beforehand and therefore runs as a general synchronising entity. This is also a typical variant of the barrier synchronization pattern.

```
1  fun() ->
2      receive
3          {transaction, Id, IdA, IdB} and {disagree, Id, IdA} ->
4              notify_failure(Id, IdB),
5              abort_transaction(Id, IdA, IdB);
6          {transaction, Id, IdA, IdB} and {agree, Id, IdA} and {sth, Id, IdB} ->
7              notify(Id, IdA),
8              notify(Id, IdB),
9              commit_transaction(Id, IdA, IdB)
10     end
11 end
```

Listing 4.15: Mutual agreement using non-linear join patterns in *JErlang*

4.2.7 Propagation

Propagation is another feature not known in the *Join-calculus* world but introduced successfully in their *Constraint Handling Rules*. Lam and Sulzmann in [37] and [35] show its usefulness in their Haskell prototype.

In *JErlang* propagation can be treated as an optional attribute that allows the developer to say that whenever a pattern matches, the value of the message is returned to the user but the message itself is not removed from the mailbox. This obviously can be implemented by sending the same message in the body of the join[4], however our feature is more readable and less error prone. In order to mark the pattern to support the propagation schema we wrap it with the **prop** closure, as if it was an argument to the function.

Additionally whenever the search is performed on the mailbox again, the message will not be placed at the end of the queue and as a result will get higher priority and the implementation will ensure that matching should be performed faster (as we do not have to reach the end of the queue). This is especially important for applications which allow for large numbers of messages in the queue and hence for longer response time.

Listing 4.16 presents an authorisation procedure where the usage of propagation is natural. In order to perform operation on the process, some previous session establishment has

[4] see section 5.4.4

to occur because by default the sender has no privileges. Whenever the matching for the first join is successful, the message satisfying the first pattern remains in the mailbox i.e. the session remains alive, as one would expect for security reasons. However, for the second join, we want to end the user session, so the **session** message is removed along with the **logout** message and the user can no longer perform any action (provided that the mailbox does not contain more matching **session** messages).

```
1 authorisation() ->
2     receive
3         prop({session, Id}) and {action, Action, Id} ->
4             perform_action(Action, Id);
5         {session, Id} and {logout, Id} ->
6             logout_user(Id)
7     end
8 end
```

Listing 4.16: Session support using propagation in *JErlang*

4.2.8 Synchronous calls

JErlang, similarly as the original *Erlang*, allows for the creation of synchronous calls (sending is normally asynchronous). This can be achieved by appending a *Pid* (process identifier) value to the message, so that the process knows with whom it communicates. Unlike the *Polyphonic C#* or *Join Java* implementations, *JErlang* does not enforce any kind of ordering or style of the patterns, nor does it limit the number of possible synchronous calls[5]. We decided not to introduce separate syntax construct for synchronous calls, but rather use the possibilities that *JErlang* already has using the values provided in the message. The introduction of any formalised style would rather limit the freedom of the programmers than increase it.

In example 4.17 we present a variant of the typical barrier synchronisation construct. The listing presents a situation for the two cases of barriers that can occur. Obviously the naming for the variables and structure of the messages is completely random - the only requirement is that messages at agreed tuple elements store their process identifier (in this case **Pid1**, **Pid2**...). In *Erlang* the synchronisation between multiple processes requires additional, in our view superfluous effort from the programmer as one has to first receive the message, store its contents and perform any necessary management/matching necessary to identify correct links between the messages. This is obviously error prone and overcomplicates the development of this relatively simple operation.

The example provides possible definitions of the processes participating in the synchronisation. In the first join the participating processes are running the function **process_func_barrier** - they first send the initial (asynchronous) message to the known central process and await the reply, thus creating a simple and intuitive send-wait pair. Processes participating in the second join execute the function **process_func_barrier_value**, with the exception of the process that sends the middle message. The latter does not participate di-

[5] *Scala* by default inserts the *Pid* of the sender to each message. We believe that such an overhead is not necessary

rectly in the synchronisation since it is only an asynchronous call. This shows the possibilities that occur when mixing synchronous and asynchronous calls in *JErlang*.

```
1  barrier() ->
2      receive
3          {sync, Name1, Pid1} and {sync, Name2, Pid2} ->
4              notify_sync(Name2, Pid1),
5              notify_sync(Name1, Pid2);
6          {accept, Pid1} and {asynchronous, Value} and {accept, Pid2} ->
7              send_value(Pid1, Value),
8              send_value(Pid2, Value)
9      end
10 end.
11
12 notify_sync(Name, Pid) ->
13     Pid ! {barrier, Name, Pid}.
14
15 send_value(Pid, Value) ->
16     Pid ! {barrier, value, Value}.
17
18 process_func_barrier(Id, MasterPid, Timeout) ->
19     MasterPid ! {sync, Id, self()},
20     receive
21         {barrier, PartnerName, PartnerPid} ->
22             success_barrier(PartnerName, PartnerPid)
23     after Timeout ->
24         fail_barrier(timeout, MasterPid)
25     end.
26
27 process_func_barrier_value(MasterPid, Timeout) ->
28     MasterPid ! {accept, self()},
29     receive
30         {barrier, value, Value} ->
31             process_value(Value)
32     after Timeout ->
33         fail_barrier(timeout, MasterPid)
34     end.
```

Listing 4.17: Variations of the barrier synchronisation pattern in *JErlang*

4.2.9 OTP platform support in JErlang

We believe that the implementation of *JErlang* wouldn't be successful without acceptance of the standards and practices typical for *Erlang* users. To provide full conformance with the *Erlang* programming world we decided to implement an extension of **gen_server**, a popular design pattern inter alia used for building complex client-server applications.
gen_joins is a natural extension of the OTP's **gen_server** design pattern that allows for the definition of joins, i.e. for synchronisation on multiple synchronous and asynchronous messages. Additionally, features like guards and propagation that are present in **receive** construct are also allowed in this implementation. Appendix C gives two examples that

we believe express the ideas in a more natural way than if they were implemented using **gen_server**.

Getting started

Different behaviour of the **gen_joins** design pattern enforces us to use different transformation module from the one given in 4.2.1. This again removes all the burden of creating required function calls from the end developer, who only has to insert the following line to us familiar *JErlang* syntax:

```
1 -compile({parse_transform, jerlang_gen_joins_parse}).
```

This informs the compiler that perform additional stage during the compilation that will produce a vald *Erlang* code. Thus again *JErlang* can be treated as an optional library.

Synchronous and asynchronous calls

gen_joins allows for synchronisation on multiple synchronous and asynchronous messages in a single join, similarly as **receive**. The difference is that in the former we have to explicitly determine if the process receiving the message needs to send the reply or not.

```
1 define(Key, Value) ->
2     jerlang_gen_joins:cast(?MODULE, {define, self(), Key, Value}).
3
4 retrieve(Key) ->
5     jerlang_gen_joins:call(?MODULE, {retrieve, self(), Key}).
6
7 handle_join( {define, PidSource, Key, Value} and {retrieve, PidTarget, Key},
8                 Status) ->
9     {[noreply, {reply, {ok, Value}}], [{Key, PidSource, PidTarget} | Status]}.
```

Listing 4.18: Distributed buffer implemented in **gen_joins** with support for audit

Listing 4.18 defines the *JErlang* program that has synchronous and asynchronous messages in the join. We follow the standard set in *Erlang*, where the former is represented by execution of the **call** and the latter by **cast**. Asynchronous messages should always return **noreply** as there is no sender awaiting a reply. The server will throw an error on a different reply type. Synchronous messages can either return **{reply, Message}** [6] or **noreply**. The latter would cause a timeout exception on the caller's side or the caller will stall forever if the timeout is set to infinity. Our implementation conforms with OTP's design of **gen_server**, where developers also have to explicitly define the action for the message received by the process.
In line 9 of the example we return explicitly the value contained in the asynchronous message sent to the process

[6] "Message" denotes any value that is to be returned to the sender of the message

4.3 Conclusions

This chapter presented an argument in favor of supporting single mailbox joins in *JErlang*. We believe that such implementation that lack of such feature is of primary concern rather than on focusing on the synchronisation of multiple distributed mailboxes.

We give definition for more powerful **receive** construct that allows for synchronisation on messages with *First-Match* semantics. As a result we gain proper ordering in the execution of the joins and can build intuitive pattern matching structures that use this features. The expressiveness is improved through the additional features that come with *JErlang*'s joins: timeouts, guards, propagation and non-linear patterns. In comparison to other implementations we do not introduce any constraints on the mentioned constructs thus leaving more freedom to the programmers. To comply with the *Erlang* standard we provide the extension of the **gen_server**behaviour that allows on the synchronisation on multiple synchronous and asynchronous calls in a compact design pattern. The latter reminds of the *chord* concept in Object Oriented Programming but with more powerful options.

Chapter 5

Implementation

The number of current *JoCaml* implementations set a high quality mark in terms of usability and performance of joins. Since the set of features that *JErlang* offers is radically different (and we believe more demanding) we had to consider numerous possible algorithms, enhancements rather than just translating the existing solutions into *Erlang*. In this chapter we present most of the challenges that we encountered during the implementation stage of our work. Some of the constraints that had to be taken into account during our research are:

- Consistency with existing *Erlang* implementations
- Determinism and fairness of the join solver
- Performance
- Degree of expressiveness
- Scalability
- Support for *Erlang*'s Open Telecom Platform
- Intuitiveness of the syntax

Typically there are two ways of extending a language: by providing a self-contained library or changing the compiler/VM to incorporate new constructs. The former allows the *Erlang* programmers to experiment with the language without any major changes to the current setup. The latter, much harder due to the complexity of the implementation and diversity of the used tools, gives a better feeling that the new features could be included in the language. Therefore apart from developing the *JErlang* as a library we decided to provide efficient and non-intrusive[1] VM and compiler changes. Another motivation for this decision

[1]We are aware that such a combination of words sounds like an oxymoron, but we believe we managed to create a reasonable compromise between the two

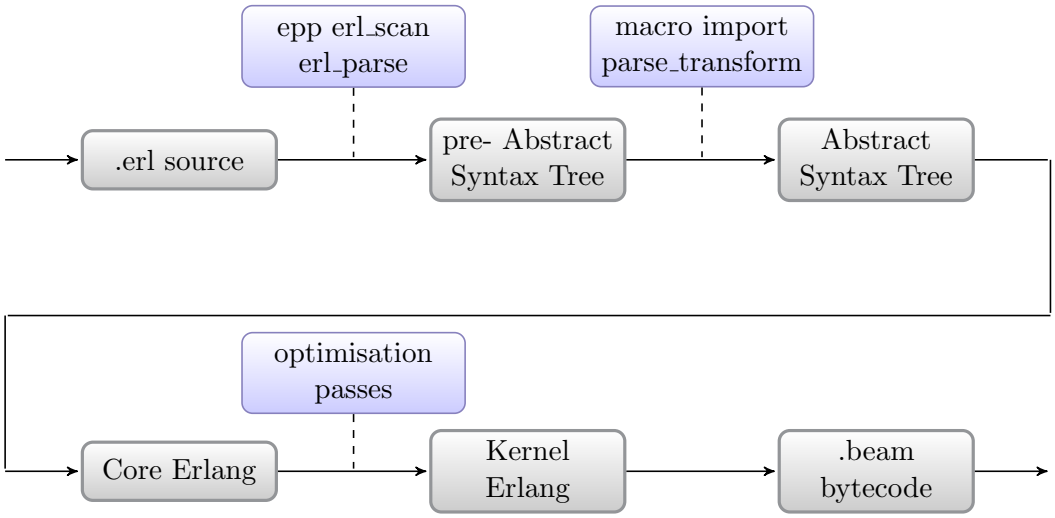

Figure 5.1: Overview of the *Erlang*'s compiler architecture

was *Erlang*'s lack of an API for manipulating the message-queueing system, apart from the standard **receive** construct. Both approaches remained consistent in terms of behaviour and we explain in detail the disadvantages and advantages of each.

Finally we describe the variations of the algorithm that we used in order to efficiently find synchronisation matches between patterns. We realise that in most of the implementations the performance significantly sacrifices the expressiveness of *Join-calculus* extensions and our aim is to reduce this effect in *JErlang*.

5.1 JErlang Compiler and VM

We present a general overview of the standard *Erlang* architecture, including details on how the compiler produces the binaries and how the Virtual Machine works. After that we introduce the changes that were necessary to fully use the possibilities of the mailboxes inside the *JErlang* Virtual Machine. Knowledge about both of the components will allow you to better understand the implementation changes that we introduced in order to support *Erlang*-like language.

5.1.1 Compiler

Erlang as a mature high-level language contains a scanner, parser and compiler written themselves in *Erlang*. Figure 5.1 presents a general overview of the path starting at the original *Erlang* code and finishing as the bytecode that is executed in the *Erlang*'s VM.
The first stage is obviously the preprocessor (*epp*), scanner (*erl_scanner*) and parser(*erl_parser*).

This creates a first abstract syntax tree, which is later processed through (possibly many) **parse_transform** functions that manipulate the input according to their definition. At this stage any macro definitions are also resolved. All of those actions are controlled by the *v3_core* module which delivers the human-readable *Core Erlang*[9] code. It is a simple, lexically scoped, functional language that still contains some common *Erlang* features like **try** or **case**. The role of *Core Erlang* is to have an intermediate language between the source code and the assembler, which is general enough to perform specialised optimisations and validation operations. This way we gain another layer of abstraction that discards language nuances. *Core Erlang* then becomes transformed into the assembler code *Kernel Erlang* and finally ends up as *Erlang*'s binary bytecode known as **BEAM**[2]. Appendix A contains an example of structures produced by the *Erlang* compiler for a simple function with the **receive** construct.
The final assembler code is very close to how the **BEAM** bytecode looks like. The latter are just word instructions followed by zero or more word operands.

5.1.2 Built-in functions (BIFs)

Function calls from the standard user-defined libraries are directly generating core *Erlang* code that is then analysed (if possible) during the compilation and verification procedures. However, apart from the standard function calls that are treated in a universal way, *Erlang* introduces special *built-in* functions, that are mostly written in *C* and available from the core **erlang** module. This allows for faster computation of critical parts of the execution as well as access to the internals of the *Erlang*'s VM. For instance standard +, **-** or ***** operators, or critical **self** and **send** function calls, are implemented using *BIFs*.

5.1.3 Erlang's Virtual Machine

The role of *Erlang*'s VM is to execute the compiled **BEAM** bytecodes in a language neutral emulator, perform necessary process management, garbage collection etc. From the detailed point of view *Erlang*'s emulator is a single function call that is continuously in a loop executing the **BEAM** instructions. The latter are locally translated into internal code (label) jumps. **BEAM** is register-based and fetches the necessary addresses of the operands into specified registers before executing the instruction. The VM maintains a program counter and stack of frames. It is interesting to note that *Erlang*'s support for a large amount of processors is not implemented by each process having its own thread, but rather the VM maintains an internal queue of available processes (organised by priorities) and a scheduler that performs their fine-grained management. Each process has associated with it a stack frame, a program counter and a heap, which is mainly used for storage of the message queue. For the **receive** construct, which is the one that is most relevant to this project, we outline the execution steps taken in the *Erlang*'s VM (we provide listing 5.1 as a simple example):

1. Each process maintains a pointer to the last checked message. When starting the **receive** it always points to the beginning of the queue.

[2]BEAM stands for Bogdan/Bjrn's Erlang Abstract Machine

2. Take the message which is pointed to by the marker in the queue and put it as the operand for matching.

3. Take the current **BEAM** instruction, which would represent the pattern, and execute the match given the operand from step 2.
   If matching is not successful, we go to step 4.
   If matching is successful, we go to step 6.

4. If the next instruction is a pattern (**BEAM**instruction) then we update the current program counter to it and again go to step 3.
   If that was the last pattern and there are still some messages left then we update the pointer of the mailbox to the next message, go the first pattern represented by BEAM instruction and execute step 2.
   If there are no further messages we go to step 5.

5. The VM sets up the timeout counter (if it wasn't already set up), and the process goes to sleep. It will either be awakened by the timer or by a new message. The former will force the jump to the **BEAM** instruction defined in the timeout action and the latter will store the timeout's current value, update the mailbox pointer and go to step 2.

6. A successful match frees the memory associated with the currently pointed-to message, updates the mailbox's marker to the beginning of the queue and jumps to the **BEAM** instruction pointed to by the pattern.

The current version of the *Erlang*'s VM supports *SMP* [3]. This results in the existence of many schedulers (usually one per CPU/core) that have shared access to the queue of the available processes. We mention this fact here, since some of the changes in our *JErlang*'s VM involved the correct usage of locking mechanism to ensure that our implementation is correct. *SMP* is an important feature of *Erlang* and we wanted our implementation to be compatible with the mainstream implementation.

```
1  basic() ->
2      receive
3          {one, One} ->
4              One.
5          {ok, Result} ->
6              Result
7      after 10000 ->
8          {error, timeout}
9      end
10 end
```

Listing 5.1: Simple **receive** construct in *Erlang*

[3] Symmetric Multiprocessing, see http://en.wikipedia.org/wiki/Symmetric_multiprocessing for a brief but thorough introduction

5.2 JErlang's VM

The main problem with implementing Joins inside the *Erlang*'s VM or language itself was lack of necessary operators that would enable us to manipulate and inspect the processes' mailboxes without the limitations presented in 4.1. Therefore we decided to implement two approaches:

- The queue is stored on the library level and fetches all the messages from the VM's mailbox in the original order. In this approach the only limitation is the semantics so we can practically can do everything with our new library-queues. The main drawback lies in the performance of the solution.

- We implement new functions and constructs that enable us to directly manipulate the mailboxes inside the VM, from the standard *JErlang*'s library level.

The subject of this section are the possibilities and challenges involved in both approaches. The decision to focus on the provision of only a minimal number of operators, instead of a full implementation of joins inside the *JErlang*'s VM, was due to the limited timescale of the project and the complexity of the VM. Since *Erlang* architects do not provide any documentation about internals of the system and we would have been drastically changing how one of the most critical parts of the product works, we leave it as an interesting project that could continue our work.
Nevertheless, we made sure that our implementation is fully operational and we remain backward compatible with the standard *Erlang* release[4] and existing applications.

5.2.1 Search mechanism

The nature of the Joins requires from our algorithm that it should be able to continuously parse the given mailbox in a similar manner as we have shown in the algorithm in section 5.1.3. Since we perform the synchronisation on multiple messages in the same mailbox we no longer want to remove the message immediately, but only receive an indication that our pattern matching was successful and the ability to restart the matching for other patterns in the join.
Example 5.2 presents a **search** expression which looks almost identical to the well-known **receive** apart from the first keyword.

[4]We implemented the changes against the most stable R12B-5 release available during our research. At the time of writing this report the new R13B version is released that was not tested against

```
search
    {first, Choice} ->
        Choice
    {valid, special} ->
        {ok, special_case}
    {valid, Result} ->
        Result
after 10000 ->
    {error, timeout}
end
```

Listing 5.2: **search** construct in *Erlang*

It is important to notice that **search** does not allow for multiple pattern declarations in a form of joins (as **receive**) but is used in the algorithm to achieve this for **receive** in *JErlang*.

The set of possible changes in accessing the processes' mailboxes were constrained by the need to remain backward compatible with the old behaviour of the **receive** construct. This is because we want to keep the standard, optimised semantics inside the *JErlang*'s standard libraries and on other applications that were not written to support join semantics. As a result we maintain a separate *search pointer* on the mailbox that influences the behaviour of the search mechanism and is independent from the original mailbox's *pointer*. This ensures that whenever the system uses the standard **receive** then pattern matching starts from the beginning of the queue and continues with usual *Erlang*semantics.

The most visible difference between **search** and **receive** semantics is that the latter automatically resets its pointer after a successful match, whereas the former does not. We use example 5.3 to present a scenario why this restriction had to be introduced.

```
receive
    {msg, A} and {reply, B} when (A > B) ->
        %% process message
after 10000 ->
    {error, timeout}
end
```

Listing 5.3: Joins construct in *Erlang*

```
search
    {msg, A} ->
        {msg, A};
    {reply, B} ->
        {reply, B}
after 10000 ->
    {search, timeout}
end
```

Listing 5.4: Search patterns used for parsing the queue of the example 5.3

Listing 5.4 presents corresponding search patterns used for resolution of the join patterns. We assume that the content of the mailbox at a given time is:
(**{msg, 12}** · **{reply, 21}** · **{reply, 9}**)

Clearly the first message would match successfully against the first pattern. The second message would similarly match against the second pattern. However this pair would fail the guard test, hence we have to continue to search further. The third message is also successful and creates a correct pair that satisfies the guard's test. We wouldn't be able to use this algorithm with *receive pointer*'s automatic reset (after successful matching), since we would always match against the same invalid message.

In the most simple approach, whenever *JErlang*'s **receive** expression is finished we loose any gathered information since the next receive (if any) usually contains different patterns. Therefore the matching search has to start from the beginning of the queue. *JErlang* provides function **search_reset/0** to explicitly set the *search pointer* to the head of the queue.

Listing 5.3 presents another challenge related to mailbox access. Even though we know that the sequence of messages 1 and 3 satisfies the join, we are unable to get the right messages using the standard **receive** construct because listing 5.5 will remove messages 1 and 2, due to the order of messages and semantics of **receive**.

```
1  return_messages() ->
2      First =
3          receive
4              {msg, A} ->
5                  {msg, A}
6          end,
7      Second =
8          receive
9              {reply, B} ->
10                 {reply, B}
11         end,
12     [First, Second]
13 end.
```

Listing 5.5: Removing expected messages using standard *Erlang*'s **receive**

We believe that this problem can be most efficiently solved by adding a unique identifier that enables to distinguish the messages as well as directly access them. Function **last_search_index/0** returns the identifier of the last message that matched the **search** expression, whereas the function **receive_index/1** removes the message associated with the identifier given as an argument to the function. The mentioned operators give us enough freedom to perform necessary mailbox manipulations.

