JErlang: Erlang with Joins

Hubert Plociniczak[1,⋆] and Susan Eisenbach[2]

[1] École Polytechnique Fédérale de Lausanne
hubert.plociniczak@epfl.ch
[2] Imperial College London
s.eisenbach@imperial.ac.uk
http://www.doc.ic.ac.uk/~susan/jerlang/

Abstract. ERLANG is an industrially successful functional language that uses the Actor model for concurrency. It supports the message-passing paradigm by providing pattern-matching over received messages. Unfortunately coding synchronisation between multiple processes is not straightforward. To overcome this limitation we designed and implemented JERLANG, a JOIN-CALCULUS inspired extension to ERLANG. We provide a rich set of language features with our *joins*. We present implementation details of our two alternative solutions, a library and an altered VM. Our optimisations provide JERLANG with good performance.

Keywords: Concurrency, Join-Calculus, Erlang, Static Analysis.

1 Introduction

Writing concurrent and distributed applications is at the heart of current software development. Even though these problems are not new, language development has yet to catch up with the current reality and programmers still have to use error prone techniques such as primitive locking mechanisms. An effort that improves on this situation is the single assignment functional language, ERLANG [1], for the development of concurrent, distributed, fault-tolerant and now multi-core systems. ERLANG, designed by Joe Armstrong from Ericsson, was aimed at tackling telecom problems, such as building zero-downtime systems, which manage millions of concurrent processes.

For concurrency and distribution ERLANG relies on the message passing, Actor paradigm, which defines the communication between processes. However synchronisation is required in many concurrent problems and Actors are not the most natural paradigm for providing it. Our aim is to extend the choice of constructs provided to the programmer, without threatening the safety that a non-shared memory language provides. For this we chose the *join*, a synchronisation construct from the JOIN-CALCULUS [2], a calculus designed with implementation in mind, which with its firm formal foundation fits well with ERLANG, while providing elegant, powerful and expressive constructs. In

⋆ The author worked on this research during his studies at Imperial College London.

this paper we introduce our Erlang extension, JErlang, which is available with the companion technical report, source code and many examples from `http://www.doc.ic.ac.uk/~susan/jerlang/`.

In Section 2 we show the need for adding joins with an Erlang example that is problematical. Section 3 is devoted to the definition and design decisions for JErlang. We wanted it to remain backward-compatible with the original version and introduce features which current Erlang programmers would want to use. Hence we couldn't just copy the ideas from existing implementations of JoCaml [3], Cϖ [4], or Scala [5], but adapted *joins* to Erlang's powerful message receive and pattern matching. Erlang wouldn't have become so popular without the existence of its Open Telecom Platform (OTP) design patterns that provide customisable solutions for client-server, fault-tolerant applications development. Therefore in order to attract the Erlang audience we provide *join* inspired client-server `behaviour` that we call **`gen_joins`**, for building synchronisation patterns within server applications.

We provide our *joins* implementation in the form of a stand-alone library, which contains transformation functions, available for the compiler, that convert JErlang into Erlang source code before it is actually checked. We have also developed a second version of JErlang which uses Erlang's Virtual Machine in modified form for performance reasons. In Section 4 we describe both implementations. We also present the algorithms used in the non-trivial implementation of our join-solver. Section 5 explains novel optimisations in the implementation done in order to boost the performance.

Our analysis in Section 6 has shown acceptable performance in most of the shown situations and more importantly an improved expressiveness and clarity in comparison with the original language. Finally we present related work in Section 7 and conclude in Section 8. A formal definition of our extension, included in the companion technical report, gave us a better understanding of how joins should fit into the Erlang language and what semantics we should chose for it.

2 From Erlang to JErlang

Central to Erlang is the notion of a *process*, created with a **`spawn`** statement which upon successful execution returns a process id (*PID*). To enable interprocess communication, each Erlang process has a single mailbox containing a queue for incoming messages. Sending asynchronous messages to other processes is done through a **!** (send) operator: `Pid ! Value`.

```
receive
  {msg, Val1, Res} when (Val1 == test) -> Res; %% pattern with guard
  {error, Error} -> none
after DefaultTimeout -> timeout end
```

Listing 1. Typical actor programming in Erlang

A *process* analyses the contents of the mailbox using the *Selective Receive* construct, as shown in listing 1. The incoming messages are tested against all patterns until a match is found (with satisfied guards) or the timeout limit is reached. Only the idle time of waiting for new messages contributes towards the latter.