Finally to avoid unnecessary double buffering of the messages, we associate and store their identifiers inside the pattern's caches[5] and later, when necessary, fetch messages to perform the tests. Selective receive by identifier is done using another function, **search_index**.

In order to implement the **search** construct in the VM, we added new jump labels inside the main execution loop, so that the BEAM instructions can "jump" to it. This also required a small number of adjustments inside the *Erlang*'s lexer and parser to mimic the structure of **receive**. Due to the complexity of the *Erlang*'s VM, other function calls were implemented as *BIFs*, instead of being completely new BEAM instructions. This enabled us

[5]Those "caches" store the functional representation of the pattern as well as the partial results

to write efficient *C* code that is able to directly access mailboxes of the processes.

5.2.2 Mailbox and Map

For large mailboxes and complicated join combinations, it could be the case that a large number of calls to the mailbox might need to be made to perform the tests for guards or non-linear patterns. Our evaluation tests have shown that accessing the mailbox implemented as a queue leads to the degradation of the performance. Given this bottleneck and unique identifiers associated with each message it seemed natural to use a better data structure. We decided to use the *uthash*[6] hash tables implementation because of their simplicity, good performance and positive opinions from other developers.
Each process in *Erlang*contains an additional hash table that maps identifiers to addresses of the messages. This mechanism is orthogonal to the existing queueing system, because we wanted to keep the backward compatibility, and the queueing of messages still performs its role well in the matching algorithm. In 6.4 we evaluate the possible advantages and disadvantages of the approach.

5.2.3 Locking

JErlang's emulator with *SMP* support introduces many concurrency threats like deadlocks, lost messages and race conditions. In order to make *JErlang* thread-safe we had to guarantee that accessing and removing the messages agrees with the overall locking system. Fortunately, since all the new functions are *BIF*s and have a direct link to the processes they are working on, it was straightforward to extend the locking to those critical sections using the provided *C* macros.

5.3 Parse_transform

In order to enable synchronisation in *JErlang* on multiple messages the programmers only need to use the familiar **receive** construct (as in listing 4.4). In reality the code that actually performs necessary join reduction is defined in the module **jerlang_core**. The main **jerlang_core:loop/3** function initiates the joins solving action when given correct definitions of patterns' tests and configuration parameters. Appendix B gives an example of the code constructed by the compiler in order to run the joins' function. It can be easily seen that even with the simple join definition the number of lines of code increases roughly from 15 to 54. The syntax of *JErlang* allows programmers' work to be less error prone, more intuitive and enables them to focus on the problem instead of the awkward parameters of the library call.

Typically, to allow for the new syntax of *JErlang* in **receive** we would have to change *Erlang*'s grammar rules for the lexer and parser, even for the non-VM's *JErlang* implementation. Since our aim was minimise the amount of effort that a potential user of *JErlang* would have to make in order to test the library, we decided to take a different approach.

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

parse_transform is a module available in *Erlang*'s standard library that allows the developers to use *Erlang* syntax, but with different semantics. In order to facilitate this feature the following criteria have to be fulfilled:

- The original code has to conform to the syntax *Erlang*
- The programmer has to define a module that implements a function **parse_transform** and exports it. The function is given two arguments: the original Abstract Syntax Tree, and additional compiler options. The developer informs the compiler of the possible transformation by including the code line:
  compile({parse_transform, module_parser}).
- The abstract syntax tree produced by the transformation process has to have a valid *Erlang* structure and semantics.

Appendix B presents an example of a transformation that is done on a standard Abstract Syntax Tree given by the **receive** with joins construct. **parse_transform** is therefore capable of constructing completely different programs which allowed us to avoid changing *Erlang*'s compiler in order to accommodate *JErlang*'s extended syntax [7].

5.3.1 Abstract Syntax Tree

Erlang, in comparison to other popular languages like *Java* or *Python*, doesn't have to maintain a costly *visitor* design pattern in order to parse the AST. Instead, we use intuitive pattern matching on the blocks of the tree and hence dismantle it into necessary branches. Manipulation of the branches follows from the inofficial (and slightly outdated) specification of the language[19]. Creating new programs is often straightforward (see appendix B) however in order to analyse general construct transformations we mostly have to parse the entire tree in search for the required patterns.

Example 5.6 presents a rule (function) for finding the function definitions in the code and performing specialised transformations on it (not defined in the fragment). It is typical to use tail recursion algorithms in order to parse the tree, since they are compiled efficiently in the *Erlang* language. [19] presents the latest available language reference manual for *Erlang*, although the current implementation differs slightly from that specification.

[7] The only drawback of this approach is the lack of support from the *Erlang*'s developers team, however the approach of changing the internals of *Erlang* is much harder and less portable

```
1  ....
2  parse_transform(AST, _Options) ->
3      Result = parse_tree(AST, []),
4      io:format("Tree: ~p~n", [AST]),
5      io:format("Result: ~p~n", [Result]),
6      Result.
7
8  parse_tree([], Ast) ->
9      lists:reverse(Ast);
10 parse_tree([{function, L, Name, Arity, Clauses} | Rest], Ast) ->
11     TransformedFunc = parse_function(L, Name, Arity, Clauses, []),
12     parse_tree(Rest, [TransformedFunc | Ast]);
13 parse_tree([Node | Rest], Ast) ->
14     parse_tree(Rest, [Node | Ast]).
15 ...
```

Listing 5.6: Extract from **parse_transform** module that searches for functions definitions in the code

In the case of **receive** construct we search for the definitions of the *receive* clause in the AST, and for **gen_joins** the headers match on the function definitions with name **handle_join**. The former is even more complicated as we can have a **receive** inside another **receive**. Obviously the AST for *JErlang*'s programs will mostly be invalid semantically (when using its language features) and the compiler will return an *invalid pattern* exception. The aim of our transformation is to find joins patterns, create necessary tests for patterns and joins (we discuss the algorithm later) and create the code that calls the main *JErlang* library in order to execute the joins correctly. The amount of code produced by the new representation is linearly dependent on the number of patterns and hence the binary code produced by the compiler will be larger than what developers would normally expect. We believe that this doesn't affect the usability of the code, since the number of **receive** constructs and patterns in them is typically small (i.e. from 2 to 5). Transformation is done only once during the compile time, hence the overhead of joins during run-time involves only the time spent on solving the joins.

5.3.2 Unbounded and Bounded Variables in parse_transform

One of the challenges of code transformation was the attempt to create output that is acceptable in *Erlang* syntax. In other words, warnings, line numbers and errors in *JErlang*'s programs that are reported by the compiler should be possible to identify by the standard *Erlang* lint[8] and compiler modules.

An example usage of the **parse_transform** involves copying fragments of the AST and pasting it to a different place, but as we will show this often creates undesired effects. In the example 5.7 the variable **Test** in a tuple and variable **Result** are used to define a function call which is compared to the original tuple, as presented in listing 5.8. Since **Test** is unbounded in the original program, the compiler will result in a single *unbounded variable*

[8]The Lint module reports any warnings and errors for fragments that were not desired by the developer

error. However when performing the transformation our compiler will return an *unbounded variable* error twice for **Test** variable. This creates unnecessary confusion to *Erlang* programmers as they should not be aware of any transformation going on behind the scenes. Clearly the output of the compiler in 5.7 doesn't make any sense, whereas in 5.7 it correctly identifies the errors, because we know how the function is defined.

For brevity we only give this relatively simple situation, but more confusing constructs are often written by the programmers resulting for instance in unclear *unused variable* warnings. The next two sections describe known techniques used for avoiding generation of "bad" code.

```
1 wrong_warning(Result) ->
2     {ok, Test} = Result,
3     ok.
4 %% Erlang's compiler output:
5 %% example.erl:2: variable 'Test' is unbound
6 %% example.erl:2: variable 'Test' is unbound
```

Listing 5.7: Example of confusing **parse_transform** transformation that disinforms the developer about the code

```
1 wrong_warning(Result) ->
2     {ok, Test} == any_function(Test, Result),
3     ok.
4 %% Erlang's compiler output:
5 %% example.erl:2: variable 'Test' is unbound
6 %% example.erl:2: variable 'Test' is unbound
```

Listing 5.8: Real *Erlang* representation after filtering 5.7 through simple **parse_transform**

5.3.3 Reaching Definitions Analysis

By definition of **receive** whenever a message arrives and it satisfies the pattern it bounds any unbounded variables in the pattern to their respective values in the message and agrees with any bounded variables. In *JErlang* we substitute the definition of **receive** with the call to *JErlang*'s solver where multiple definitions of anonymous functions that represent the patterns in the joins are given as arguments. Since in *JErlang*'s implementation we perform isolated tests only for patterns, without taking into consideration the body of the join (as described in 5.4), the former would create multiple *unused variable* warnings by the compiler. To eliminate this problem we substitute each such variable with a neutral _ variable[9]. However this implementation would be wrong for the latter case, when the variable is already bounded and testing of the message has to take into account this information.

Therefore we perform a variation of *Reaching Definitions Analysis* which determines the number of variable definitions that reach any expression. This allows us to deterministically distinguish between those two cases. For the patterns' test functions we leave the original name for the bounded variables and perform the substitution for the unbounded ones. Listing

[9]Variable _ describes a fresh variable, and thus is ignored by the compiler

5.9 presents a simple definition of the joins in *JErlang* and an example of test functions that would be created for it. Clearly without this analysis line 13 would create an *unused variable* warning for header **{A, Rest}**.

```
1  test_receive(Input) ->
2      A = 12,
3      receive
4          {ok, Input} and [A, Rest] ->
5          valid
6      end.
7
8  %% Sample patterns' tests created for above join
9  receive_patterns() ->
10     First  = fun({ok, Input}) -> true end,
11     Second = fun([A, _]) -> true end,
12     {First, Second}.
```

Listing 5.9: Simple joins definition in *JErlang* and schematic tests for joins patterns'

Reaching Definitions Analysis is easier to perform than in procedural languages (as defined in [31]) due to the functional nature of *Erlang*Ȧdditionally we are only interested in the number of variable definitions (in *Erlang* it can be either none or 1), rather than the exact program points as in the classical analysis.

5.3.4 Live Variable Analysis

A different form of analysis is needed, in comparison to 5.3.3, when it comes to transformation of **gen_joins** code. In **gen_joins** behaviour joins are specified in the header of the function instead of in the body and hence do not require any past knowledge of the variables. In 5.4.4 we define an efficient algorithm that performs successive matching on partial joins and also on *status* and *guards*. To avoid the situation described in 5.3.2 and execute the matching tests on non-linear patterns as soon as possible, we perform a variant of static *Live Variable Analysis*. This process enables to determine whether the variable in the partial joins' test should be substituted with the neutral _ one or left unchanged because it used more than once in the join.

Example 5.10 presents a simple schema that illustrates the process of analysing and building **gen_joins** tests. For simplicity reasons headers of the functions match on the lists of patterns. In the example we define three auxiliary functions which decompose the original **handle_join** joins into consecutive stages. Unbinding the variables that are used only once prevents the occurrence of multiple *unused variable* warnings. Similarly leaving the bounded variables allows for early filtering of the messages that have to agree on the values of the variables with the same name.

```
1  handle_join( {first, A} and {second, A, B} and [B, Result] and {ok, final},
2  Status) ->
3  ...
4  %% Execute action and return new state and replies
5
6  %% Example tests generated for partial joins that are
7  %% generated by the JErlang's parse_transform
8  first_test( [{first, A}, {second, A, _}], _ ) ->
9      true.
10
11 second_test( [{first, A}, {second, A, B}, [B, _]], _ ) ->
12     true.
13
14 third_test( [{first, A}, {second, A, B}, [B, _], {ok, final}], _ ) ->
15     true.
```

Listing 5.10: Extract from the program implementing **gen_joins** behaviour in *JErlang* and schematic tests for joins patterns'

5.4 Algorithms

Finding a solution to the multiple pattern-matching problem has been the subject of many research works. The main aim of ours is not to define an algorithm that will surpass all of the existing ones (for obvious reasons), but rather combining their advantages in order to produce the most efficient and suitable design for solving joins as they are defined in *JErlang*.

Currently popular joins implementations in languages like *Scala*[16] or *Polyphonic C#*[5] follow the tradition of the first main *Join-calculus* implementation in *JoCaml*[26]. Additionally in *Polyphonic C#* messages can be uniquely assigned to the channels on which the synchronisation is done[10]. Solving joins in such a system naturally corresponds to building finite state automata with the sequence of channels' sizes (current number of messages) uniquely determining the state. Figure 5.2 adapted from [26] presents a full transition image that is produced for the fragment of the *JoCaml* application given in listing 5.11.

```
1 let Set(number) | Get() = reply number to get
2 and Set(number) | Print() = print_int x; 0
3 ;;
```

Listing 5.11: Simple joins definition in *JoCaml*

Similarly as in Maranget's work we denote none of the messages in the channel by **0** and at least one message by **N**. The straight lines of the diagram describe the transitions performed upon retrieval of the new message to the specific channel, dashed lines inform of a successful pattern-match applied to the joins. The latter simply decreases the number of messages in the channels that participate in the synchronisation. Clearly the automaton represents non-deterministic behaviour because at any given time we can receive a message on any channel, and (if possible) perform joins reduction. It is possible to force deterministic behaviour by applying a joins transformation as early as possible and this is what most of the existing implementations do. Thus retrieval of the messages is only done when joins matching is not possible.

Finite state automata immediately force some limitations on the joins, which then influence the extended languages. For instance, *Polyphonic C#*'s *chords* can be very efficiently compiled statically but prevent the usage of non-linear arguments and pattern matching on arguments as well as restrict the number of synchronous calls to maximally one [11]. The *Scala* library is much more flexible in terms of the available options. Matching on sequences of *events* is performed by transforming the joins into series of nested **case** statements that check for all the possible variations of the events in order to fire the join. This does not yet allow for having non-linear pattern matching, yet the authors also introduce *guards* where we can perform standard operations (like equality). Unfortunately, the original paper does not provide any benchmarks with respect to the performance of *Scala* joins therefore it is hard to determine the efficiency of this solution.

[10] In case of *Polyphonic C#* we consider function calls also known as *chords*, whereas in *Scala* synchronisation is performed on *events*

[11] see 2.2.2

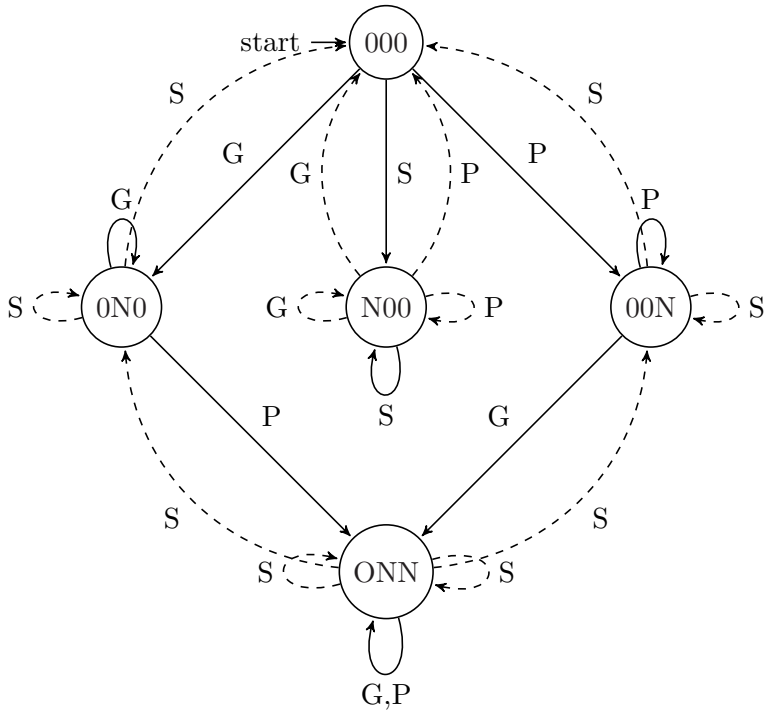

Figure 5.2: Non-deterministic transition graph representing joins in *JoCaml*. G, S and P stand for Get, Set and Print respectively.

Internally, *Scala*'s extensible pattern matching [16] is implemented using partial functions and *extractors*. Each incoming *event* is associated with the possible join pattern and using the partial functions it is determined whether it is valid to enqueue it in the internal buffer associated with the pattern. In order to perform compound join on the patterns, the *events* are *extracted* and applied to the joins.

In the *JErlang* implementation we borrow some of the ideas from *Scala*'s algorithm as it relates partially to how pattern matching exists in *Erlang*. However our implementation brings more freedom to the programmers when building complex join definitions.

5.4.1 Standard

In addition to the previous discussion our algorithm borrows ideas from *CHR*[36]. The latter is focused on an efficient solving of transformations for multi-set constraints. We believe that the key to the adoption of joins in *JErlang* lies in the adoption of the optimized search engine, so that the gain in developers' expressiveness outweights the inevitable performance

drawback in the general implementation[12].

It is important to remember that *JErlang* provides support for different kinds of joins. The **receive** construct and **gen_joins** behaviour present different characteristics that can be used to create efficient solutions. In the following discussion we make a clear distinction whenever the implementation is applicable to only ony of the approaches, and you can assume that in general features apply to all our types of joins.

In *JErlang*'s **receive** each of the join's patterns has an associated test function. Any message that is sent to the process is checked by the join patterns' tests. This is the consequence of *JErlang*'s semantics where a message can match more than a single pattern and ensures the consistency of our implementation. In the case of joins in **receive**, which are always executed in some context, some of the variables used in the test can already be bounded and functions reflect this state, since they are build at compile time in the same context. Example 5.9 defines two functions necessary to initially test the join's patterns.

In our algorithm we use an additional data structure that servers as a cache for storage and retrieval of the results of the matching. We believe that the overhead associated with it is acceptable, because we store only the message's unique index and perform the matching operation less frequently. Some of the implementations that do not store this partial result suffer from repeated (unnecessary) pattern checking. The standard algorithm follows the *First-Match* semantics[13]. In general terms the basic version of the algorithm for joins resolution can be described as following:

1. Read the incoming message from the head of the queue and take the first join.

2. Take the list of test functions associated with the join and check the message on each of the patterns. If the test was successful, store its index in the cache of the corresponding pattern.

3. If none of the patterns' caches was updated go to step 6, otherwise go to step 4.

4. Take the final test function associated with the join and run it on all possible permutations of the satisfying messages. Go to step 5 if there is at least one successful run, or step 6 otherwise.

5. Take the first successful run and associated messages for each pattern. Update the context to include any previously unbounded variables that existed in the patterns and run the corresponding join's body in it.

6. Since the execution of the current join was not successful, update it to the next one and go to step 2. If the current join is the last one and there are still some messages

[12]By performance drawback we refer to the personalied solution to the synchronisation problem that can always be written by the programmer

[13]see 3.32 for the advantages and disadvantages of this approach

in the queue, update the message pointer and go to step 1, otherwise stall until a new message arrives to the process.

For simplicity, we do not describe the action to be taken when a *timeout* action is specified. This is analogical to the algorithm described already in 5.1.3. In the case of *JErlang* that uses the modified VM, step 1 actually corresponds to the **search** construct.
In the non-VM version we use a standard receive construct that matches any message. In order to prevent data loss we actually maintain two mailboxes for this case: one with messages already processed that are stored in the "internal" library queue and the other with those still awaiting in the VM's standard mailbox. Obviously any new **receive** join execution first goes in order through the messages in the former queue to ensure the expected semantics. For illustration purposes figure 5.3 presents a schema that follows the basic algorithm to solve simple joins.

In **search** we used optimisation, where we order the matching patterns such that the most specific ones are checked first and the most general ones are at the end, as illustrated on the example 5.13. This also avoids generating spurious compiler warnings as well as uses VM's efficient internal matching mechanism to filter out messages that have no possibility to succeed. This doesn't affect the original ordering of the messages, because messages are not removed until the final join's fire and the cache information is invalid after the execution of the expression anyway.

In general, the implementation of **gen_joins** behaviour follows the algorithm defined above, but there are few additional challenges associated with it. In **receive** once we find a successful sequence of the messages that satisfies the join, we immediately execute the body and reject all gathered information about processed messages. This is because any new **receive** will have new join definitions and the cost of checking whether it is the same or at least similar to the previous one outweights any performance gained. **gen_joins** on the other hand has joins defined only once, during compile time and there is the possibility of reusing gathered knowledge. The joins' solver doesn't have to repeat the tests for the patterns for the already parsed messages since successful run of the test-function is independent of other factors. Another challenge introduced by **gen_joins** is the internal *status* variable[14]. Since execution of the join may have side-effects on its value, joins that previously couldn't be fired may now have become successful with the new value for the *status* variable. This operation has to be performed immediately after firing a join. Our algorithm takes into account the possibility of a chain of join actions that does not involve analysis of any new packages (this agrees with the strategy taken by *JoCaml*to create deterministic automata).

We found that this intuitive algorithm, although it performs the synchronisation correctly, is not very efficient. It becomes a bottleneck for applications where a large size of mailboxes is allowed and joins with many patterns are used. In the next section we further discuss this problem and in sections 5.4.3 and 5.4.4 present a novel and efficient approach to solving complex joins in *JErlang*.

[14]Status variable allows for the server to maintain internal storage (memory),instead of being a pure action-reaction client-server design pattern

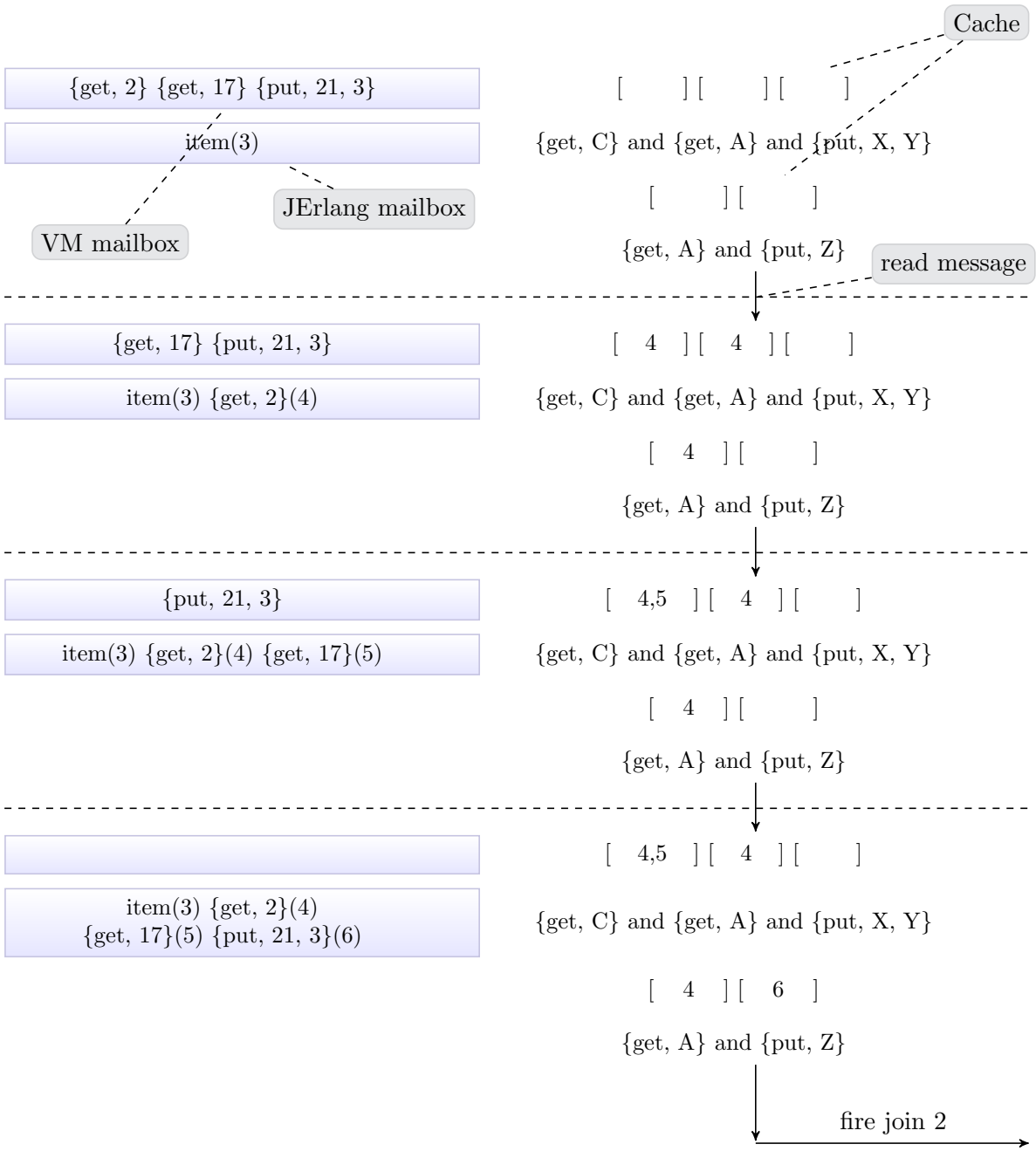

Figure 5.3: A general architecture for solving simple **receive** joins in *JErlang* for listing 5.12. We are given an initial mailbox for VM and library, internal storage for partial tests' results and corresponding patterns. Messages inside the *JErlang*'s queue are assigned unique indexes (numbers in brackets). Message **item** is a "leftover" from previous joins execution, but since it does not match any of the patterns it is later no longer considered in matching.

```
1  diagram(A=2) ->
2      receive
3          {get, C} and {get, A} and {put, X, Y} ->
4              ... %% first join
5          {get, A} and {put, Z} ->
6              ... %% second join
7      end.
```

Listing 5.12: Simple definition of joins available in *JErlang* used in figure 5.3

```
1  simple_joins() ->
2      receive
3          {read, A} and {book, tolkien} ->
4              borrow_book(A, tolkien);
5          {read, john} and {magazine, B} ->
6              borrow_only_to_john(B)
7      end.
8
9  %% First stage that uses Erlang's internal matching
10 %% to filter useless messagess
11 first_stage() ->
12     search
13         {read, john}=V ->
14             V;
15         {read, A}=V ->
16             V;
17         {book, tolkien}=V ->
18             V;
19         {magazine, B}=V ->
20             V
21     end.
```

Listing 5.13: Ordering of patterns in **search** construct used in the first stage of joins resolution in **receive**

5.4.2 State space explosion

As a motivating example for showing the challenges related to the standard algorithm used in *JErlang*, we will use the program used for solving the *Santa Claus* problem6.2. The details of the problem are irrelevant at the moment. The only important thing that the reader should take note of is the existence of multiple, identical patterns in the joins. This quite usual situation will take unacceptably long time to perform using *JErlang*'s standard algorithm. It is easy to create similar problems by sending different messages that match many patterns.

Given the message **{reindeer, a}** it matches all nine patterns in the first join defined by **receive**. Having nine such messages we know that the join can be satisfied. Nevertheless, our (primitive) algorithm will compute and test all the possible permutations which can take at least $9! = 362880$ test operations. Clearly this is an unacceptable number. This situation does not arise in *Polyphonic C#* or *Scala* due to its syntax limitations.

5.4.3 Lazy evaluation

It can be easily noticed that the basic algorithm performs unnecessary work related to testing the messages. We use a better one where instead of performing all possible computations, it finishes immediately once a successful sequence of messages is found. This significantly reduces the time necessary to find and fire a matching join. We also implemented techniques that enable us to prune branches of the search tree which cannot be successful. For instance this can mean exclusion of sequences where the same message appears more than once.

At this stage of the implementation we believed that there was a fair number of optimisations that could be introduced during compile- and run-time in order to increase the efficiency of our algorithm. One of the examples would include detection of the situation presented in 5.4.2, where multiple patterns could be satisfied as soon as we receive the number of messages equal to the number of the join's patterns. Similarly, we could statically analyse the code in order to detect patterns that are dependent upon each other and only perform final tests on those. Instead of working on the optimisations for the joins' solver immediately, we decided to focus our efforts on finding the framework that addresses the efficiency issues for more general cases. Only then the application of the improvements mentioned above made sense. In the next section we present an algorithm relatively similar to our original work that, we believe satisfies those conditions.

5.4.4 RETE implementation

The problem of finding pattern-matches is a long standing problem that had already been considered in the early days of computing[15]. In order to improve the efficiency of our solver we borrowed ideas from the RETE algorithm[16], even though it relates to a slightly different discipline. Surprisingly only one existing implementation, *HaskellJoinRules*[22][23], based its pattern resolution mechanism on the research in the area of *Production Rule Systems*.

A fundamental decision that we made in *JErlang*'s solver was that it will use a sequential version of the RETE algorithm. Authors of *HaskellJoinRules* focused on parallelising their solution by using a combination of the LEASE algorithm (a derivative of RETE) and a nowadays popular locking mechanism called Software Transactional Memory (STM). The most important advantage is the speedup that they can gain when using *HaskellJoinRules* on the multicore machines. In the case *JErlang*, we decided to give higher priority to two main features that couldn't be easily satisfied in the former solution:

- The ordering of the joins influences the decision to fire the join i.e. given a situation when two joins can be fired, the first one that is defined is always executed.
- The order of the messages in the mailbox has to be preserved.
- *JErlang* has to preserve the fundamental architecture of *Erlang*, where there is no shared memory between processes.

[15] see `http://drofmij.awardspace.com/snobol/` and `http://en.wikipedia.org/wiki/SNOBOL`
[16] see 2.4.2

In order to parallelise the execution of the solver each of the helper processes would have to posses either a copy of the master's mailbox or have a possibility of directly accessing messages in it. The latter would clearly violate *Erlang*'s no-sharing principle and the former would involve frequently copying a large number of messages, which we believe outweights the efficiency of parallelism.

It is important to underline the fact that the RETE algorithm creates a reasonable high memory burden by creating alpha- and beta-reductions in an iterative way. Therefore, the main implementation of the RETE algorithm was imposed on the **gen_joins** behaviour, since it can preserve some knowledge after a successful match. The introduction of this algorithm to **receive** would be a straightforward work, given what we have learned on joins in **gen_joins**, and was not finished only due to the time constraints we are subjected to.

Alpha-reduction is performed in a standard way by having local test functions for each of the joins' patterns. In comparison to RETE, a new message does not run tests on all of the patterns, but only on those which belong to the currently tested join. This strategy agrees with our initial algorithm and allows for lazy evaluation of the joins, thus avoiding unnecessary computation.

Beta-reduction on the other hand is performed by having multiple test functions that perform partial checks of the joins. Again, we differ here with RETE by not formulating partial test for the single pattern check to avoid unnecessary computation.

Checking for consistency is performed through the headers matching, since in **gen_joins** we do not have to take into account the context of the join. Joins of length 1 would have a single beta function and for joins with n patterns, we produce n - *1* beta functions. Listing 5.14 presents a **handle_join** function with 4 patterns and 5.15 the corresponding beta-functions for RETE algorithm.

```
1  handle_join( {operation, Id, Op} and {num, Id, A} and {num, Id, B} and {num,
2  Id, C},
3  Status) when (A > B) ->
4  Result = Op([A,B,C]),
5  {[{reply, {ok, Result}}, noreply, noreply, noreply], [Result |
6  Status]}.
```

Listing 5.14: Multipattern join in **gen_joins**

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

Listing 5.15: Beta-reduction tests for listing 5.14

Each of the test functions (alpha and beta) has a corresponding cache memory that stores indices of messages that satisfy the tests. In case of beta-reductions we store sequences of indices. The only exception from this is the last beta-reduction function which is always empty, since its successful run means we found a correct join.

In a straightforward implementation of RETE in *JErlang* we have to clean up all of the beta-caches because of the **Status** variable available in **gen_joins**. The state can be changed during the execution of the join's body and any sequences of patterns that were dependent on it become invalid. There is some place for improvement by detecting if the state has changed at all and also checking which of the beta-reductions are actually dependent on it. Our work does not yet support this kind of optimisation and is an interesting subject for analysis in the future.

Typically messages are to be removed from the mailbox in the joins execution step and given as arguments to the synchronisation definition. However, the propagated values are only copied, thus leaving the ordering preserved. In comparison to *HaskellJoinRules* we do not introduce any constraints on the sequence of patterns (in *HaskellJoinRules*propagation messages always have to be defined at the beginning of the chord definition). Additionally *HaskellJoinRules* support for propagation is done in a primitive way - the library simply consumes the message and produces it again inside the join's body. This was a technically possible, but unacceptable way in *JErlang* as it would not introduce anything substantial to the language.

5.4.5 Pruning search space

In order to improve the efficiency of the algorithm we prune branches of the search space that cannot be satisfied or that are easily satisfiable by some previous branches and determined in an efficient way. For the latter case we maintain a separate global cache that gathers in the groups the indices of equal messages. This way we perform computation for one sequence of values and all other sequences that only permute messages having equal values are simply pruned. This for instance radically improves the performance of the Santa Claus problem solution in the **gen_joins** implementation[17].

By doing static analysis of the code we can further improve the pruning of "bad" branches. In the example 5.15 guards and **Status** variable are applied only to the last test function. This approach is used in implementations that support guards or similar features. We believe that this construction insufficiently uses the knowledge about the patterns, because variables **A** and **B** in line 5 already provide information necessary to use the guard $\mathbf{A} > \mathbf{B}$. Therefore a challenging idea is to check, to which earliest beta-function we can apply guards and additional variables[18], so that the filtering of invalid messages does not produce false partial results. We found this idea especially interesting for the case when *JErlang* has to handle very large mailboxes, a situation that is also the Achilles' heel of *Erlang*.

[17] see Appendix D

[18] [36] provides theoretical background applicable to the *CHR*and only brief overview of the possible implementation for *Haskell*

5.4.6 Patterns ordering optimisation

Our final optimisation attempt also relays on the static analysis of the patterns in *JErlang* programs. We decided to investigate the dependency between ordering of the patterns and efficiency of the joins solver, especially in the context of the RETE algorithm. A similar approach was taken in [18], where the authors focused on optimizing compilation of *Constraint Handling Rules*. Their research also includes other possibilities not applicable to *JErlang*. Unfortunately the paper evaluates the impact of all factors and it is hard to determine how *ordering* on its own improves the performance. We believed that the order of the patterns *JErlang*'s joins is an important factor that could improve the usability of our language. Especially appealing is the fact that we perform the analysis during compilation, so we gain a run-time performance boost.

As a motivating example, we consider listing 5.17 with multiple patterns that could represent joins defined by a typical programmer. Obviously, the efficiency of the joins solver must not require from the developer to create patterns in the order that is most efficient, but rather most convenient. It is important to remember that each partial set of patterns from the joins increases the time to solve them. The idea behind the optimisation is to abort the incorrect sequence of messages as soon as it is possible. Example 5.17 gives a transformed **handle_join** function that, we believe, would perform better in our implementation due to the non-linear dependencies between the patterns as well as the existence of dependent variables in the guards and status. This feature is crucial for the matching against larger number of messages and/or patterns as we store an increasing number of partial results.

```
1 handle_join({value, B} and [#product{name=A, price=C} | _]
2               and {ok, size, D} and {range, C, E} and {value, A},
3                   {dict, D} ) when ( A < E) ->
4         %% Perform some computation
5         Reply = ...
6         NewD = ...
7         {[noreply, noreply, {reply, Reply}, noreply, noreply], {dict, NewD}}
```

Listing 5.16: Multiple-pattern joins in **gen_server**with interdependecies using *JErlang*

```
1 handle_join( [#product{name=A, price=C} | _] and {value, A}
2                   and {range, C, E} and {ok, size, D} and {value, B},
3               {dict, D}) when ( A < E) ->
4         %% Perform some computation
5         Reply = ...
6         NewD = ...
7         {[noreply, noreply, noreply, {reply, Reply}, noreply], {dict, NewD}}
```

Listing 5.17: Improved layout of Multiple-pattern joins in **gen_server**with interdependecies using *JErlang* of the 5.17 example

The algorithm analyses the structure of each pattern and assigns a rank to each of them, depending on the number of variables that it shares with other patterns. We also take into account the existence of variables inside guards and the status parameter. The optimisation selects the pattern with the highest rank and follows with inclusion of the dependent patterns in a recursive way. In order to be consistent, the algorithm transforms

accordingly the order of the (a)synchronous replies to the calls at the end of the **handle_join** method. The latter introduces only two lines of code that return the new ordering of replies to the **gen_joins** internals. We cannot perform an exact static transformation on the last part because programmers might use more complicated constructs like list comprehensions in order to create the reply, instead of manually creating the list. Our optimisation does not perform any code transformations once it detects that the original ordering is optimal.

Chapter 6

Evaluation

Introducing new features to the language is often hard as programmers may doubt their usefulness or complain about the performance. *JErlang* is no exception to this rule and as an additional challenge *Erlang* has a large community of programmers and architects with a very conservative mindset. In this chapter we present situations where the join semantics is more natural to express the programming problems than previous approaches.

6.1 Correctness

Before attempting to implement any of the language features presented in 4.2 we designed a set of test suits that enable us to check the correctness of the joins behaviour. Altough there can never be, enough test-cases, we do believe that those we provide is extensive enough to say that it works as expected. Listing 6.1 gives a feeling of the type of functionality tests that can be run on *JErlang*(defined in **jerlang_tests**).

```
1  ...
2  test_timeout() ->
3      io:format("Starting timeout test...~n", []),
4      clear_mailbox(),
5      ok = receive
6               _ ->
7                   error
8           after 0 ->
9               ok
10         end,
11
12     self() ! {foo, 1},
13     self() ! {foo, 2},
14
15     ok = receive
16              {bar, _X, _Y} ->
```

```
17                  error;
18              {foo, 3} ->
19                  error
20          after 1000 ->
21              ok
22      end,
23      B = -100,
24      ok = try
25              receive
26                  _ ->
27                      error
28              after B ->
29                  error
30              end
31          catch
32              exit:{error, {invalid_timeout, _}} ->
33          ok
34       end,
35
36      Self = self(),
37      spawn(fun() ->
38              timer:sleep(1000),
39              Self ! {foo, bar, 100}
40        end),
41      %% TODO: better ranges checking?
42      ok = receive
43              {X, Y, Z} ->
44                  X = foo, Y = bar, Z = 100, ok
45          after 3000 ->
46              error
47          end,
48        ....
49
50  test_guards() ->
51      io:format("Starting guards test...~n", []),
52      clear_mailbox(),
53
54      self() ! {foo, 12, 14},
55      self() ! {bar, invalid},
56      self() ! final,
57      ok = receive
58              {foo, X1, _X2} and {bar, _} when (X1 > 20) ->
59                  error;
60              {foo, X1, _X2} and {bar, _} when (X1 < 20) ->
61                  ok
62          end,
63        ...
```

Listing 6.1: Extract from tests module

The usage of the macro in line 3 ensures that non-VM and VM implementations have to pass the same number of tests, and the only difference to the programmer in the code is the name of the transformation module (the latter obviously requires the patched version of R12B-5 *Erlang* release). This ensures the stability of the extension as a complete package.

Apart from testing the standard **receive** construct, we provide additional examples for **gen_joins** *Open Telecom Platform*. Those are defined in separate modules whose names start with *jerlang_gen_joins_test_*, and the module **jerlang_gen_joins_test_main** provides automated execution that creates processes and checks the synchronisation patterns by calling asynchronous and synchronous functions.

Unfortunately, we were not able to get access to the test-suite that could have been used to extensively check the stability of our VM behaviour. The most likely reason is that the Ericsson team still pursues the closed-source policy in this matter (we doubt that they were able to produce this system without a consistent test-suite for the *Erlang*'s Virtual Machine). Nevertheless, compiling the run-time of *JErlang* from source doesn't cause any segmentation faults nor unexpected errors, and any invalid accesses would crash the compiler. Additionally we ran *JErlang* on a popular AMQP[1] project, *RabbitMQ*[2] entirely written in *Erlang*, and the system remained stable for a long period of time, while exchanging messages.

To separately check the functionality of our new *BIF*s and cooperation between **search** and **receive** constructs we also provide test-suits. Those are located in the module **jerlang_tests_vm** are defined in a similar style as 6.1.

It can be seen that the examples 6.1 present a more deterministic behaviour where we know the ordering of the messages. Apart from that in all our tests, we have also scenarios where we spawn new processes that typically sleep for some normally distributed amount of time and then send messages, thus creating a non-deterministic environment. Our joins handle these cases as expected.

A number of small applications were created including:

- Santa Claus problem (numerous variants)
- Dining philosophers problem
- Multiple reader, single writer problem
- Chat-system
- Distributed calculator
- Wide-finder challenge[3]

Some of those examples are used in the following sections in order to better understand the applicability of *JErlang*.

[1] Advanced Message Queueing Protocol http://www.amqp.com
[2] http://www.rabbitmq.com
[3] http://www.tbray.org/ongoing/When/200x/2007/09/20/Wide-Finder

6.1.1 Dialyzer

In our implementation we followed a general practice to define optional type system definitions called *specs*. Commonly, *Erlang* developers do not provide definitions for all of the functions but only those which are *exported* from the modules. Dialyzer then is able to perform static analysis of the code which in our case helped to remove a few redundant matching patterns. We believe that our specs have increased the maintainability factor of the whole library, because the parameters of our *JErlang*functions are often non-trivial[4].

6.2 Language and expressiveness

This aim of this section is to analyse the degree to which our extension allows for more freedom to *Erlang* programmers.

6.2.1 Santa Claus problem

First defined by Trono in [39], this problem is an extension of a typical semaphore problem and serves as a good test of how expressive a language is in terms of solving concurrency - synchronisation problems. Ironically the solution originally provided by Trono along with the paper proved not to be entirely correct, which shows how the problem actually is.

Definition 6.1 (Santa Claus problem)**.** Santa Claus sleeps at the North pole until awakened by either all of the nine reindeer, or by a group of three out of ten elves. He performs one of two indivisible actions:

- If awakened by the group of reindeer, Santa harnesses them to a sleigh, delivers toys, and finally unharnesses the reindeer who then go on holidays.
- If awakened by a group of elves, Santa shows them into his office, consults with them on toy R&D, and finally shows them out so they can return to work constructing toys.

A waiting group of reindeer must be served by Santa before a waiting group of elves. Since Santas time is extremely valuable, marshaling the reindeer or elves into a group must not be done by Santa.

Many solutions were proposed, typically using constructs like semaphores (or similar), but since *Join-calculus* it became obvious that these problems could be more elegant. For instance [7] shows a *Polyphonic C#* implementation. We will compare the reasonable solution provided by Richard A. O'Keefe [5] in *Haskell* and *Erlang* with the one written in *JErlang*. We provide the former for reference in the Appendix D.

All of the existing joins-inspired solutions seem to treat the *Santa Claus* problem as a variation of two unbounded buffers, that can be fired when they acquire necessary size. We provide a solution in this style in D.2, but we believe that such attempt is against the

[4]This is obviously not a result of badly designed library, but the complexity of the patterns
[5]http://www.cs.otago.ac.nz/staffpriv/ok/santa

programmers' intuition. Listing 6.2 presents a minimised version of the *Santa Claus* problem, that wasn't described by any of the existing implementations - in fact such combination is prohibited in most of them.