The **`receive`** construct has its limitations. Consider synchronising on two messages that match the patterns {**`get, A`**} and {**`set, B`**}, where A and B are equal, given the following mailbox (the oldest message is on the left): ({**`get, 1`**} · {**`set, 4`**} · {**`set, 2`**} · {**`get, 2`**}). A typical implementation by a beginner Erlang programmer is presented below:

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

When retrieving messages from a mailbox, the operation will match the first {**`get, X`**} pattern and get stuck, since there is no message {**`set, 1`**} in the mailbox, as X is now bounded. The programmers must consider the possible layouts of the mailbox and how the processes would interact.

Listing 2 presents a refined program where we continuously fetch messages from the queue and at the stage when no synchronisation can be fulfilled with the first message (**`after 0`**), we resend it (**`self() !`**) and call the function again until successful (**`Func(Func)`**). The order of the messages in the queue is not preserved, the solution is error prone and inefficient and wastes computational power, whenever the second pattern cannot be satisfied. Erlang programmers also lack the language support they need in several other areas, such as matching on multiple messages that convey priority values. This limitation of the language leads to complicated code, which in turn leads to the re-invention of the mailbox mechanism at the application level.

```
A = fun(Func) ->
  receive
    {get, X} -> receive {set, Y} when (X == Y) -> {found, X}
                after 0 -> self() ! {get, X}, Func(Func) end
    {set, X} -> receive {get, Y} when (X == Y) -> {found, X}
                after 0 -> self() ! {set, X}, Func(Func) end
  end,
A(A)
```

Listing 2. Synchronisation on two messages without order preservation

The Join-Calculus [2] is a process calculus, computationally equivalent to the π-Calculus [2], but designed to be a basis for a concurrent programming language. The Join-Calculus introduces *multi-way join patterns* that enable synchronisation of multiple message patterns on different communication channels. It is this construct that made it our choice for adding to Erlang. As a

Join-Calculus example program consider a single-cell stack. In this buffer you can only **push** an item if the buffer is empty. If the buffer is full and a **pop** occurs, the **pop** and **push** *join* is reduced, and the item is retrieved, making the buffer empty.

$$def\ pop\langle\mu\rangle \mid s\langle\nu\rangle \triangleright empty\langle\rangle \mid \mu\langle\nu\rangle \wedge push\langle t\rangle \mid empty\langle\rangle \triangleright s\langle t\rangle$$

Symbol $|$ defines the *and* semantics in the join operation, symbol $\wedge$ combines the related join definitions together and $\triangleright$ precedes the join's action definition.

3 JErlang Language Features

JErlang provides synchronisation semantics with a **receive**-like join construct. Using the guards in listing 3 we end up with the intuitive (and correct) solution to the motivating problem from listing 2 which reduces the eight lines to one.

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

Listing 3. Synchronisation on two messages with guards in JErlang

In the Join-Calculus implementations of JoCaml [3] and Cϖ [4] whenever there is more than one satisfiable join then the execution is non-deterministic. In the implementation of **receive** in JErlang we assume that the semantics of *First-Match* [6] provides more predictable behaviour. With *First-Match* strategy we gradually increase the "window" of the mailbox which is used to analyse deterministically if any join can be satisfied with it. The strategy ensures that whenever a join is satisfied by a strict prefix of the original mailbox then it is the smallest prefix that can satisfy any join. Whenever the current mailbox prefix is able to satisfy more than a single join, then the order of declaration of the joins determines the successful one. Obviously, the order of messages for such prefix is significant for matching, but here it is enough to assume that the matching of the messages follows the total ordering scheme.

```
self() ! {foo, one}, self() ! {error, 404}, self() ! {bar, two},
receive
    {foo, A} and {bar, B} -> {error, {A, B}};
    {error, 404}          -> {ok, error_expected};
    {error, Reason}       -> {error, Reason}       %% general error
end
```

Listing 4. Impact of First-match semantics on joins

Example 4 underlines the consequence of our matching strategy 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. By looking only at the first two messages of the mailbox in the example (starting with an empty mailbox), the second join is satisfied after the analysis of the first two messages, whereas the first join at this stage still misses one more successful pattern. By ensuring this deterministic behaviour in our implementation we believe that developers gain more control. *First-Match* also allows for having the typical design schema where patterns are described from the least to the most general.

```
receive
  {amount, Transaction, Money} and {limit, LowerLimit, UpperLimit}
      when (Money < UpperLimit and Money > LowerLimit) ->
    commit_withdrawal(Money, Limit);
  {abort, Trans} and {amount, Transaction, Money}
      when (Trans == Transaction) ->
    abort_withdrawal(Transaction, Money)
after Timeout -> abort_withdrawal_timeout(Transaction) end
```

Listing 5. Guards and timeout used in cash withdrawal in JErlang

Guards and timeouts in joins have often been omitted in Join-Calculus implementations due to complexity and performance issues. Yet it was important to include them in our work as they are commonly used in Erlang. In listing 5 we use guards (using **`when`**) to perform sanity checks for the withdrawal transaction that was requested by an external user and a timeout (**`after`**) to abort withdrawal actions that take too long. As in Erlang, guards in JErlang cannot contain side effects, therefore assignments or user-defined functions are prohibited.