The example 6.2 has a minimally smaller number of lines than the *Erlang* solution in D.1 and much less than the solution provided in *Polyphonic C#*. However the main advantage of our solution is the ability to express joins in lines 13-23 and as such the code is much less error prone. With *JErlang* we are simply able to say: "Synchronise on 9 reindeer or 3 elves, with the priority given to the former". The priority is described here in the most obvious way (lines 14-16 and 20), instead of providing obscure hacks that are hard to maintain (lines 51-58).

The main drawback of this join definition would be a situation when we want to synchronise on a larger number of patterns, say 100 reindeer. Although such an implementation is possible in *JErlang* it is obscure. It is important however to remember that *Join-calculus* wasn't designed with such cases in mind, and specialised algorithms will always perform much better in is such scenarios.

We produced a variation on the *Santa Claus* problem that we call *Sad Santa Claus*, where we want to include *orks* that destroy the presents. Synchronisation requires 3 orks to fire but there is a restriction that one of the *orks* has to be "captain". We provide a detailed description and solution in Appendix D.2. Obviously there can be many other synchronisation patterns that closely relate to real applications. This example shows the power of *JErlang* when matching against multiple complicated patterns, which we still found it hard to express in other *Join-calculus* implementations.

6.2.2 Dining philosophers problem

Definition 6.2 (Dining philosophers problem[6])**.** Five philosophers are sitting at a table and doing one of two things: eating or thinking. While eating, they are not thinking, and while thinking, they are not eating. The five philosophers sit at a circular table with a large bowl of spaghetti in the center. A fork is placed in between each philosopher, so each philosopher has one fork to his or her left and one fork to his or her right. As spaghetti is difficult to serve and eat with a single fork, it is assumed that a philosopher must eat with two forks. The philosopher can only use the fork on his or her immediate left or right.

Philosophers' problems has been discussed and implemented with so many variants in many languages so we are not trying to convince ourselves that our solution is better than others. Our aim was to investigate how intuitive the solution written in *JErlang* could be. For evaluation we compare it with a solution written entirely in standard *Erlang*[7]. In terms of lines of code, the core solution has been reduced from around 100 lines to less than 30. We believe that the *JErlang*solution is more readable and provides less possibilities for race conditions. Example 6.3 uses a strategy where the waiter serves as the synchronisation point between the philosophers. In order to eat, they first make an order to the waiter and wait for the forks to be brought. The waiter is able to handle the order whenever the join in line 29 is

[6]see http://en.wikipedia.org/wiki/Dining_philosopher's_problem
[7]see http://rosettacode.org/wiki/Dining_philosophers#Erlang

```
1  -module(jerlang_santa_claus_minimal).
2  -export([start/0]).
3
4  -ifdef(use_joins_vm).
5  -compile({parse_transform, jerlang_vm_parse}).
6  -else.
7  -compile({parse_transform, jerlang_parse}).
8  -endif.
9
10 santa() ->
11     io:format("It was a long night. Time to bed~n"),
12     Group =
13     receive
14         {reindeer, Pid1} and {reindeer, Pid2} and {reindeer, Pid3}
15           and {reindeer, Pid4} and {reindeer, Pid5} and {reindeer, Pid6}
16           and {reindeer, Pid7} and {reindeer, Pid8} and {reindeer, Pid9} ->
17             io:format("Ho, ho, ho! Let's deliver presents!~n"),
18             [Pid1, Pid2, Pid3, Pid4,
19              Pid5, Pid6, Pid7, Pid8, Pid9];
20         {elf, Pid1} and {elf, Pid2} and {elf, Pid3} ->
21             io:format("Ho, ho, ho! Let's discuss R&D possibilities!~n"),
22             [Pid1, Pid2, Pid3]
23     end,
24     [ Pid ! ok || Pid <- Group],
25     santa().
26
27 worker(Santa, Type, Id, Action) ->
28     generate_seed(Id),
29     worker1(Santa, Type, Id, Action).
30
31 worker1(Santa, Type, Id, Action) ->
32     receive after random:uniform(4000) -> ok end,
33     Santa ! {Type, self()},
34     io:format("~p ~p: Waiting at the gate~n", [Type, Id]),
35     receive ok -> ok end,
36     io:format("~p ~p: ~p~n", [Type, Id, Action]),
37     worker1(Santa, Type, Id, Action).
38
39 generate_seed(Seed) ->
40     {A1, A2, A3} = now(),
41     random:seed(A1+Seed, A2*Seed, A3).
42
43 start() ->
44     Santa = spawn(fun() -> santa() end),
45     [spawn(fun() -> worker(Santa, reindeer, I, " delivering toys.\n") end)
46       || I <- lists:seq(1, 9)],
47     [spawn(fun() -> worker(Santa, elf, I, " meeting in the study.\n") end)
48      || I <- lists:seq(1, 10)].
```

Listing 6.2: Minimal Santa Claus solution in *JErlang*

satisfied, i.e. at the given time he has the left and right fork (in his "kitchen"). Philosophers receive the forks, eat and once finished return the forks to the waiter.

```
1  -module(jerlang_philosophers2).
2  -export([philosopher/3, waiter/2, test/0]).
3
4  -compile({parse_transform, jerlang_parse}).
5
6  philosopher(Number, Name, Waiter) ->
7      philosopher_think(Name),
8      Waiter ! {order, self(), Number},
9      {F1, F2} =
10         receive
11             {forks, Left, Right} ->
12                 {Left, Right}
13         end,
14     philosopher_eat(Name),
15     Waiter ! {fork, F1},
16     Waiter ! {fork, F2},
17     philosopher(Number, Name, Waiter).
18
19 philosopher_eat(Name) ->
20     io:format("Philosopher ~p eats...~n", [Name]),
21     timer:sleep(random:uniform(1000)).
22
23 philosopher_think(Name) ->
24     io:format("Philosopher ~p thinks...~n", [Name]),
25     timer:sleep(random:uniform(1000)).
26
27 waiter(Name, Size) ->
28     receive
29         {order, Phil, Number} and {fork, Left} and {fork, Right}
30             when ((Left - 1) rem Size) == Right ->
31         Phil ! {forks, Number, (Number -1) rem Size}
32     end,
33     waiter(Name, Size).
34
35 test() ->
36     Size = 10,
37     W = spawn(?MODULE, waiter, [adam, Size]),
38     [spawn(?MODULE, philosopher, [Id, Id, W]) || Id <- lists:seq(1, Size)],
39     [W ! {fork, Id} || Id <- lists:seq(1, Size)],
40     receive
41     ok ->
42         nothing
43     end.
```

Listing 6.3: Minimal solution to dining philosophers' problem in *JErlang*

Admittedly, the most beautiful code solving the dining philosophers in *JErlang* would involve distributed joins. Each fork could then be a separate *channel* and the joins would synchronise the calls to the pairs of left and right forks. A similar idea is presented in a no longer maintained *VODKA*[33] programming language that supported distributed joins.

6.3 Scalability

Joe Armstrong defined a challenge that is supposed determine the degree of concurrency of the language[8]:

- Put N processes in a ring
- Send a simple message round the ring M times
- Increase N until the system crashes

Clearly, the problem is biased towards *Erlang* due to its ability to create large number of lightweight processes. Also in the original form it is more useful for testing the languages that introduce the concept of actors. The problem is artificially constructed and does not really reflect the challenges of the real world applications. Nevertheless, we decided to adapt this challenge to our extension in a following way:

- Put N processes in a ring.
- Associate each process with three consecutive neighbors.
- Send synchronisation messages to the last two neighbors and the main, simple message to the first neighbor.
- Each process performs join on the main message and the two synchronisation messages coming from two different processes.
- The main message needs to be sent in M rounds.
- Increase N until the system crashes.
- Increase M to see how long does it take to process the message.

The point that we were trying to investigate was how joins behave under heavy load. It often happens that libraries that work well for small problems behave completely unexpectedly for bigger problems. Our modified problem has to handle an increased number of messages in the VM (each process sends 2 more messages) as well as run the joins. To simulate the real situation we introduce random distributed delays (between 1 and 1000 milliseconds) for the processes.

Figure 6.2 presents the result of running the modified Joe Armstrong's challenge in *JErlang* without the support from VM changes. As we can see the system remains in a relatively stable state and there are no sudden peeks in the performance of the test. Although we do not show on the figure, the system remains stable up to around 45 000 to 50 000 processes running concurrently with synchronisation, when it eventually crashes[9]. This result remains

[8] see `http://ll2.ai.mit.edu/talks/armstrong.pdf` for the original definition of the problem

[9] The results vary depending on the version of *Erlang*, the operating system, the available memory, CPU etc.

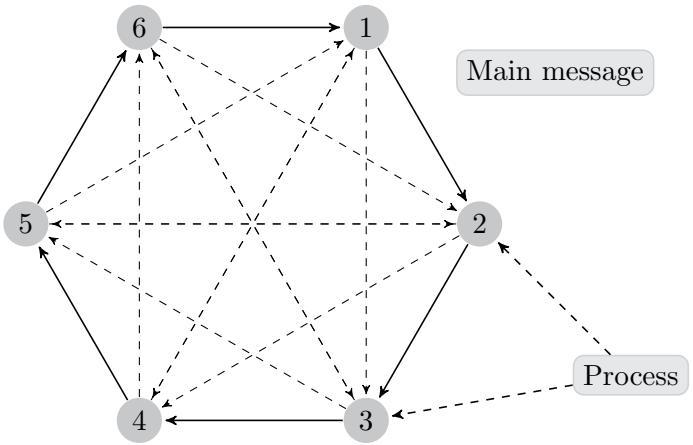

Figure 6.1: Variation on Joe Armstrong's challenge. Each process sends two asynchronous messages to neighbours (dashed lines) and enters join patterns that consist of the main message (sending represented by solid lines), and two synchronisation patterns received from neighbours (on its left)

consistent with other experimental Actor implementations like AiR[10] or STAGE[11]. Obviously the execution time of our solution is larger due to the increased number of messages and synchronisation.

Figure 6.3 shows the results of running the same set of tests on the same challenge, but using also the VM changes that we implemented for *JErlang*. Not surprisingly this version gives much better results for a large number of processes and rounds of the message circulation than the previous attempt. We believe that this is also another indicator of usefulness of implementing VM-accelerated joins. We predict that the increase in the performance would be even bigger if the joins were fully implemented using **BEAM** instructions. Since the join operation is performed in sequence we decided to modify the problem even further, by introducing deliberate delays between the sending of messages and receiving. Obviously, the non-determinism of delays excludes measuring the time as a correct metric, but it was interesting to see whether the system remains stable for large number of processes.

The results reassure that the decision to support joins using the changes inside the VM was correct, because it keeps the system stable and performs the join calculations in a much quicker way. It is important to note that our test is biased towards rather small mailboxes and does not reflect all the situations that can happen[12].

[10] see http://www3.imperial.ac.uk/pls/portallive/docs/1/45405697.PDF
[11] see http://www.doc.ic.ac.uk/teaching/projects/report-10.pdf
[12] see 6.4.4 for further discussion

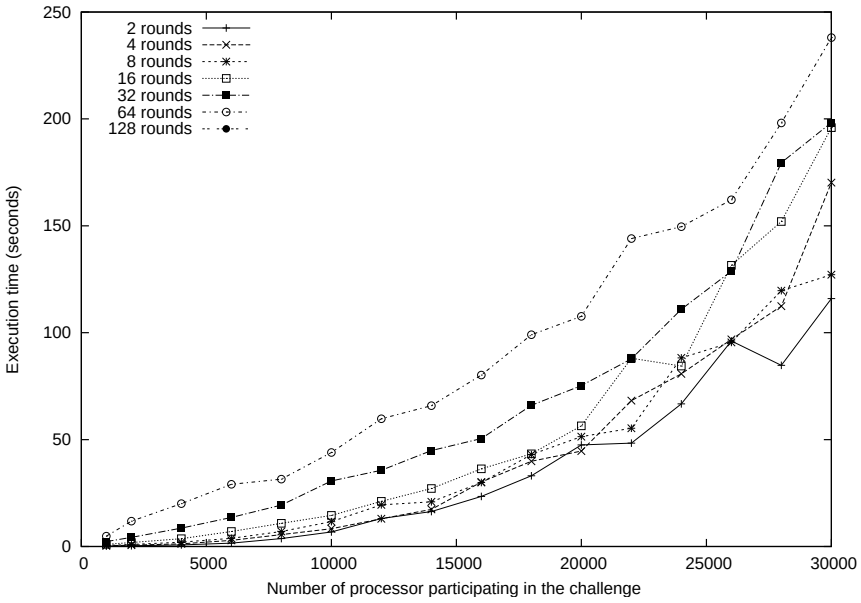

Figure 6.2: Performance of modified Joe Armstrong's challenge with synchronisation between multiple processes in non-VM *JErlang*

6.4 Performance

In the previous section we have noticed a significant boost in the VM-supported *JErlang*. Now we want to focus more on the advantages and disadvantages of both approaches (non-VM and VM *JErlang*) by comparing the performance in different scenarios.
We will also outline the impact of our optimisations on the overall state of the implementation.

6.4.1 Small mailboxes with quick synchronisation

In order to compare the non-VM and VM supported *JErlang* we developed a new **gen_joins** version of the *Santa Claus* problem as given in the Appendix D. The implementation presents the same possibilities as example 6.2 with an additional synchronisation at the point when animals finish their work with santa i.e. all elves have discussed R&D possibilities and all reindeer delivered presents and are free to go on holiday[13].

Figure 6.4 presents an average time of execution of the *Santa Claus* problem in a standard version of *Erlang*, *JErlang*with and without the VM support. Clearly the time execution of the simulation increases linearly with the number of required synchronisations. The execution of the program in the VM-supported *JErlang* surprisingly performs on average similarly as the non-VM version. This underlines the fact that the speedup that we would expect in this situation (as in section 6.3) is not as big. It is interesting that the *JErlang* implementation

[13]see D.2 for details

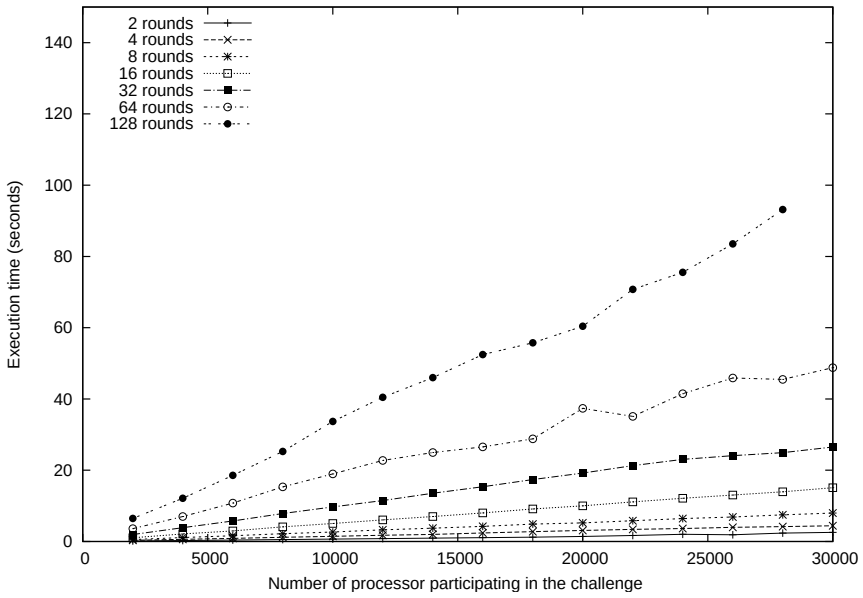

Figure 6.3: Performance of modified Joe Armstrong's challenge with synchronisation between multiple processes in VM-accelerated *JErlang*. Note different scale in comparison to previous results

is faster than the one implemented using usual *Erlang*. This is most probably a result of some of the optimisations that we do during the code analysis.

6.4.2 Queue size factor in joins synchronisation

One of the main culprits of slow performance in *Erlang*applications, apart from possibly bad design, are large process mailboxes. Scanning the mailbox in search for the message that satisfies a pattern is an expensive operation already with the original language[14]. Unfortunately, this also seems a bad case for our implementation of joins, or actually any implementation that allows for as much freedom as *JErlang*. In order to check the behaviour of our patterns solving algorithm, which does a reasonably large amount of work behind the scenes, we designed tests that increasingly build up the mailbox and in which the synchronisation messages are sent randomly. To get proper results our randomisers use a known *seed* and we get consistent and pseudo-random[15] numbers.
Appendix E presents a **gen_joins** module used for creating such environment. It is important to note that we define two main join patterns which share the synchronous message **notify**. Our tests measure the number of times the synchronous messages are replied in a limited amount of time. The number of processes that constantly send asynchronous messages is relatively large, and the diagrams reflect the effect of increasing this number on the overall

[14]see for example at http://www.lshift.net/blog/2007/10/01/too-much-mail-is-bad-for-you
[15]see http://erlang.org/doc/man/random.html for details on **random** module

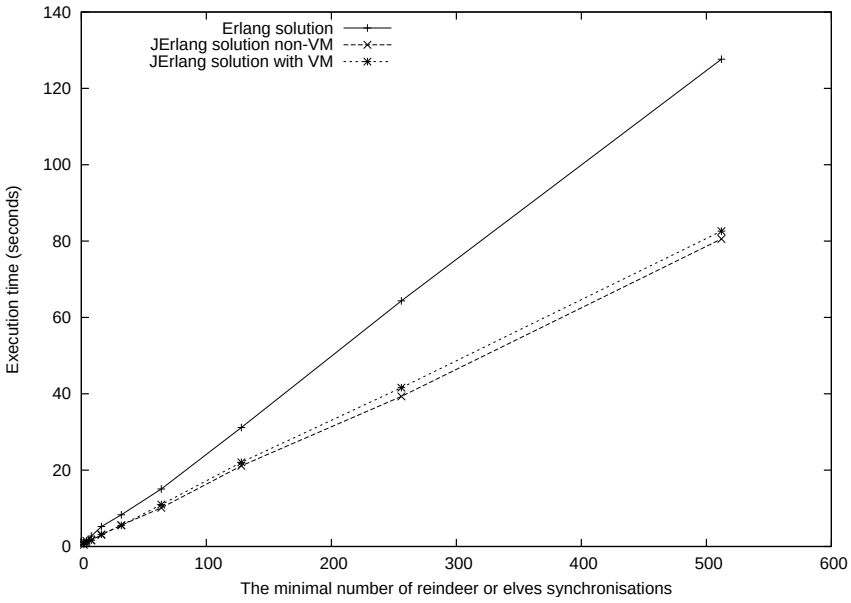

Figure 6.4: Execution of the *Santa Claus* problem in **gen_joins**, implemented on standard *Erlang* as well as *JErlang* with and without VM support. The problem artificially limits the synchronisation to measure the correct time

performance of the joins solver[16].
The benchmark artificially throttles the execution of the processes by putting them into "sleep" for a randomly distributed amount of time from 0 to 1000. We introduce this delay for each process after successful send of the message (either synchronous or asynchronous) to simulate the pseudo-non-deterministic behaviour of the environment. It is therefore important to note that this scenario is created artificially, yet we believe that it presents interesting, close to real, situations.

Figure 6.5 presents rather disappointing performance results, where the number of processes sending the synchronous messages is equal to 10 and the number of synchronisation actions reduces drastically as the number of messages increases (with a larger number of processes sending asynchronous messages). The sudden peak in the number of synchronisation actions happens due to the relatively small number of processes creating asynchronous messages. The joins solver is able to handle quite efficiently the situation when messages do not hit the mailbox at a lower rate. The performance drops even though the messages match the correct patterns and pass the partial tests. However **notify** messages happen less frequently (in comparison to the size of the mailbox) and many of the final beta reductions, which are computationally quite expensive operations, will fail.

In figure 6.6 we increased the number of processes creating synchronous **notify** messages (necessary to fire any join). It can be seen that sudden peeks in successful joins reduce again

[16]In an ideal situation an increase in the number of synchronous call would linearly increase the number of successful joins actions

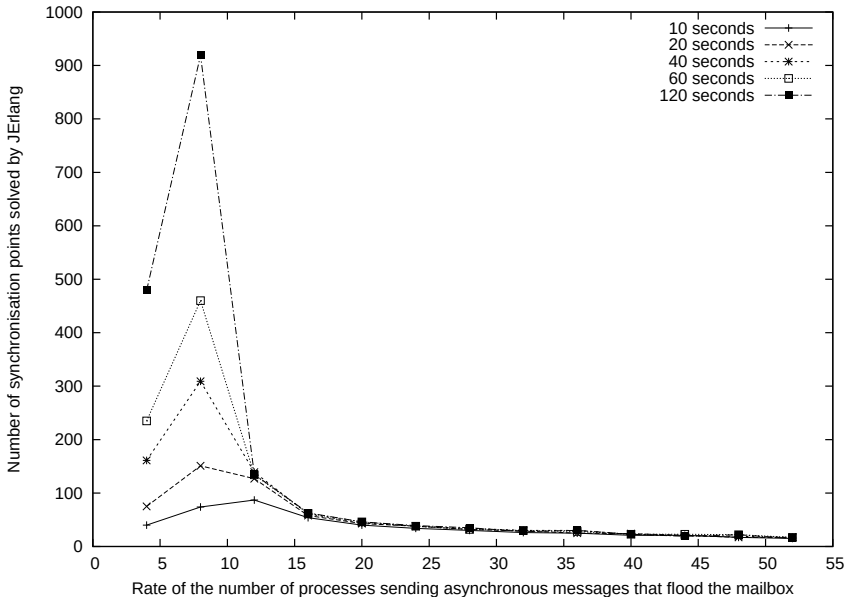

Figure 6.5: Effect of increasing the number of processes producing asynchronous messages on the amount of synchronisation points that the joins solver is able to analyse in a given amount of time. The number of processes creating the synchronous messages is 10.

later due to the increase in the size of the mailbox and the number of computations that have to be done each time a message is received.

For comparison we also include figure 6.7 which presents the results of matching in a benchmark similar to the previous ones, with the exception of processes that used to create messages solely directed for the second join. This creates a situation where we have much lower network load and the mailbox remains smaller than in previous scenarios). We believe that our experiments suitably predict the behaviour of the joins, where patterns are inherently dependent upon each other.

Our tests were performed on the non VM-accelerated version of *JErlang* and the one with VM changes included gives similar results. As we have later found out the fact that the code is compiled using our optimisation for joins ordering and therefore the real ordering of patterns in the joins is significantly different, is very important. We discuss this influence on the drop in the performance of our joins in the next section.

6.4.3 Influence of joins ordering on the performance

Our ordering optimisation was originally focused on improvement of large joins. We have designed a few evaluation tests where biased, complex joins are hammered by the messages that match the first patterns. Without the ordering the performance drops quite radically and contributes to the problem of large numbers of comparisons done for the beta reductions. With the ordering optimisation on, the memory burden is reduced and joins are

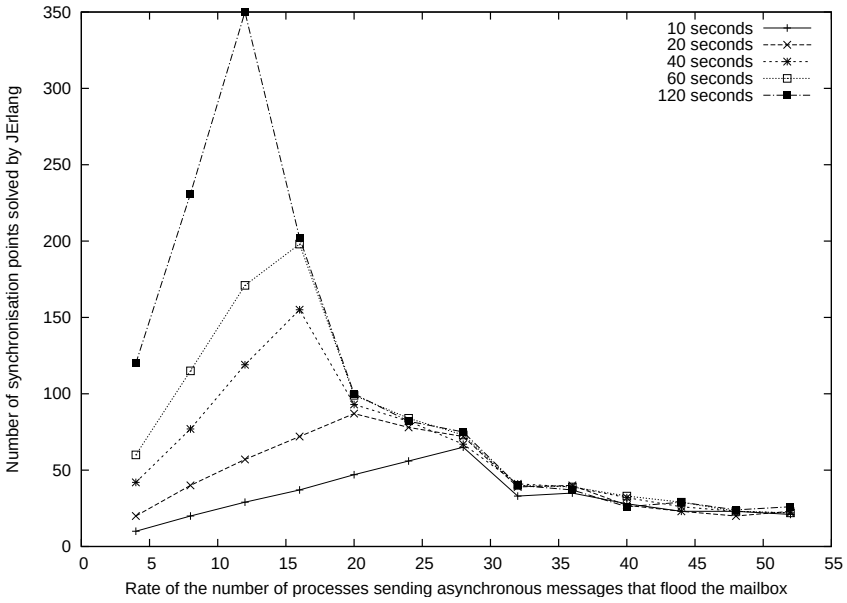

Figure 6.6: Effect of increasing the number of processes producing asynchronous messages on the amount of synchronisation points that the joins solver is able to analyse in a given amount of time. The number of processes creating the synchronous messages is 40.

resolved quicker. We do not produce any numerical results as they represent the predictable improvement.

Nevertheless we were initially surprised when we ran the benchmark from the previous section on the code that was compiled with the joins ordering optimisation turned off. For clarity listing 6.4 presents the definition of the module as stated in the code and listing 6.5 presents the real order of patterns after the compiler's optimisation.

```
1 handle_join(notify and #packet{value=V1, id=Id} and {buy, Id, _, _Previous},
2                 State) ->
3             {[{reply, {ok, buy, Id}}, noreply, noreply], State - V1};
4 handle_join(notify and {deposit, V1} and #packet{value=_, id=Id}
5                 and {secure, Id, _}, State) ->
6             {[{reply, {ok, sell, Id}}, noreply, noreply, noreply],
7                 State + (V1)};
```

Listing 6.4: Joins definition in **gen_joins** from example E.1 as defined by the programmer

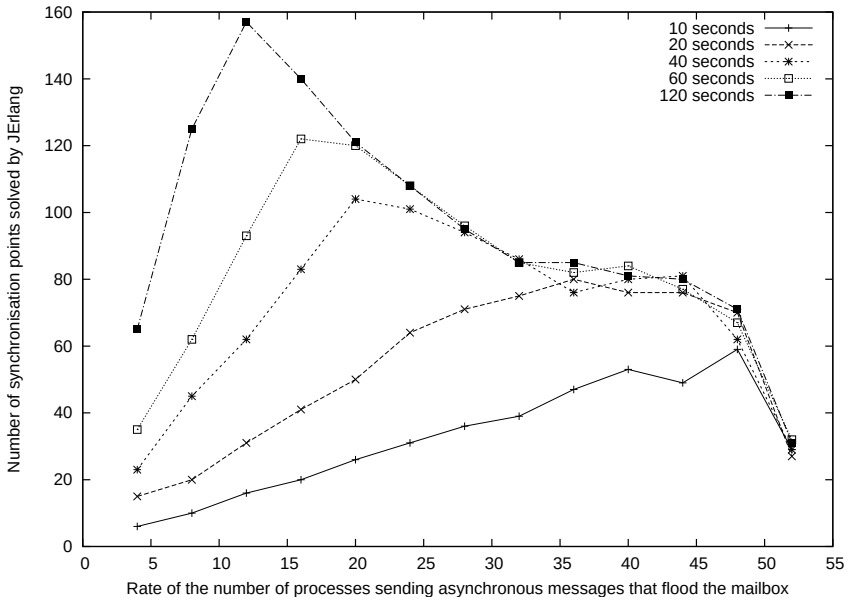

Figure 6.7: Effect of increasing the number of processes producing asynchronous messages on the amount of synchronisation points that the joins solver is able to analyse in a given amount of time. The number of processes creating the synchronous messages is 40. In comparison to 6.6 the test doesn't produce messages that are strictly related to the second join. This lowers the network load and the size of the process' mailbox

```
1 handle_join(#packet{value=V1, id=Id} and {buy, Id, _, _Previous} and notify,
2                 State) ->
3             {[noreply, noreply, {reply, {ok, buy, Id}}], State - V1};
4 handle_join(#packet{value=_, id=Id} and {secure, Id, _} and {deposit, V1}
5                 and notify, State) ->
6             {[noreply, noreply, noreply, {reply, {ok, buy, Id}}],
7                 State + (V1)};
```

Listing 6.5: Optimised join definition in **gen_joins** from example E.1

Knowing how the beta reductions for joins are built we can better understand the origins of the low performance presented in the previous section. We consider only the first join, the situation for the second is the same. The first beta reduction test in the first example checks two patterns: **notify** and **#packet{value=V1, id=Id}**. Since the former is received less frequently, in almost all cases by the time the **notify** message is processed we can easily find a successful sequence of messages.
For the second example however **notify** pattern is only in the last beta reduction test and joins solver spends a lot time performing computations that will often fail due to the infrequent **notify** messages.

Figures 6.8 presents results of our benchmark with the joins ordering turned off. Therefore we managed to show that changing the order of patterns in joins, where joins solver

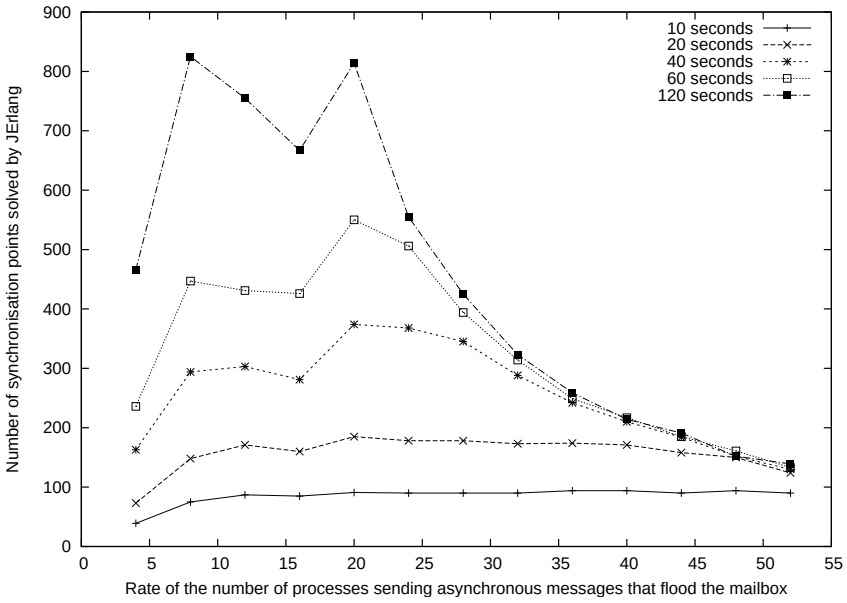

Figure 6.8: The effect of turning off the joins ordering optimisation on the overall performance of the 6.5 benchmark. Note the different scale used between the two figures.

implements RETE algorithm, can lead to unsatisfactory performance. Nevertheless it is important to remember that optimisation is a useful in some scenarios. By default we turned it off, however *JErlang*'s library can be simply recompiled with the **ordering_on** flag in order to turn it on. We leave it to the programmer to observe the different behaviour and decide which one is better on a case by case basis.

6.4.4 Standard queues vs hash-map accelerated queues

The empirical results have shown that the difference between using non-VM *JErlang* and a hash-map data structure inside VM is negligible for small mailboxes. However the latter handles larger mailboxes with frequent joins partial tests more efficiently. We have not managed to produce any reasonable benchmark that would solely test this optimisation simply because of lack of time. An interesting subject for future research would be to investigate the effect of other data structures for *JErlang*'s mailbox[17].

6.5 Applications

The presented examples have shown the advantages and disadvantages of using joins in small problems. In order to fully evaluate the capabilities of the *JErlang* extension it is necessary to look at real examples where it would be useful. Surprisingly, one of the hardest things in our work was to find bigger problems that could be expressed using *JErlang* in a more

[17]Possibly the standard *Erlang* could also gain from such research

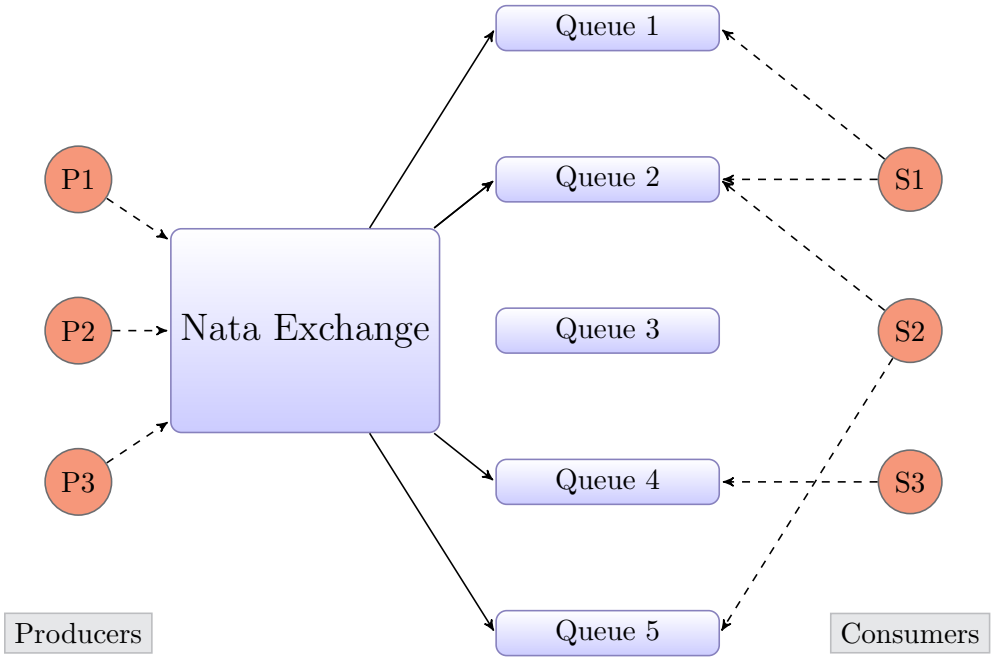

Figure 6.9: Overview of the *NataMQ* architecture

elegant way. We present two minimised examples that show some of the applicability of joins.

6.5.1 Message routing system with synchronisation on messages

Our main application is inspired by the successful open-source product *RabbitMQ*[18]. It is an implementation of *Advanced Message Queuing Protocol*(*AMQP*)[19], an emerging standard for high performance messaging. *RabbitMQ* is fully written in *Erlang* and thanks to its stability and performance has a growing community of users. Our initial attempt was to find a way to incorporate *JErlang* semantics into the application, but we decided that it would be hard to isolate parts of the system.
Therefore we implemented our own, *AMQP*-inspired implementation with a minimal set of basic concepts taken from the protocol. Figure 6.9 presents a simplified architecture of our routing-oriented system, that we called *NataMQ*.

We base our example on a typical producer-consumer schema, with additional semantics that are implemented in the *Nata Exchange*:

- Producers send messages independently to the *Nata Exchange*. Every message should contain a *routing key*, an *id* and a *value*.

[18] http://www.rabbitmq.com
[19] http://www.amqp.org

- Each *Nata Exchange* has a set of *Nata Queues* associated with it.
- Consumers can request the creation of *Nata Queues* and define routing algorithms to be used when receiving messages with specific keys.
- *Nata exchange*, apart from using other *JErlang*features, defines the synchronisation on messages that it receives from the consumers.
- The exchange allows for creation of routing rules that synchronise two or three messages with the same routing key.
- Messages are only passed to the queues when the routing definition on the key is defined and the join definition is successful.
- Consumers receive messages from the *Nata Queues* to which they are subscribed.

The critical part of our implementation of *NataMQ* is the synchronisation on messages, which are only passed to the queues when the conditions are satisfied. *AMQP* defines different types of routing for the messages, and we felt that our minimised version nicely fits into the general protocol image. We only provide two types of routing, but the real power of our messaging system lies in its extensibility. Programmers using *JErlang* could easily adapt the routing strategies to their needs, which makes the system customisable. For instance, any message can be passed to the queues, when it synchronises with the respective *control message* that caries necessary privileges to publish the original one.

In the traditional system it is the programmers' responsibility to define the logic for retrieving the messages, the algorithms for finding matches between the necessary messages etc. We believe that the *JErlang* solution helps in the general maintenance of the problem.

Currently, the producers and consumers are only written in *Erlang*and exchange the messages using VM infrastructure. Time restrictions didn't allow us to write *Scala* or *Python* libraries that would subsume their roles and communicate with the *NataMQ* through standard TCP/IP sockets (altough this is irrelevant from the perspective of testing the usability of *JErlang*).

6.5.2 Chat system

A typical chat system would be written using the client-server model. Thanks to **gen_joins** behaviour we can implement a minimised chat applications that specifically uses *JErlang* features to maintain and manipulate the state of the system. It is important to notice that our implementation does not introduce any functionality that would not be expressible in the traditional *Erlang*but synchronisation scenarios are much easier to understand. The listing of the code is given in the Appendix C.

In order for the client to send messages, he has to first log-in to the system. Any messages sent to the system are routed to all the users currently available. But users have the opportunity to create private chat rooms for having non-public conversation.

The server maintains the list of all users currently available, each action requires authentication check. The creation of private chat rooms is allowed by awaiting for the mutual agreement between the two users. These are expressed in terms of synchronous call to function **private_channel** and we use the barrier pattern in order to wait for the corresponding message (line 50). The state of the private channel is persisted through the propagation that needs synchronisation with any message sent to the private channel. When any of the users requests to close the room, we synchronise on the message as well as the room number. This uses a typical style of propagation, where messages can be treated as session indicators.

We can see here the advantage of having a sequential joins execution property in order to avoid writing spurious join definitions (lines 66 and 80). Whenever a "close" message is sent in relation to a private channel it is given a special meaning i.e. it does not just route the value to the other user, but treats it as a system message. Such an assertion cannot be made in the case of non-deterministic execution that is made by most of the current *Join-calculus* implementations. This results in a more maintainable code.

6.6 Conclusion

In this chapter we have evaluated *JErlang* from different points of view. The language allows for the creation of complex constructs using intuitive (for *Erlang* programmers) semantics. We presented many examples where the solution provided in our extension is more elegant than in many of the existing implementations. This helps in performing rapid software development without losing clarity in the code. We critically assessed the applicability of *JErlang*'s features and can say that it offers features that are useful in some applications, but not in all.

In our tests we also focused on an important issue of performance. We managed to build benchmarks that reliably check the efficiency of the new construct. The performance of the solver when put under a heavy stress is still a serious issue, but some of the techniques that we used helped to reduce this effect. Additionally, one of the optimisations results in awkward behaviour when tested in different scenarios.

We believe we managed to show the usefulness of the language in real applications. The most valuable feedback on *JErlang* as a language would be that of other developers that would like to use it in their current or future applications.

Chapter 7

Conclusion

This report presents our work done on extending the *Erlang* language with Joins constructs. The visible results of the project are a formalised syntax and semantics of minimal *JErlang*, a stand-alone *JErlang* library and a *JErlang*library that uses modified a *Erlang*compiler and Virtual Machine.

Chapter 2 introduces the paradigm of *Join-calculus* and explains its relation to current concurrency problems. I explore several available languages, each of which adapts the *Join-calculus* theory to its existing infrastructure. Nevertheless I do not limit myself to only joins-like languages but also explore the concept of *Constraint Handling Rules*which also influenced *JErlang*. This work enabled me to explore the possible usage scenarios for the joins construct. Therefore the initial scepticism about joins being "just another simple feature" was quickly turned down.

When I started the project I had minor experience with *Erlang*, having worked on small pieces of the *RabbitMQ* project. This project enabled me to fully explore the possibilities of the language and investigate to what extent it helps in the development of concurrent applications. Finally I managed to lose myself in the internals of *Erlang*'s compiler and Virtual Machine, while searching for a possibility of manipulating processes' internal mailboxes. All of the *Erlang* and *JErlang* programs were developed using *emacs* and thanks to the compatible syntax of the latter and minor tweaks in the editor I did not have to use the "plain old" notepad.

The work in the chapter 3 that formally defines the syntax and semantics of the minimal *JErlang* allowed me to precisely formulate the behaviour that I later wanted to convert into a usable language. I decided to investigate the formal approach in order to create a complete language. None of the current *Join-calculus* implementations provide formalism of their joins[1]. My intent was to provide enough basis for the future work on, for example, language verification tools. The two approaches to solving the joins serve as a starting point for the

[1] *HaskellJoinRules* is an exception to this statement but its main focus is on *Constraint Handling Rules*, rather than *Join-calculus*

implementation of *JErlang*'s joins. Although the existence of a formalism in my extension does not increase the chances of it being adopted by the community, I believe it makes harder to reject immediately by the usual programmers and theoretical-oriented academics.

The pure Actor-Oriented approach for building sophisticated synchronisation constructs is often inadequate, therefore it seemed natural to provide better support for such features in *Erlang*. Chapter 4 provides detailed information about all the constructs available with the introduction of joins.
Erlang programmers find themselves repeatedly constructing algorithms that try to match against multiple patterns. The introduction of familiar **receive** feature but with more powerful semantics was better than creating something completely new. It was surprising that given the increased interest in *Join-calculus* no one from the *Erlang* community implemented it before. I decided to support a range of new features, unlike what was done with *chords* in *Polyphonic C#*, because timeouts or guards are an integral part of the standard language. This not original, but definitely a useful idea, which I later improved by the novel approach of non-linear patterns available during synchronisation. The latter is commonly used in *Erlang* standard expressions, therefore it did not make sense to artificially limit the language, like in for example *JoCaml*. With the design of new Open Telecom Platform design pattern, programmers are able build client-server applications in a familiar environment.
I turned down the initial idea of supporting distributed joins due to the inefficiency of the implementation in the *Erlang*'s message passing architecture. I believe that there was no point in developing an extension that would contain theoretically interesting properties but would be disastrous in terms of efficiency.

The increased expressiveness presents challenging performance problems. This was one of the main subjects discussed in chapters 5 and 6, where I attempt to define joins algorithm that would perform well in most of the scenarios. The decision to use the RETE implementation came partially from the fact that OTP behaviour presents the possibility of maintaining some information about past matching results. As the actual performance impact was undecided I deferred the decision to support this algorithm in the standard **receive** construct. This gave me opportunity to experiment with the implementation. Especially interesting was the *Santa Claus* problem from chapter 7 that has influenced the design of joins since the beginning of the project.
Some performance drawbacks in the joins solving algorithm that I describe in chapter 6, were inevitable but I believe that these (initial) costs had to be taken in order to better understand the nature of the problem. The possibility of working on Abstract Syntax Trees through the powerful **parse_transform** module instead of directly changing the compiler, created many interesting opportunities for optimising the code and allowed for the existence of a stand-alone *JErlang*library. I expect that static code analysis performed using this technique has even more potential than what I have already done. The novel changes, from the perspective of the *Erlang* architecture, that I decided to implement within the Virtual Machine allow not only for the combined VM/library *JErlang*implementation but also for direct manipulation of the mailboxes available to the programmers. Many *Erlang*developers have complained in the past about lack of this feature from the level of the language.

It is important to remember that *JErlang* is not a remedy for the all the concurrency problems programmers are having nowadays. It can be often misused, like any other typical

language feature. However when used properly *JErlang* solves non-trivial synchronisation problems in two or three lines.

7.1 Further work

I believe that development of *JErlang* presents many interesting and challenging opportunities which I have not been able to accomplish due to the time constraints.

7.1.1 Formal definition

One of the important features that I have not maneged to do, was to formally define a mapping between the standard *Erlang* and *JErlang*. In other words, we would like to show the theoretical equivalence of both languages. Although intuitively we can say that this mapping exists, because I have just implemented the extension of *Erlang*, it is non trivial to prove it formally. This would require the definition of *Erlang* similar to what I have done in chapter 3.