Unlike all of the other Join-Calculus's implementations JErlang allows for non-linear patterns. In other words patterns in joins can contain the same unbound variables and therefore synchronise on their values as well (due to Erlang's single value assignment) in a structural equivalence manner. Listing 6 presents a shorter version of example 3. Non-linear patterns are often used by Erlang programmers in function headers or simple matching with variables. Non Erlang programmers might initially regard it as a possible source of confusion when mixing bound and unbound variables, yet as we said its usage is very common in the original language.

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

Listing 6. A one-cell buffer in JErlang

JErlang introduces an optional propagation attribute which allows developers to say that whenever a pattern matches, all the unbound variables in it should become bound in the join's body but the message itself should not be removed from the mailbox. To enable it, the programmer wraps the pattern with the **`prop`** closure. Propagation is not known in the Join-Calculus world but introduced successfully in Constraint Handling Rules[7,6]. It can obviously be implemented by implicitly sending the same message in the body of the join (see Haskell prototype by Lam and Sulzmann in [8]), however we are convinced that our feature is more readable and less error prone. More importantly, whenever a search is performed on the mailbox again, the message will not be placed at the end of the queue, and thus will get higher priority so that the matching should be performed faster. Dynamic propagation within the body would significantly deteriorate the clarity of the matching logic, as for example it is unclear at which point the other messages should be discarded. Listing 7 presents an authorisation procedure using propagation.

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

Listing 7. Session support using propagation in JErlang

JErlang, as in Erlang, allows for the creation of synchronous calls. This can be achieved by appending a process identifier value to the message, so that the receiver knows where to send the reply.

```
accept(Key) -> jerlang_gen_joins:call(dest, {accept, self(), Key}).
enter(Value) -> jerlang_gen_joins:call(dest, {enter, Value}).
valid(Amount) -> jerlang_gen_joins:cast(dest, {valid, Amount}).

handle_join({accept, PidA, Key} and {enter, Pid1, Val1} and
            {enter, Pid2, Val2} and {valid, 2}, State) ->
 {[{reply, {ok, 2}}, {reply, {ok, Key}}, {reply, {ok, Key}}, noreply],
     [{Key, Pid, Val1, Val2} | State] }      %% new state
```

Listing 8. Barrier synchronisation in **gen_joins** with combination of synchronous and asynchronous messages

To provide full conformance with the Erlang infrastructure we decided to implement an extension of **gen_server**[1], a popular design pattern used for building complex, fault-tolerant client-server applications. **gen_joins** is a natural extension of the **gen_server** design pattern that allows for the definition of joins, i.e. for synchronisation on multiple synchronous and asynchronous messages (calls). Listing 8 shows an extract of a JErlang program that has synchronous (**accept, enter**) and asynchronous (**valid**) tuple messages in the join. We follow Erlang's standards, where the former is represented by execution of **call** and the latter by **cast**. Functions for sending the messages (**dest** is just the name of the target process) and a separate callback function **handle_join** allow for clear separation of the API from the server implementation. The callback function mirrors the action of **receive** with the additional parameter (here named **State**) representing the internal state of the server process. Asynchronous messages should always return a **noreply** value,

[1] See `http://erlang.org/doc/man/gen_server.html`

whereas synchronous ones can either return a value (that conforms to the structure {**reply, ReplyVal**}) or **noreply**. In the latter case the caller will stall forever or timeout.

4 Implementation

The main problem with implementing joins inside Erlang's VM was the lack of the necessary operators to enable us to manipulate and inspect the processes' mailboxes. Apart from that we were constrained by consistency, intuitiveness and determinism of execution of the standard Erlang, so that current programmers feel eager to try out our extension. We decided to implement two different systems, both of which include a transformation module (**parse_transform**, explained later) that can produce valid Erlang code:

– The pure library version. It supports an internal queue that fetches, analyses and stores messages in the same order as they appear in the VM mailbox. The main drawback lies in its performance.
– We provide low-level functions, constructs and logic to manipulate the mailboxes inside Erlang's VM and then use them from a higher-level JErlang library (different from the above) to provide the necessary joins logic. The main drawback lies in it providing a non-standard VM.