7.1.2 Performance

The biggest drawback of the current implementation is the unpredictability of joins solver under heavy load. Although the scenario of large mailbox is often a result of bad design we would like to make sure that the general performance is much better. Some of the possible research work includes:

- Further optimisations of the current sequential algorithm by searching for the regular patterns inside the joins. Unfortunately we risk here to get into the infinite loop of minimal optimisations.
- In the current *JErlang*implementation we lose quite a lot of information after a successful execution of a join. In the RETE algorithm for joins we only leave the *alpha memory*. It would be very useful to extend the algorithm for checking whether the internal state of the joins has actually changed and which *beta memory* needs to be rebuilt (if any).
- Identification of the impact of different data structures on the implementation of *JErlang*'s mailboxes (inside the VM and library). For instance usage of trees instead of having the orthogonal concepts of hash map and queue.
- Introduction of a parallel joins solver at the cost of losing some of the expressiveness, like out-of-order execution of joins. An interesting research is done in this area by Sulzmann and Lam in [23] and [22] who implement parallel algorithm for solving *Constrain Handling Rules*. Unfortunately they immediately lose many features used in *JErlang*.
- Implementation of the joins solver inside the Virtual Machine - a possible, but unlikely proposition since it would require the redesign of a large part of *Erlang*'s compiler/VM.

An interesting investigation could be performed to find out a reasonable trade-off between *JErlang*'s performance and expressiveness. Unfortunately setting restrictions on the language features is a controversial action once programmers get used to its powerful constructs.

7.1.3 New Erlang release

At the time of writing, a new version of *Erlang* was released. *Erlang* R13B-0 contains many improvements in the region of mailboxes and general concurrency issues. The current *JErlang* was only designed to work with R12B-5 so it would be interesting to examine any differences between the two in the context of our extension.

7.1.4 Benchmarks

The final proposition relates to the general problem of testing not only in the context of *JErlang* but in terms of all implementations. I believe that it is possible to define a complete or partial set of benchmarks that compare performance of different approaches. The tests could be used for measuring the effectiveness of the optimisations used in the algorithms. The harder part is to create benchmarks that are close to real applications in behaviour.

7.2 Closing remarks

JErlang presents interesting and powerful semantics for the programmers. Judging from its ability to perform elegant solutions for complex concurrency problems, acceptable performance and simplicity, it can be a successful extension of *Erlang*. All in all the end result of the development of any language is to help programmers to build more reliable, stable and efficient applications. I believe that this project achieved this goal.

Bibliography

[1] Martín Abadi and Andrew D. Gordon. A calculus for cryptographic protocols: The spi calculus. In *Fourth ACM Conference on Computer and Communications Security*, pages 36–47. ACM Press, 1997. [cited at p. 6]

[2] Joe Armstrong. A history of erlang. In *HOPL III: Proceedings of the third ACM SIGPLAN conference on History of programming languages*, pages 6–1–6–26, New York, NY, USA, 2007. ACM. [cited at p. 27, 32]

[3] Joe Armstrong. *Programming Erlang: Software for a Concurrent World*. Pragmatic Bookshelf, July 2007. [cited at p. 31]

[4] Don Batory. The leaps algorithm. Technical report, Austin, TX, USA, 1994. [cited at p. 38]

[5] Nick Benton, Luca Cardelli, and Polyphonic C. Modern concurrency abstractions for c. In *ACM Trans. Program. Lang. Syst*, pages 415–440. Springer, 2002. [cited at p. 14, 17, 18, 84]

[6] Gerard Berry and Gerard Boudol. The chemical abstract machine. In *POPL '90: Proceedings of the 17th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 81–94, New York, NY, USA, 1990. ACM. [cited at p. 6]

[7] Polyphonic C and Nick Benton. Jingle bells: Solving the santa claus problem. In *in Polyphonic C*. [cited at p. 98, 147, 148]

[8] Sophia Drossopoulou, Alexis Petrounias, Alex Buckley, and Susan Eisenbach. SCHOOL: a Small Chorded Object-Oriented Language. *Electronic Notes in Theoretical Computer Science*, 135(3):37–47, March 2006. [cited at p. 19]

[9] Erlang. Core erlang, 2000-2009. [cited at p. 73]

[10] Cédric Fournet, Cosimo Laneve, Luc Maranget, and Didier Rév. Inheritance in the join-calculus (extended abstract). In *In FST TCS 2000: Foundations of Software Technology and Theoretical Computer Science. Lecture Notes in Computer Science*, pages 397–408. Springer-Verlag, 2000. [cited at p. 16]

[11] Thom Frühwirth. Introducing simplification rules. Technical Report ECRC-LP-63, European Computer-Industry Research Centre, Munchen, Germany, October 1991. Presented at the Workshop Logisches Programmieren, Goosen/Berlin, Germany, October 1991 and the Workshop on Rewriting and Constraints, Dagstuhl, Germany, October 1991. [cited at p. 23]

[12] Thom Frühwirth. Theory and practice of Constraint Handling Rules. *J. Logic Programming, Special Issue on Constraint Logic Programming*, 37(1–3):95–138, 1998. [cited at p. 23]

[13] David Kearney G. von Itzstein. Join java: An alternative concurrency semantic for java. Technical report, University of South Australia, 2001. [cited at p. 21]

[14] M. Jasiunas G. von Itzstein. On implementing high level concurrency in java. Advances in Computer Systems Architecture, Springer Verlag, 2003. [cited at p. 20]

[15] Georges Gonthier and Inria Rocquencourt. The reflexive cham and the join-calculus. In *In Proceedings of the 23rd ACM Symposium on Principles of Programming Languages*, pages 372–385. ACM Press, 1996. [cited at p. 2, 7, 8, 63]

[16] Philipp Haller and Tom Van Cutsem. Implementing Joins using Extensible Pattern Matching. In *10th International Conference on Coordination Models and Languages*, Lecture Notes in Computer Science, pages 135–152. Springer, 2008. [cited at p. 84, 85]

[17] C. A. R. Hoare. Communicating sequential processes. *Commun. ACM*, 21(8):666–677, 1978. [cited at p. 6]

[18] Christian Holzbaur, Mara Garca de la Banda, David Jeffery, Peter J. Stuckey, and Peter J. Optimizing compilation of constraint handling rules, 2001. [cited at p. 93]

[19] S. R. Virding J. Barklund. *Erlang 4.7.3, Reference Manual (Draft 0.7)*. Ericsson AB, 1999. [cited at p. 79]

[20] L. Maranger L. Mandel. *JoCaml Documentation and Manual (Release 3.11)*. INRA, 2008. [cited at p. 10]

[21] Edmund Lam and Martin Sulzmann. Finally, a comparison between constraint handling rules and join-calculus. In *Fifth Workshop on Constraint Handling Rules*, CHR 2008. [cited at p. 23, 64]

[22] Edmund S.L. Lam and Martin Sulzmann. Concurrent goal-based execution of Constriant Handling Rules. submitted to *Journal of Theory and Practice of Logic Programming*, 2009. [cited at p. 90, 117]

[23] Edmund S.L. Lam and Martin Sulzmann. Parallel join patterns with guards and propagation. 2009. [cited at p. 90, 117]

[24] John Launchbury and Simon L Peyton Jones. Concurrent haskell. pages 295–308. ACM Press, 1996. [cited at p. 21, 22]

[25] Milind Mahajan and V. K. Prasanna Kumar. Efficient parallel implementation of rete pattern matching. *Comput. Syst. Sci. Eng.*, 5(3):187–192, 1990. [cited at p. 38]

[26] Luc Maranget and Fabrice Le Fessant. Compiling join-patterns. In *Electronic Notes in Computer Science*. Elsevier Science Publishers, 1998. [cited at p. 84]

[27] Satoshi Matsuoka and Akinori Yonezawa. Analysis of inheritance anomaly in object-oriented concurrent programming languages. In G. Agha, P. Wegner, and A. Yonezawa, editors, *Research Directions in Concurrent Object-Oriented Programming*, pages 107–150. MIT Press, 1993. [cited at p. 16]

[28] R. Milner. *A Calculus of Communicating Systems*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1982. [cited at p. 6]

[29] Robin Milner. A calculus of mobile processes, parts. *I and II. Information and Computation*, 100:1–77, 1992. [cited at p. 6]

[30] Daniel P. Miranker. *TREAT: a new and efficient match algorithm for AI production systems*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1990. [cited at p. 38]

[31] Flemming Nielson, Hanne R. Nielson, and Chris Hankin. *Principles of Program Analysis*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1999. [cited at p. 39, 82]

[32] Alexis Petrounias. *On The Design Of Chorded Languages*. PhD thesis, Imperial College London, 2008. [cited at p. 19]

[33] Tiark Rompf. *Design and Implementation of a Programming Language for Concurrent Interactive Systems*. PhD thesis, University of Lbeck, http://vodka.nachtlicht-media.de/index.html, 2007. [cited at p. 101]

[34] Claudio V. Russo. Join patterns for visual basic. In *OOPSLA '08: Proceedings of the 23rd ACM SIGPLAN conference on Object oriented programming systems languages and applications*, pages 53–72, New York, NY, USA, 2008. ACM. [cited at p. 25]

[35] Martin Sulzmann, Edmund S. L. Lam, and Peter Van Weert. Actors with multi-headed message receive patterns. In Doug Lea and Gianluigi Zavattaro, editors, *COORDINATION*, volume 5052 of *Lecture Notes in Computer Science*, pages 315–330. Springer, 2008. [cited at p. 59, 66]

[36] Martin Sulzmann and Edmund S.L. Lam. Compiling Constraint Handling Rules with lazy and concurrent search techniques. pages 139–149, 2007. [cited at p. 85, 92]

[37] Martin Sulzmann and Edmund S.L. Lam. Haskell - join - rules. In Olaf Chitil, editor, *IFL '07: 19th Intl. Symp. Implementation and Application of Functional Languages*, pages 195–210, Freiburg, Germany, sep 2007. [cited at p. 23, 66]

[38] Herb Sutter. The free lunch is over: A fundamental turn toward concurrency in software. *Dr. Dobb's Journal*, 30(3), 2005. [cited at p. 1]

[39] John A. Trono. A new exercise in concurrency. *SIGCSE Bull.*, 26(3):8–10, 1994. [cited at p. 98]

[40] Andrew K. Wright and Matthias Felleisen. A syntactic approach to type soundness. *Information and Computation*, 115:38–94, 1992. [cited at p. 47]

[41] Lars ke Fredlund. *A framework for reasoning about Erlang code.* PhD thesis, 2001. Trita-IT. AVH ; 01:04, URI: urn:nbn:se:kth:diva-3210, SICS dissertation: SICS-D-29. [cited at p. 41]

Appendices

Appendix A

Erlang Compiler Result

A.1 Original code

```
1  -module(simple_receive).
2
3  -compile({parse_transform, simple_parse}).
4
5  -export([start/0]).
6
7  start() ->
8      receive
9      {atomic, Value} ->
10         {ok, Value};
11     {commit, TransId}  ->
12         {ok, {trans, TransId}};
13     _ ->
14         {error, unexpected}
15     end.
```

Listing A.1: Session support using propagation in *JErlang*

A.2 Abstract syntax tree for the first program

```
[{attribute,1,file,{"./simple_receive.erl",1}},
 {attribute,1,module,simple_receive},
 {attribute,5,export,[{start,0}]},
 {function,7,start,0,
     [{clause,7,[],[],
          [{'receive',8,
               [{clause,9,
                    [{tuple,9,[{atom,9,atomic},{var,9,'Value'}]}],
                    [],
                    [{tuple,10,[{atom,10,ok},{var,10,'Value'}]}]},
                {clause,11,
                    [{tuple,11,[{atom,11,commit},{var,11,'TransId'}]}],
                    [],
                    [{tuple,12,
                         [{atom,12,ok},
                          {tuple,12,[{atom,12,trans},{var,12,'TransId'}]}]}]},
                {clause,13,
                    [{var,13,'_'}],
                    [],
                    [{tuple,14,[{atom,14,error},{atom,14,unexpected}]}]}]}]}]},
 {eof,16}]
```

A.3 First program expressed in Core Erlang

```
module 'simple_receive' ['module_info'/0,
 'module_info'/1,
 'start'/0]
    attributes []
'start'/0 =
    %% Line 7
    fun () ->
let <_cor20> =
    %% Line 9
    ( fun (_cor18) ->
  case _cor18 of
    <{'atomic',_X_Value}> when 'true' ->
'true'
    <_cor29> when 'true' ->
'false'
  end
      -| [{'id',{0,41589281,'-start/0-fun-0-'}}] )
in  let <_cor24> =
%% Line 9
( fun (_cor22) ->
      case _cor22 of
<{'test',_X_Value}> when 'true' ->
    'true'
```

```
<_cor30> when 'true' ->
    'false'
      end
  -| [{'id',{1,116202316,'-start/0-fun-1-'}}] )
    in  let <_cor28> =
    %% Line 11
    ( fun (_cor26) ->
  case _cor26 of
    <{'commit',_X_TransId}> when 'true' ->
'true'
    <_cor31> when 'true' ->
'false'
  end
      -| [{'id',{2,59711518,'-start/0-fun-2-'}}] )
in  let <_cor8> =
%% Line 8
( fun (_cor5,_cor4) ->
      case _cor5 of
<%% Line 9
 [{'atomic',Value}|[{'test',_cor32}|[]]]>
    when call 'erlang':'=:='
  (_cor32,
   %% Line 9
   Value) ->
    case _cor4 of
      <'test_entry'> when 'true' ->
  'true'
      <'run_all'> when 'true' ->
  %% Line 10
  {'ok',Value}
      ( <_cor3> when 'true' ->
    primop 'match_fail'
({'case_clause',_cor3})
-| ['compiler_generated'] )
    end
( <_cor1> when 'true' ->
      primop 'match_fail'
  ({'badmatch',_cor1})
  -| ['compiler_generated'] )
      end
  -| [{'id',{3,26522577,'-start/0-fun-3-'}}] )
    in  let <_cor16> =
    %% Line 8
    ( fun (_cor13,_cor12) ->
  case _cor13 of
    <%% Line 11
     [{'commit',TransId}|[]]> when 'true' ->
let <Guard1244312501970288> =
    %% Line 11
```

```
    call 'erlang':'>'
(TransId, 0)
in  case _cor12 of
      <'test_entry'> when 'true' ->
  Guard1244312501970288
      <'run_all'> when 'true' ->
  %% Line 12
  {'ok',{'trans',TransId}}
      ( <_cor11> when 'true' ->
    primop 'match_fail'
({'case_clause',_cor11})
-| ['compiler_generated'] )
    end
    ( <_cor9> when 'true' ->
  primop 'match_fail'
      ({'badmatch',_cor9})
      -| ['compiler_generated'] )
  end
      -| [{'id',{4,94909485,'-start/0-fun-4-'}}] )
in  let <_cor0> =
%% Line 13
( fun () ->
      %% Line 14
      {'error','timeout'}
  -| [{'id',{5,92030033,'-start/0-fun-5-'}}] )
    in  %% Line 8
call 'jerlang_core':'loop'
    ([%% Line 9
      [{1,{'pattern_joins',_cor20,[],'no'}}|[{2,{'pattern_joins',_cor24,[],'no'}}|[]]]|[%% Lir
[{3,{'pattern_joins',_cor28,[],'no'}}|[]]|[]]], [_cor8|[_cor16|[]]], %% Line 13
     [{'timeout',10000,_cor0}|[]])
'module_info'/0 =
    fun () ->
call 'erlang':'get_module_info'
    ('simple_receive')
'module_info'/1 =
    fun (_cor0) ->
call 'erlang':'get_module_info'
    ('simple_receive', _cor0)
end
```

A.4 Assembler code (Kernel Erlang) corresponding to the first program

```
{module, simple_receive}.  %% version = 0

{exports, [{module_info,0},{module_info,1},{start,0}]}.
```

```
{attributes, []}.

{labels, 11}.


{function, start, 0, 2}.
  {label,1}.
    {func_info,{atom,simple_receive},{atom,start},0}.
  {label,2}.
    {loop_rec,{f,6},{x,0}}.
    {test,is_tuple,{f,5},[{x,0}]}.
    {test,test_arity,{f,5},[{x,0},2]}.
    {get_tuple_element,{x,0},0,{x,1}}.
    {get_tuple_element,{x,0},1,{x,2}}.
    {test,is_atom,{f,5},[{x,1}]}.
    {select_val,{x,1},{f,5},{list,[{atom,atomic},{f,3},{atom,commit},{f,4}]}}.
  {label,3}.
    remove_message.
    {test_heap,6,3}.
    {put_tuple,2,{x,1}}.
    {put,{atom,value}}.
    {put,{x,2}}.
    {put_tuple,2,{x,0}}.
    {put,{atom,ok}}.
    {put,{x,1}}.
    {'%live',1}.
    return.
  {label,4}.
    remove_message.
    {test_heap,6,3}.
    {put_tuple,2,{x,1}}.
    {put,{atom,trans}}.
    {put,{x,2}}.
    {put_tuple,2,{x,0}}.
    {put,{atom,ok}}.
    {put,{x,1}}.
    {'%live',1}.
    return.
  {label,5}.
    remove_message.
    {move,{literal,{error,unexpected}},{x,0}}.
    return.
  {label,6}.
    {wait,{f,2}}.

{function, module_info, 0, 8}.
  {label,7}.
    {func_info,{atom,simple_receive},{atom,module_info},0}.
  {label,8}.
```

```
    {move,{atom,simple_receive},{x,0}}.
    {call_ext_only,1,{extfunc,erlang,get_module_info,1}}.


{function, module_info, 1, 10}.
  {label,9}.
    {func_info,{atom,simple_receive},{atom,module_info},1}.
  {label,10}.
    {move,{x,0},{x,1}}.
    {move,{atom,simple_receive},{x,0}}.
    {call_ext_only,2,{extfunc,erlang,get_module_info,2}}.
```

Appendix B

Parse_transform result

B.1 Original code

```
1  -module(simple_receive).
2
3  -compile({parse_transform, jerlang_parse}).
4
5  -export([start/0]).
6
7  start() ->
8      receive
9          {atomic, Value} and {test, Value} ->
10             {ok, Value};
11         {commit, TransId} when (TransId > 0)  ->
12             {ok, {trans, TransId}}
13     after 10000 ->
14         {error, timeout}
15     end.
```

Listing B.1: Selective receive expression in *JErlang*

B.2 JErlang's code resulting from running parse_transform on the first program

```
1  -module(simple_receive_parsed).
2
3  -include("jerlang.hrl").
4
5  -export([start/0]).
6
7  start() ->
```

```
8     jerlang:loop(
9       [
10       [{1, #pattern_joins{test=
11               fun(Value1) ->
12                   case Value1 of
13                       {atomic, _Value} -> true;
14                       _ -> false
15                   end
16               end, msgs=[]}},
17        {2, #pattern_joins{test=
18               fun(Value2) ->
19                   case Value2 of
20                       {test, _Value} -> true;
21                       _ -> false
22                   end
23               end}}],
24       [{3, #pattern_joins{test=
25               fun(Value3) ->
26                   case Value3 of
27                       {commit, _TransId} -> true;
28                       _ -> false
29                   end
30               end, msgs=[]}}]],
31      [fun(Res1, Case1) ->
32           [{atomic, Value}, {test, Value}] = Res1,
33           Ret = true,
34           case Case1 of
35               test_entry ->
36                    Ret;
37               run_all ->
38                   {ok, Value}
39           end
40       end,
41       fun(Res2, Case2) ->
42           [{commit, TransId}] = Res2,
43           Ret = (TransId > 0),
44           case Case2 of
45               test_entry ->
46                   Ret;
47               run_all ->
48                   {ok, {trans, TransId}}
49           end
50       end],
51      [{timeout, 10000,
52                fun() ->
53                    {error, timeout}
54                end}]).
```

Listing B.2: Selective receive expression in *JErlang*

B.3 Abstract Syntax Tree representing the first program

```
[{attribute,1,file,{"src/simple_receive.erl",1}},
          {attribute,1,module,simple_receive},
          {attribute,5,export,[{start,0}]},
          {function,7,start,0,
              [{clause,7,[],[],
                   [{'receive',8,
                        [{clause,9,
                             [{op,9,'and',
                                  {tuple,9,[{atom,9,atomic},{var,9,'Value'}]},
                                  {tuple,9,[{atom,9,test},{var,9,'Value'}]}}],
                             [],
                             [{tuple,10,[{atom,10,ok},{var,10,'Value'}]}]},
                         {clause,11,
                             [{tuple,11,
                                  [{atom,11,commit},{var,11,'TransId'}]}],
                             [[{op,11,'>',{var,11,'TransId'},{integer,11,0}}]],
                             [{tuple,12,
                                  [{atom,12,ok},
                                   {tuple,12,
                                       [{atom,12,trans},
                                        {var,12,'TransId'}]}]}]}],
                        {integer,13,10000},
                        [{tuple,14,[{atom,14,error},{atom,14,timeout}]}]}]}]},
          {eof,17}]
```

B.4 Abstract Syntax Tree resulting from running parse_transform

```
[{attribute,1,file,{"src/simple_receive.erl",1}},
         {attribute,1,module,simple_receive},
         {attribute,1,record,
          {pattern_joins,
           [{record_field,1,{atom,1,test}},
            {record_field,1,{atom,1,msgs},{nil,1}},
            {record_field,1,{atom,1,prop},{atom,1,no}}]}},
         {attribute,5,export,[{start,0}]},
         {function,7,start,0,
          [{clause,7,[],[],
            [{call,8,
              {remote,8,{atom,8,jerlang_core},{atom,8,loop}},
              [{cons,8,
                {cons,9,
                 {tuple,9,
                  [{integer,9,1},
                   {record,9,pattern_joins,
                    [{record_field,9,
                      {atom,9,test},
                      {'fun',9,
                       {clauses,
                        [{clause,9,
                          [{var,9,'Value1243560520536039'}],
                          [],
                          [{'case',9,
                            {var,9,'Value1243560520536039'},
                            [{clause,9,
                              [{tuple,9,[{atom,9,atomic},{var,9,'_Value'}]}],
                              [],
                              [{atom,9,true}]},
                             {clause,9,
                              [{var,9,'_'}],
                              [],
                              [{atom,9,false}]}]}]}]}}},
                     {record_field,9,{atom,9,msgs},{nil,9}},
                     {record_field,9,{atom,9,prop},{atom,9,no}}]}]},
                 {cons,9,
                  {tuple,9,
                   [{integer,9,2},
                    {record,9,pattern_joins,
                     [{record_field,9,
                       {atom,9,test},
                       {'fun',9,
                        {clauses,
                         [{clause,9,
                           [{var,9,'Value1243560520536058'}],
                           [],
```

```
              [{'case',9,
                {var,9,'Value1243560520536058'},
                [{clause,9,
                  [{tuple,9,[{atom,9,test},{var,9,'_Value'}]}],
                  [],
                  [{atom,9,true}]},
                 {clause,9,
                  [{var,9,'_'}],
                  [],
                  [{atom,9,false}]}]}]}]}}},
       {record_field,9,{atom,9,msgs},{nil,9}},
       {record_field,9,{atom,9,prop},{atom,9,no}}]}]},
   {nil,9}}},
 {cons,8,
  {cons,11,
   {tuple,11,
    [{integer,11,3},
     {record,11,pattern_joins,
      [{record_field,11,
        {atom,11,test},
        {'fun',11,
         {clauses,
          [{clause,11,
            [{var,11,'Value1243560520536123'}],
            [],
            [{'case',11,
              {var,11,'Value1243560520536123'},
              [{clause,11,
                [{tuple,11,
                  [{atom,11,commit},{var,11,'_TransId'}]}],
                [],
                [{atom,11,true}]},
               {clause,11,
                [{var,11,'_'}],
                [],
                [{atom,11,false}]}]}]}]}}},
       {record_field,11,{atom,11,msgs},{nil,11}},
       {record_field,11,{atom,11,prop},{atom,11,no}}]}]},
   {nil,11}},
  {nil,8}}},
{cons,8,
 {'fun',8,
  {clauses,
   [{clause,8,
     [{var,8,'ResArgName1243560520536063'},
      {var,8,'CaseArg1243560520536066'}],
     [],
     [{match,8,
       {cons,9,
```

```
          {tuple,9,[{atom,9,atomic},{var,9,'Value'}]},
          {cons,9,
           {tuple,9,[{atom,9,test},{var,9,'Value'}]},
           {nil,9}}},
         {var,8,'ResArgName1243560520536063'}},
        {match,8,{var,25,'Guard1243560520536069'},{atom,8,true}},
        {'case',8,
         {var,8,'CaseArg1243560520536066'},
         [{clause,8,
           [{atom,8,test_entry}],
           [],
           [{var,8,'Guard1243560520536069'}]},
          {clause,8,
           [{atom,8,run_all}],
           [],
           [{tuple,10,[{atom,10,ok},{var,10,'Value'}]}]}]}]}]}},
 {cons,8,
  {'fun',8,
   {clauses,
    [{clause,8,
      [{var,8,'ResArgName1243560520536128'},
       {var,8,'CaseArg1243560520536130'}],
      [],
      [{match,8,
        {cons,11,
         {tuple,11,[{atom,11,commit},{var,11,'TransId'}]},
         {nil,11}},
        {var,8,'ResArgName1243560520536128'}},
       {match,8,
        {var,25,'Guard1243560520536132'},
        {op,11,'>',{var,11,'TransId'},{integer,11,0}}},
       {'case',8,
        {var,8,'CaseArg1243560520536130'},
        [{clause,8,
          [{atom,8,test_entry}],
          [],
          [{var,8,'Guard1243560520536132'}]},
         {clause,8,
          [{atom,8,run_all}],
          [],
          [{tuple,12,
            [{atom,12,ok},
             {tuple,12,
              [{atom,12,trans},{var,12,'TransId'}]}]}]}]}]}]}},
   {nil,8}}},
 {cons,13,
  {tuple,13,
   [{atom,13,timeout},
    {integer,13,10000},
```

```
          {'fun',13,
           {clauses,
            [{clause,13,[],[],
              [{tuple,14,[{atom,14,error},{atom,14,timeout}]}]}]}]},
       {nil,13}]}]}]},
{eof,17}]
```

Appendix C

gen_joins OTP's behaviour

C.1 Calculator

Listing C.1 presents an example of the usage of **gen_joins**. Module **jerlang_gen_joins_calculator** implements a simple Remote Procedure Call calculator. Clients are allowed to provide operands for the operations using the asynchronous operation defined in **number/2** function, as well as wait for the result of the operation using synchronous calls to **add/1**, **multiply/1** and **divide/1**. Calculator's internal state records the number of invalid operations that occured during its lifetime and hence behaves like a standard **gen_server** application.
Operations are done on the operands that agree on the identification number thanks to the possibility of non-linear patterns in *JErlang*. The order of operands is non-deterministic when it comes to pattern matching. The last **handle_join** will be successful if and only if all the combinations of the available messages fail for the previous join.
Synchronization on the messages using joins allows for more intuitive description of the problem. In a standard **gen_server**'s behaviour implementation we would have to explicitly store each of the messages in the state and perform necessary *Id* synchronisation ourselves.

```
1  -module(jerlang_gen_joins_calculator).
2  -behaviour(jerlang_gen_joins).
3
4  -compile({parse_transform, jerlang_gen_joins_parse}).
5
6  -export([init/1, handle_join/2, terminate/0]).
7  -export([start/0, add/1, multiply/1, divide/1, number/2]).
8
9  start() ->
10     jerlang_gen_joins:start({local, ?MODULE}, ?MODULE, [], []).
11
12 terminate() ->
13     jerlang_gen_joins:call(?MODULE, stop).
14
15 init(_Args) ->
16     {ok, 0}.
```

```
17
18 number(Number, Id) ->
19     jerlang_gen_joins:cast(?MODULE, {number, Id, Number}).
20
21 add(Id) ->
22     jerlang_gen_joins:call(?MODULE, {add, Id}).
23
24 multiply(Id) ->
25     jerlang_gen_joins:call(?MODULE, {multiply, Id}).
26
27 divide(Id) ->
28     jerlang_gen_joins:call(?MODULE, {divide, Id}).
29
30 %% ------------------------------------------------------------------
31 %% ---------------- CALLBACKS ---------------------------------------
32 %% ------------------------------------------------------------------
33 handle_join({number, Id, First} and {number, Id, Second}
34         and {number, Id, Third} and {add, Id}, Status) ->
35
36     io:format("[gen_joins]: Calculate 'add' operation~n", []),
37     {[noreply, noreply, {reply, First + Second + Third}],
38       Status};
39 handle_join({number, Id, First} and {number, Id, Second}
40         and {multiply, Id}, Status) ->
41
42     io:format("[gen_joins]: Calculate 'multiply' operation~n", []),
43     {[noreply, noreply, {reply, First * Second}], Status};
44 handle_join({number, Id, First} and {number, Id, Second}
45         and {divide, Id}, Status) when (Second =/= 0) ->
46
47     io:format("[gen_joins]: Calculate 'divide' operation~n", []),
48     {[noreply, noreply, {reply, First / Second}],
49       Status};
50 handle_join({number, Id, First} and {number, Id, Second}
51         and {divide, Id}, Status) when (Second =/= 0) ->
52
53     io:format("[gen_joins]: Calculate 'divide' operation~n", []),
54     {[noreply, noreply, {reply, First / Second}],
55       Status};
56
57 handle_join({number, Id, _} and {number, Id, Second}
58         and {divide, Id}, Status) when (Second =:= 0) ->
59     io:format("[gen_joins]: Invalid divide operation~n", []),
60     {[noreply, noreply, {reply, {error, division_by_zero}}],
61       Status + 1}.
```

Listing C.1: Calculator server implemented in *JErlang*using **gen_joins**'s behaviour

C.2 Chat system

Listing C.2 presents a fully featured chat system, where messages are normally broadcasted to all users. Additionally users can agree to go on a private channel. The latter happens when two of the

users agree on the creation of the room. We perform message synchronisation using joins in order to ensure this. Private session is enabled in the system using propagation of the room's session key. It is only removed once one of the two users decides to close the room.
Having a session message is much easier to understand since guards cannot actually perform more complicated executions like checking for the existence of elements in the list.

```
1  -module(jerlang_chat_system).
2  -compile({parse_transform, jerlang_gen_joins_parse}).
3
4  -behaviour(jerlang_gen_joins).
5
6  -export([init/1, handle_join/2, start/0, terminate/0]).
7  -export([login/1, logout/1, private_channel/2]).
8  -export([send/2, send/3, close_room/2]).
9
10 -record(user, {id, pid}).
11 -record(system, {users, rooms=[]}).
12
13 -define(SERVER, jerlang_gen_joins).
14
15 start() ->
16     ?SERVER:start({local, ?MODULE}, ?MODULE, [], []).
17
18 terminate() ->
19     ?SERVER:call(?MODULE, stop).
20
21 init(_) ->
22     {ok, #system{users=dict:new(), rooms=dict:new()}}.
23
24 login(User) ->
25     ?SERVER:call(?MODULE, {login, {User, self()}}).
26
27 logout(User) ->
28     ?SERVER:call(?MODULE, {logout, {User, self()}}).
29
30 private_channel(User1, User2) ->
31     ?SERVER:call(?MODULE, {private, {User1, self()}, User2}).
32
33 send(User, Msg) ->
34     ?SERVER:cast(?MODULE, {msg, {User, self()}, Msg}).
35
36 send(User, Room, Msg) ->
37     ?SERVER:cast(?MODULE, {msg, {User, self()}, Room, Msg}).
38
39 close_room(User, Room) ->
40     ?SERVER:call(?MODULE, {msg, close, {User, self()}, Room}).
41
42 %% ------------------------------------------------------
43
44 handle_join({login, User}, S) ->
45     {NS, Result} = add_login(
46                 valid_user(User, S),
47                 User, S),
```

```
48
49      {[{reply, Result}], NS};
50  handle_join({private, {User1, Pid1}, User2} and {private,{User2, Pid2}, User1},
51          #system{rooms=Rs}=S) ->
52      %% Open private channel
53      {Reply, NS} =
54          case valid_user({User1, Pid1}, S)
55                  and valid_user({User2, Pid2}, S) of
56              true ->
57                  %%  Assume unique enough
58                  Room = make_ref(),
59                  ?SERVER:cast(?MODULE, {room, Room}),
60                  {{room, Room}, S#system{rooms=dict:store(Room,
61                                  [User1, User2], Rs)}};
62              false ->
63                  {{error, invalid_authorization}, S}
64      end,
65      {[{reply, Reply}, {reply, Reply}], NS};
66  handle_join({room, Room} and {msg, close, User, Room}, S) ->
67      % Determine whether the user belongs to this room
68      Result =
69          case other_user(Room, User, S) of
70              {error, invalid} ->
71                  %% Invalid user tries to close the room
72                  ?SERVER:cast(?MODULE, {room, Room}),
73                  {error, invalid_user};
74              {ok, Pid2} ->
75                  Pid2 ! {chat, private, Room, User, close_room_request},
76                  {ok, closed}
77          end,
78      NS = remove_room(Room, S),
79      {[noreply, {reply, Result}], NS};
80  handle_join(prop({room, Room}) and {msg, User, Room, Msg}, S) ->
81      %% Find users of the room
82      case other_user(Room, User, S) of
83          {ok, Pid} ->
84              Pid ! {chat, private, Room, User, Msg};
85          _ ->
86              %% Ignore msg
87              ok
88          end,
89      {[noreply, noreply], S};
90  handle_join({msg, User, Msg}, S) ->
91      case valid_user(User, S) of
92          true ->
93              send_all(S, User, Msg);
94          _ ->
95              %% Ignore msg
96              ok
97          end,
98      {[noreply], S}.
99
100 %% ------------------------------------------------------
101 add_login(true, _, S) ->
```

```
102     {S, {error, invalid_user}};
103 add_login(_, {User, Pid}, #system{users=Users}=S) ->
104     {S#system{users=
105           dict:store(User, #user{id=User, pid=Pid}, Users)},
106      valid}.
107
108 valid_user({User, Pid}, #system{users=Users}) ->
109     try
110     #user{pid=Pid} = dict:fetch(User, Users),
111     true
112     catch
113     _:_ ->
114         false
115     end.
116
117 send_all(#system{users=Us}, User, Msg) ->
118     lists:map(
119       fun({_, #user{pid=Pid}}) ->
120           Pid ! {chat, public, User, Msg}
121       end, dict:to_list(Us)),
122     ok.
123
124 other_user(Room, {User, _}, #system{users=Us, rooms=Rs}) ->
125     case other_user0(dict:fetch(Room, Rs), User) of
126     {ok, Other} ->
127         #user{pid=Pid} = dict:fetch(Other, Us),
128         {ok, Pid};
129     Error ->
130         Error
131     end.
132
133 other_user0([User, Other], User) ->
134     {ok, Other};
135 other_user0([Other, User], User) ->
136     {ok, Other};
137 other_user0(_, _) ->
138     {error, invalid}.
139
140 remove_room(Room, S) ->
141     dict:erase(Room, S).
```

Listing C.2: Calculator server implemented in *JErlang* using **gen_joins**'s behaviour

Appendix D

Variations of the Santa Claus problem

D.1 Santa Claus in Erlang

```
1  -module(santa).
2  -author('ok@cs.otago.ac.nz'). % Richard A. O'Keefe
3  -export([start/0]).
4
5  %   This is an Erlang solution to "The Santa Claus problem",
6  %   as discussed by Simon Peyton Jones (with a Haskell solution using
7  %   Software Transactional Memory) in "Beautiful code".
8  %   He quotes J.A.Trono "A new exercise in concurrency", SIGCSE 26:8-10, 1994.
9  %
10 %   Santa repeatedly sleeps until wakened by either all of his
11 %   nine reindeer, back from their holidays, or by a group of three
12 %   of his ten elves.  If awakened by the reindeer, he harnesses
13 %   each of them to his sleight, delivers toys with them, and finally
14 %   unharnesses them (allowing them to go off on holiday).  If
15 %   awakened by a group of elves, he shows each of the group into
16 %   his study, consults with them on toy R&D, and finally shows them
17 %   each out (allowing them to go back to work).  Santa should give
18 %   priority to the reindeer in the case that there is both a group
19 %   of elves and a group of reindeer waiting.
20 %
21 %   Inspired by an old example of Dijkstra's, I solve this problem by
22 %   introducing two secretaries: Robin and Edna.  The reindeer ask Robin
23 %   for appointments.  As soon as she has nine waiting reindeer she sends
24 %   them as a group to Santa.  The elves as Edna for appointments.  As
25 %   soon as she has three waiting elves she sends them as a group to Santa.
26 %
27 %   The Haskell version is 77 SLOC of complex code.
28 %   The Erlang  version is 43 SLOC of straightforward code.
29
30 worker(Secretary, Message) ->
```

```
31      receive after random:uniform(1000) -> ok end,  % random delay
32      Secretary ! self(),           % send my PID to the secretary
33      Gate_Keeper = receive X -> X end,   % await permission to enter
34      io:put_chars(Message),        % do my action
35      Gate_Keeper ! {leave,self()},   % tell the gate-keeper I'm done
36      worker(Secretary, Message).     % do it all again
37
38  secretary(Santa, Species, Count) ->
39      secretary_loop(Count, [], {Santa,Species,Count}).
40
41  secretary_loop(0, Group, {Santa,Species,Count}) ->
42      Santa ! {Species,Group},
43      secretary(Santa, Species, Count);
44  secretary_loop(N, Group, State) ->
45      receive PID ->
46          secretary_loop(N-1, [PID|Group], State)
47      end.
48
49  santa() ->
50      {Species,Group} =
51      receive             % first pick up a reindeer group
52          {reindeer,G} -> {reindeer,G}% if there is one, otherwise
53      after 0 ->
54              receive         % wait for reindeer or elves,
55                  {reindeer,G} -> {reindeer,G}
56                ; {elves,G}    -> {elves,G}
57              end         % whichever turns up first.
58      end,
59      case Species
60        of reindeer -> io:put_chars("Ho, ho, ho!  Let's deliver toys!\n")
61         ; elves    -> io:put_chars("Ho, ho, ho!  Let's meet in the study!\n")
62      end,
63      [PID ! self() || PID <- Group], % tell them all to enter
64      [receive {leave,PID} -> ok end  % wait for each of them to leave
65      || PID <- Group],
66      santa().
67
68  spawn_worker(Secretary, Before, I, After) ->
69      Message = Before ++ integer_to_list(I) ++ After,
70      spawn(fun () -> worker(Secretary, Message) end).
71
72  start() ->
73      Santa = spawn(fun () -> santa() end),
74      Robin = spawn(fun () -> secretary(Santa, reindeer, 9) end),
75      Edna  = spawn(fun () -> secretary(Santa, elves,    3) end),
76      [spawn_worker(Robin, "Reindeer ", I, " delivering toys.\n")
77       || I <- lists:seq(1, 9)],
78      [spawn_worker(Edna,  "Elf ",      I, " meeting in the study.\n")
79       || I <- lists:seq(1, 10)].
```

Listing D.1: Santa claus problem solution as defined in *http://www.cs.otago.ac.nz/staffpriv/ok/santa/santa.erl*

D.2 JErlang solution to Santa Claus using popular style

Listing D.2 presents a solution that is inspired by typically cited work when using *Join-calculus* for solving the *Santa Claus* problem. Clearly it is a minimal solution, where the synchronisation joins itself takes 14 lines of plain old patterns and actions on successful matches. The positive approach is that due to the properties of *JErlang*'s joins we do not have to perform any management for prioritising *reindeers* as it is done for example in [7]. We believe however that the solution discussed in 6.2.1 is still much more intuitive than this one. The advantage of the latter is that it can be easily adapted to the situation where we need more *reindeers* or *elves* to synchronise. The disadvantage is that in a hypothetical, yet very possible situation when each *reindeer* or *elf* is distinct and requires different action, the codebase and complexity increases dramatically in example D.2. But we believe that all the other existing *Join-calculus* implementations would have problems with that as well due to the language limitations.

The solution uses a secretary for finding necessary synchronisation. Initially the **secretary** sends two messages that describe the initial number of required *reindeers* and *elves* (lines 13 and 14). There are decremented each time

```
1  -module(jerlang_santa_claus_common).
2  -export([santa/0, worker/4, secretary0/1]).
3  -export([start/0]).
4  -ifdef(use_joins_vm).
5  -compile({parse_transform, jerlang_vm_parse}).
6  -else.
7  -compile({parse_transform, jerlang_parse}).
8  -endif.
9
10 secretary0(SantaPid) ->
11     self() ! {reindeers, 9},
12     self() ! {elves, 3},
13     secretary({[], []}, SantaPid).
14
15 secretary({Elves, Reindeers}, SantaPid) ->
16     {Res, Wait} =
17         receive
18             {reindeer, Pid} and {reindeers, 1} ->
19                 self() ! {reindeers, 9},
20                 notify_santa(reindeers, SantaPid, [Pid | Reindeers]),
21                 {{Elves, []}, true};
22             {reindeer, Pid} and {reindeers, N} ->
23                 self() ! {reindeers, N - 1},
24                 {{Elves, [Pid | Reindeers]}, false};
25             {elf, Pid} and {elves, 1} ->
26                 self() ! {elves, 3},
27                 notify_santa(elves, SantaPid, [Pid | Elves]),
28                 {{[], Reindeers}, true};
29             {elf, Pid} and {elves, N} ->
30                 self() ! {elves, N - 1},
31                 {{[Pid | Elves], Reindeers}, false}
32         end,
33     ok = wait_for_santa(Wait),
34     secretary(Res, SantaPid).
35
```

```
36 wait_for_santa(true) ->
37     receive {santa, back} -> ok end;
38 wait_for_santa(_) ->
39     ok.
40
41 notify_santa(Animal, Santa, Pids) ->
42     Santa ! {wakeup, self(), Animal, Pids}.
43
44 notify(Pids) ->
45     lists:foreach( fun(Pid) -> Pid ! continue end, Pids).
46
47 santa() ->
48     Pid =
49     receive
50         {wakeup, Secretary, Animal, Pids} ->
51             santa_says(Animal),
52             notify(Pids),
53             Secretary
54     end,
55     timer:sleep(random:uniform(1000)),
56     io:format("Santa is back. Going to sleep~n", []),
57     Pid ! {santa, back},
58     santa().
59
60 worker(Type, Secretary, Id, String) ->
61     timer:sleep(random:uniform(1000)),
62     Secretary ! {Type, self()},
63     receive continue -> ok end,
64     worker(Type, Secretary, Id, String).
65
66 santa_says(reindeers) ->
67     io:format("Ho, ho, ho! Let's deliver presents!~n", []);
68 santa_says(elves) ->
69     io:format("Ho, ho, ho! Let's discuss R&D possibilities!~n", []).
70
71 start() ->
72     Santa = spawn(?MODULE, santa, []),
73     Secretary = spawn(?MODULE, secretary0, [Santa]),
74     [ spawn(?MODULE, worker,
75         [elf, Secretary, X, "Elf "]) || X <- [1,3] ],
76     [ spawn(?MODULE, worker,
77         [reindeer, Secretary, X, "Reindeer "]) || X <- [1,3,4,7,8,9] ],
78     [ spawn(?MODULE, worker,
79         [elf, Secretary, X, "Elf "]) || X <- [2,4,5] ],
80     [ spawn(?MODULE, worker,
81         [reindeer, Secretary, X, "Reindeer "]) || X <- [2,5,6] ].
```

Listing D.2: Santa claus problem solved in a similar style as in [7] but with a more intuitive design

D.3 Sad Santa Claus problem

In order to complicate things even more we introduce a variation on the *Santa Claus* problem that involves the participation of *orks*. These creatures obviously do not like Christmas, so from time to time, when they manage to get to Santa's factory they destroy presents. But *orks* are usually primitive so in order to attack the factory, at the same time there have to be two "sergeant" *orks* and one "captain" *ork*, who gives orders. Additionally *orks* are much faster than *elves* and slower than *reindeers* which determines the order of the execution of actions.

The indentation of the *Sad Santa Claus* is intended to provoke thinking about how the synchronisation patterns get complicated with more diversity. We have different types of *orks*, different priorities. Expressing these in any known *Join-calculus* implementation certainly involves additional management of messages etc. The solution for example in standard *Erlang* will become even more cumbersome: a new secretary, "hacks" that ensure correct priorities, search for a "captain" in the team of *orks* etc.

Listing D.3 presents a solution to the *Sad Santa Claus* problem in *JErlang*. In comparison to the original problem we added 4 lines (20 -24) of which only a single describes the synchronisation. Additionally we added code that properly manages the new *ork* processes. *ork* processes typically take longer to send messages, since we increased the random distribution of wait from 4000 to 8000 (line 36), but that does not correspond directly to the nature of the problem - we just wanted to have a bigger chance of getting presents.

```
1  -module(jerlang_santa_claus_sad).
2  -export([start/0]).
3
4  -ifdef(use_joins_vm).
5  -compile({parse_transform, jerlang_vm_parse}).
6  -else.
7  -compile({parse_transform, jerlang_parse}).
8  -endif.
9
10 santa() ->
11     io:format("It was a long night. Time to bed~n"),
12     Group =
13     receive
14         {reindeer, Pid1} and {reindeer, Pid2} and {reindeer, Pid3}
15           and {reindeer, Pid4} and {reindeer, Pid5} and {reindeer, Pid6}
16           and {reindeer, Pid7} and {reindeer, Pid8} and {reindeer, Pid9} ->
17             io:format("Ho, ho, ho! Let's deliver presents!~n"),
18             [Pid1, Pid2, Pid3, Pid4,
19              Pid5, Pid6, Pid7, Pid8, Pid9];
20         {ork, sergeant,Pid1} and {ork, captain,Pid2} and {ork, sergeant,Pid3} ->
21             io:format("Ho, ho, ho? No presents this year.
22                        Orks destroyed the factory!~n"),
23             [Pid1, Pid2, Pid3];
24         {elf, Pid1} and {elf, Pid2} and {elf, Pid3} ->
25             io:format("Ho, ho, ho! Let's discuss R&D possibilities!~n"),
26             [Pid1, Pid2, Pid3]
27     end,
28     [ Pid ! ok || Pid <- Group],
29     santa().
30
31 worker(Santa, Type, Id, Action) ->
```

```
32      generate_seed(Id),
33      worker1(Santa, Type, Id, Action).
34
35  worker1(Santa, {ork, Type}, Id, Action) ->
36      receive after random:uniform(8000) -> ok end,
37      Santa ! {ork, Type, self()},
38      io:format("~p '~p' ~p: Waiting at the gate~n", [ork, Type, Id]),
39      receive ok -> ok end,
40      io:format("~p '~p' ~p: ~p~n", [ork, Type, Id, Action]),
41      worker1(Santa, {ork, Type}, Id, Action);
42  worker1(Santa, Type, Id, Action) ->
43      receive after random:uniform(4000) -> ok end,
44      Santa ! {Type, self()},
45      io:format("~p ~p: Waiting at the gate~n", [Type, Id]),
46      receive ok -> ok end,
47      io:format("~p ~p: ~p~n", [Type, Id, Action]),
48      worker1(Santa, Type, Id, Action).
49
50  generate_seed(Seed) ->
51      {A1, A2, A3} = now(),
52      random:seed(A1+Seed, A2*Seed, A3).
53
54  start() ->
55      Santa = spawn(fun() -> santa() end),
56      [spawn(fun() -> worker(Santa, reindeer, I, " delivering toys.\n") end)
57       || I <- lists:seq(1, 9)],
58      [spawn(fun() -> worker(Santa, elf, I, " meeting in the study.\n") end)
59       || I <- lists:seq(1, 10)],
60      [spawn(fun() -> worker(Santa, {ork, sergeant}, I, " destroying presents.\n")
61          end) || I <- lists:seq(1,4)],
62      spawn(fun() -> worker(Santa, {ork, captain}, 5, " making orders.\n") end).
```

Listing D.3: Solution to the *Sad Santa Claus* problem in *JErlang*

D.4 Santa Claus in gen_joins

The **gen_joins** implementation given in listing D.4 contains all the functionality presented in the original *JErlang* solution of the problem. Additionally we introduce synchronisation on the synchronous call **status** that allows for the design of a test-suite that counts the time necessary to perform some given number of *reindeers* and *elves* actions. In other words the synchronous **status** call stalls until the correct internal state of the **gen_joins** process is reached.

```
1
2  -export([start/0, stop/0, elf/0, reindeer/0]).
3  -export([elf_done/0, reindeer_done/0, status/1]).
4
5  start() ->
6      jerlang_gen_joins:start({local, ?MODULE}, ?MODULE, [], []).
7
8  stop() ->
9      jerlang_gen_joins:call(?MODULE, stop).
10
11 terminate() ->
12     ok.
13
14 init(_) ->
15     {ok, {{0,0}, sleeping}}.
16
17 elf() ->
18     jerlang_gen_joins:call(?MODULE, elf, infinity),
19     ok.
20
21 elf_done() ->
22     jerlang_gen_joins:cast(?MODULE, {done, elf}).
23
24 reindeer() ->
25     jerlang_gen_joins:call(?MODULE, reindeer, infinity).
26
27 reindeer_done() ->
28     jerlang_gen_joins:cast(?MODULE, {done, reindeer}).
29
30 status(Status) ->
31     jerlang_gen_joins:call(?MODULE, {status, Status}, infinity).
32
33 handle_join({status, {A, B}}, {{C, D}, _}=S) when ((A =< C) and (B =< D)) ->
34     io:format("I am done ~p~n", [{C, D}]),
35     {[{reply, ok}], S};
36 handle_join(stop, _) ->
37     io:format("Stopping the server~n", []),
38     {stop, normal};
39 handle_join({done, reindeer} and {done, reindeer} and {done, reindeer} and
40         {done, reindeer} and {done, reindeer} and {done, reindeer} and
41         {done, reindeer} and {done, reindeer} and {done, reindeer},
42         {Counter, awake_reindeer}) ->
43     io:format("All reindeers returned~n",[]),
44     {[ noreply || _ <- lists:seq(1,9)], {Counter, sleeping}};
45
```

```
46 handle_join({done, elf} and {done, elf} and {done, elf},
47         {Counter, awake_elf}) ->
48     io:format("All elves returned~n",[]),
49     {[ noreply || _ <- lists:seq(1,3)], {Counter, sleeping}};
50
51 handle_join(reindeer and reindeer and reindeer and
52         reindeer and reindeer and reindeer and
53         reindeer and reindeer and reindeer,
54         {{Reindeers, Elves}, sleeping}) ->
55     io:format("Ho, ho, ho! Let's deliver presents~n",[]),
56     {[ {reply, ok} || _ <- lists:seq(1,9)], {{Reindeers+1, Elves}, awake_reindeer}};
57
58 handle_join(elf and elf and elf,
59         {{Reindeers, Elves}, sleeping}) ->
60     io:format("Ho, ho, ho! Let's discuss R&D possibilities~n",[]),
61     {[ {reply, ok} || _ <- lists:seq(1,3)], {{Reindeers, Elves+1}, awake_elf}}.
```

Listing D.4: **gen_joins** solution to the *Santa Claus* problem in *JErlang*

Appendix E

Benchmarks

E.1 Multiple producers test-suite

Example E.1 presents an artificial situation where the programmer wants to achieve synchronisation on multiple calls. The run of the test (not included here, but available in module **multiple_producers_test.erl** runs multiple "producer" processes that create large amounts of messages by calling **notify**, **sell**, **buy**, **credentials** and **deposit**. All of those functions (apart from **notify**) create asynchronous messages as fast they can. For a realistic situation we throttle them using randomly distributed **sleep** function (from 1 to 1000 miliseconds). Any of the first two joins requires synchronous call to **notify** and to create a large backlog of messages and push our joins solver to the limits, there are only a few (in comparison to functions) processes that try to send synchronous **notify** message.

We measure the effectiveness of our rather complex joins solver by the number of joins it is capable of firing in the given amount of time.

```
1  -module(multiple_producers).
2  -compile({parse_transform, jerlang_gen_joins_parse}).
3
4  -behaviour(jerlang_gen_joins).
5
6  -export([init/1, handle_join/2, terminate/0]).
7  -export([start/0, stop/0]).
8  -export([notify/0, notify/1, credentials/2,
9      deposit/1, sell/2, buy/3]).
10
11 -record(packet,{id, value}).
12
13 -define(LIMIT, 250).
14 -define(GEN_JOINS, jerlang_gen_joins).
15
16 start() ->
17     ?GEN_JOINS:start({local, ?MODULE}, ?MODULE, [], []).
18
19 stop() ->
```

```
20      ?GEN_JOINS:call(?MODULE, stop).
21
22  terminate() ->
23      ok.
24
25  init(_) ->
26      {ok, 0}.
27
28  notify() ->
29      ?GEN_JOINS:call(?MODULE, notify, infinity).
30
31  notify(Timeout) ->
32      ?GEN_JOINS:call(?MODULE, notify, Timeout).
33
34  sell(Id, Value) ->
35      ?GEN_JOINS:cast(?MODULE, #packet{value=Value, id=Id}).
36
37  buy(Id, Value, Previous) ->
38      ?GEN_JOINS:cast(?MODULE, {buy, Id, Value, Previous}).
39
40  credentials(Id, Password) ->
41      ?GEN_JOINS:cast(?MODULE, {secure, Id, Password}).
42
43  deposit(V) ->
44      ?GEN_JOINS:cast(?MODULE, {deposit, V}).
45
46  %% JOINS
47
48  handle_join(notify and #packet{value=V1, id=Id} and {buy, Id, _, _Previous},
49                  State) ->
50              {[{reply, {ok, buy, Id}}, noreply, noreply], State - V1};
51  handle_join(notify and {deposit, V1} and #packet{value=_, id=Id}
52                  and {secure, Id, _}, State) ->
53              {[{reply, {ok, sell, Id}}, noreply, noreply, noreply],
54                  State + (V1)};
55  handle_join(stop, _) ->
56              {stop, normal}.
```

Listing E.1: Application with numerous asynchronous calls in complex joins in *JErlang*

Appendix F

NataMQ

For the detailed description of *NataMQ* application please refer to section 6.5.1.

F.1 Nata Exchange

Nata Exchange is central to the existence of the *NataMQ* system. It is the only entity capable of creating queues to which clients can subscribe. It also stores routing declarations that determine the action on incoming messages. Two join constructs are used in order to allow for the creation of synchronisation patterns on the messages' keys (lines 60-61 and 64-65). The propagation feature is used in order allow for checking for any key's rounting information, since such information cannot be performed inside the guards. A lack of propagation would result in performance drop, since we want keep the routing information to match as early as possible, especially in a high-load environment for which *NataMQ* was designed.

```
1  -module(nata_exchange).
2
3  -compile({parse_transform, jerlang_gen_joins_parse}).
4
5  -include("nata.hrl").
6
7  -behaviour(jerlang_gen_joins).
8
9  -export([init/1, handle_join/2, start/0, terminate/0]).
10 -export([create_route/3, delete_route/1]).
11 -export([publish/1]).
12
13 -define(CHMODULE, nata_channel).
14 -define(SERVER, jerlang_gen_joins).
15 -define(MAX, 4).
16
17 init(_) ->
18     {ok, {dict:new(), dict:new()}}.
19
```

```
20 start() ->
21     ?SERVER:start({local, ?MODULE}, ?MODULE, [], []).
22
23 terminate() ->
24     ?SERVER:call(?MODULE, stop).
25
26 create_route(Key, Synchr, Channels) ->
27     ?SERVER:call(?MODULE, {route, Key, Synchr, Channels}).
28
29 delete_route(Key) ->
30     ?SERVER:call(?MODULE, {remove_route, Key}).
31
32 publish(Msg) ->
33     ?SERVER:cast(?MODULE, Msg).
34
35 %% ------------------------- CALLBACKS -------------------------
36
37 handle_join({route, Key, Synchr, Channels}, Status)
38   when ((Synchr > 0) and (Synchr < ?MAX)) ->
39     %% Whenever rout already exists we only update
40     %% the number of channels
41
42     io:format("[Ex] Create route ~p~n", [Key]),
43     {Reply, NS} =
44     try
45         {Pids, {NewRoute, UpdatedS}} =
46             get_route(Key, Synchr, Channels, Status),
47         ok = set_route(NewRoute, Key, Synchr),
48         {{ok, Pids}, UpdatedS}
49     catch
50         throw:_ ->
51         {{error, {invalid_synchr, Synchr}}, Status}
52     end,
53     {[{reply, Reply}], NS};
54 handle_join({remove_route, Key} and {route, Key, _}, Status) ->
55     NS = remove_route(Key, Status),
56     {[{reply, ok}, noreply], NS};
57 handle_join(#msg{key=none}, Status) ->
58     io:format("[Ex] Invalid message. Key required~n", []),
59     {[noreply], Status};
60 handle_join(#msg{key=Key, value=V1} and #msg{key=Key, value=V2}
61         and prop({route, Key, 2}), Status) ->
62     route_message(Key, [V1, V2], Status),
63     {[noreply || _ <- lists:seq(1, 3)], Status};
64 handle_join(#msg{key=Key, value=V1} and #msg{key=Key, value=V2} and
65         #msg{key=Key, value=V3} and prop({route, Key, 3}), Status) ->
66
67     route_message(Key, [V1, V2, V3], Status),
68     {[noreply || _ <- lists:seq(1,4)], Status}.
69
70 %% --------------------------------------------------------
71 %% ---- INTERNAL FUNCTIONS ----------------------------------
72 %% --------------------------------------------------------
73
```

```
74  get_route(Key, Synchr, ChNames, {S1, S2}) ->
75      {Update, NS1} =
76      case dict:find(Key, S1) of
77          {ok, {Synchr, Values}} ->
78          NChNames = (Values ++ (ChNames -- Values)),
79          {no, dict:store(Key, {Synchr, NChNames}, S1)};
80          {ok, _} ->
81          throw(invalid_synchr);
82          error ->
83          {yes, dict:store(Key, {Synchr, ChNames}, S1)}
84      end,
85      {ChPids, NS2} =
86      start_channels(ChNames, S2),
87      {ChPids, {Update, {NS1, NS2}}}.
88
89
90  set_route(yes, Key, Synchr) ->
91      ?SERVER:cast(?MODULE, {route, Key, Synchr}),
92      ok;
93  set_route(_, _, _) ->
94      ok.
95
96  route_message(Key, Value, {S1, S2}) ->
97      {_, ChNames} = dict:fetch(Key, S1),
98      ChPids = lists:map(
99            fun(Name) ->
100               dict:fetch(Name, S2)
101           end, ChNames),
102     lists:map(
103       fun(Pid) ->
104           ?CHMODULE:route(Pid, Key, Value)
105       end, ChPids).
106
107 start_channels(Channels, Store) ->
108     start_channels(Channels, [], Store).
109
110 start_channels([], Result, Store) ->
111     {lists:reverse(Result), Store};
112 start_channels([Ch | Rest], Result, Store) ->
113     {Pid, NS} =
114     case dict:find(Ch, Store) of
115         {ok, ChPid} ->
116         {ChPid, Store};
117         error ->
118         {ok, ChPid} = ?CHMODULE:start_link(Ch),
119         NStore = dict:store(Ch, ChPid, Store),
120         {ChPid, NStore}
121     end,
122     start_channels(Rest, [Pid | Result], NS).
123
124 remove_route(Key, Status) ->
125     dict:erase(Key, Status).
```

Listing F.1: Nata Exchange written in *JErlang*

```
-record(msg, {key=none,
              id=none,
              value}).
-record(channel_msg, {msg}).
```

Listing F.2: *NataMQ*'s header file

F.2 Nata Channel

Channel is used for storing the messages that were passed by the *Nata Exchange*. The entity currently does not perform persistent storage of the messages i.e. whenever there are subscribers to the queue, then all the messages are immediately passed to them, otherwise we stall. The application could be designed for more sophisticated purposes like no automatic removal of the message, authentication requirements etc., but for clarity we omit the details.

Each *Nata Channel* runs as a separate process in a typical client-server model which increase the possibility of extending the behaviour to new functionality.

```
-module(nata_channel).

-behaviour(gen_server).

%% API
-export([start_link/1]).

-export([init/1, handle_call/3, handle_cast/2, handle_info/2,
      terminate/2, code_change/3]).

-export([subscribe/1, unsubscribe/1, route/3]).

-record(state, {name, msgs, subs=[]}).
-define(SERVER, gen_server).


start_link(Name) ->
    ?SERVER:start_link(?MODULE, [Name], []).

init([Name]) ->
    {ok, #state{name=Name, msgs=queue:new()}}.

subscribe(ChPid) ->
    ?SERVER:call(ChPid, consume).

unsubscribe(ChPid) ->
    ?SERVER:call(ChPid, not_consume).

%% --------------------------------------------------------------------
route(ChPid, Key, Msg) ->
    ?SERVER:cast(ChPid, {msg, Key, Msg}).

%% ---------------------- CALLBACKS ----------------------------------

```

```
35 handle_cast({msg, Key, Msg}, #state{msgs=Q, subs=[]}=S)  ->
36     {noreply, S#state{msgs=queue:in(Q, {Key, Msg})}};
37 handle_cast({msg, Key, Msg}, #state{name=N, subs=Subs}=S) ->
38     io:format("[Channel ~p] Notify of ~p~n", [N, Msg]),
39     notify_subscribers(Msg, Key, Subs),
40     {noreply, S};
41 handle_cast(_, State) ->
42     {noreply, State}.
43
44 handle_call(consume, {From, _}, #state{name=N, msgs=Q, subs=[]}=S) ->
45     io:format("[Channel ~p] First consumer ~p~n", [N, From]),
46     notify_with_old_messages(Q, From),
47     {reply, ok, S#state{msgs=queue:new(), subs=[From]}};
48 handle_call(consume, {From, _}, #state{name=N, subs=Subs} = S) ->
49     io:format("[Channel ~p] Consumer ~p~n", [N, From]),
50     {reply, ok, S#state{subs=[From | Subs]}};
51 handle_call(_, _, State) ->
52     {noreply, State}.
53
54 handle_info(_Info, State) ->
55     {noreply, State}.
56
57 terminate(_Reason, _State) ->
58     ok.
59
60 code_change(_OldVsn, State, _Extra) ->
61     {ok, State}.
62
63 %%--------------------------------------------------------------------
64 %%% Internal functions
65 %%--------------------------------------------------------------------
66
67 notify_subscribers(_, _, []) ->
68     ok;
69 notify_subscribers(Msg, Key, [S | Rest]) ->
70     S ! {channel, Key, Msg},
71     notify_subscribers(Msg, Key, Rest).
72
73 notify_with_old_messages(Q, Subs) ->
74     L = queue:to_list(Q),
75     notify(L, Subs).
76
77 notify([], _) ->
78     ok;
79 notify([{Key, M} | Rest], Subs) ->
80     Subs ! {channel, Key, M},
81     notify(Rest, Subs).
```

Listing F.3: Nata Channel written in *JErlang*

F.3 Nata Publish and Subscribe

Nata Publish and *Nata Subscribe* are basic processes provided for completeness. They use the API provided by the *Nata Exchange*. The initial configuration parameters are self-explanatory and for more examples of usage see the module **nata_test** in the source code (not included in the Appendix).

```
1  -module(nata_publish).
2
3  -include("nata.hrl").
4
5  -define(DELAY, 5000).
6
7  -export([start/1]).
8
9  start(State) ->
10     init(State),
11     publish(State).
12
13 init({_, Seed, _Key, _Type}) ->
14     {A1, A2, A3} = now(),
15     random:seed(A1+Seed, A2*Seed, A3*Seed),
16     io:format("[Producer ~p]: Start ~n", [Seed]),
17     ok.
18
19 publish({Exchange, Seed, Key, normal}=State) ->
20     timer:sleep(random:uniform(?DELAY)),
21     io:format("[Producer ~p]: publish message [~p]~n", [Seed, Key]),
22     Exchange:publish(#msg{key=Key,
23                 id=none,
24                 value={test, Seed}}),
25     check_proceed(State).
26
27 check_proceed({_, Seed, _, _} = State) ->
28     receive
29     stop ->
30         io:format("[Producer ~p]: stop~n", [Seed])
31     after 0 ->
32         publish(State)
33     end.
```

Listing F.4: Nata Publish written in *JErlang*

```
1  -module(nata_subscribe).
2
3  -include("nata.hrl").
4
5  -export([start/1]).
6
7  -define(CHANNLE, a).
8  -define(EXCHANGE, nata_exchange).
9  -define(CHMODULE, nata_channel).
10
11 start(Conf) ->
```

```
12      NConf = init(Conf),
13      consume(NConf).
14
15  consume({_, Id, _, _, _}=Conf) ->
16      receive
17      {channel, Name, Msg} ->
18          io:format("[Subscriber ~p]: Received ~p on [~p]~n",
19                  [Id, Msg, Name]),
20          ok;
21      stop ->
22          exit(normal)
23      end,
24      consume(Conf).
25
26  init({Exchange, Id, Key, Synchr, Channels}) ->
27      {ok, CHPids} = Exchange:create_route(Key, Synchr, Channels),
28      lists:foreach( fun(Ch) -> ?CHMODULE:subscribe(Ch) end, CHPids),
29      {Exchange, Id, Key, Synchr, CHPids}.
```

Listing F.5: Nata Subscribe written in *JErlang*