Join constructs are written using the familiar **receive** (or **handle_join**) construct. In order to facilitate this feature we use **parse_transform**, an experimental module available in Erlang's standard library, that allows the developers to use Erlang syntax, but with different semantics. The transform-function receives syntactically valid Erlang abstract syntax tree (AST), JErlang in our case, and creates a new semantically valid AST. The aim of our transformation is to find joins patterns, create necessary tests for patterns and joins (in the form of multiple anonymous functions) and the code that initiates the call to the library modules, which perform the actual join operations. This allows the programmers to write clear and intuitive join definitions without studying a library API.

```
test_receive(Input) ->
    A = 12,
    receive {ok, Input} and [A, Rest] -> valid end.
-------------------
FirstPartialTest  = fun({ok, Input}) -> true end,
SecondPartialTest = fun([A, _]) -> true end,
FinalTest  = fun([{ok, Input}, [A, Rest]]) -> true end.
```

Listing 9. Simple joins in JErlang and corresponding tests

Since in the implementation of JErlang, we perform isolated tests only for patterns, without taking into consideration the body of the join, the former would create multiple false *unused variable* warnings by the compiler. Therefore we perform a simplified (syntax) *Reaching Definitions Analysis* [9]. This allows us to create valid test functions for patterns: we leave the original name for the bound variables and substitute the unbound variables with the neutral _ as presented in listing 9. Without this analysis, line 6 would create an *unused variable* warning for header **`[A, Rest]`**.

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. To avoid *unbound variable* errors and execute the partial matching tests on non-linear patterns as soon as possible, we perform a variant of static *Live Variable Analysis* [9]. This enables us to determine whether the variable in the test for partial joins should be substituted with the neutral _ or left unchanged because it used more than once in the join. This way we can also eliminate the unsatisfiable branches in the joins solver quickly and still create valid code.

The Erlang VM executes the bytecode, called BEAM, which is the result of a few transformation phases on the initial Abstract Syntax Tree. As an example of execution we consider accessing the process' mailbox using the **`receive`** construct, which results roughly in the following set of steps:

1. Each process maintains a pointer to the last tested message within the mailbox. The pointer is re-set to the beginning of the queue only when first entering the **`receive`** construct.
2. Take the next message as the operand for matching.
3. Take the current instruction representing the pattern, and try to match it with the message from step 2.
   If matching is not successful, we go to step 4, otherwise we go to step 6.
4. If there are more patterns then we increment the current program counter (PC) and go to step 3. If we reached the last pattern and there are still some messages left then we update the pointer of the mailbox to the next message, update the PC to the first pattern of **`receive`** and execute step 2. Otherwise we go to step 5.
5. The VM sets up the timeout counter (if this were not done already), and the process is suspended. It will either be awakened by the timer (and jump to the timeout action) or by a new message (go to step 1).
6. A successful match frees the memory associated with the currently pointed-to message, sets the mailbox pointer to the head of the queue and jumps to the BEAM instruction associated with the pattern.

One of the constructs that we incorporated into the modified VM and which enabled us to parse the message queues more freely without immediate discards (and duplicate queues), is **`search`**. It follows the same syntax as a standard **`receive`**, yet has slightly different semantics. Namely it maintains a separate *search pointer* on the mailbox that is independent from the original mailbox's *pointer*. For large mailboxes and complicated join combinations, it could be the case that a large number of calls to the mailbox need to be made to do pattern tests. To improve the performance over a queue based implementation, we use orthogonally the *uthash*[2] hash tables for each JErlang process, which maps identifiers to the addresses of the messages.

To reduce the number of repeated (and often unnecessary) pattern matching, which most other Join-Calculus implementations do not do, we added a new data structure that serves as a cache for storage and retrieval of the partial results of the matching. The overhead is acceptable, because we store only the messages' indices. We also had to modify the already presented Erlang **`receive`** algorithm to incorporate the joins resolution mechanism. The simplified description, which follows the formal definition of the operational semantics, is given below:

1. Take the message from the queue and the first join.
2. Take the list of tests associated with the join and check the message on each of the patterns. For each successful test, we store the message's index in the cache of the corresponding pattern.
3. We go to step 6 if none of the patterns' caches was updated, otherwise to step 4.
4. We take the final test function, i.e. the one that checks all the patterns and guards together, associated with the join and run it on all possible permutations of the satisfying messages. We go to step 5 if there is at least one successful run, or step 6 otherwise.
5. Retrieve the associated messages for each pattern and execute the join's body in the new context that updates the previously unbound variables.
6. We take the next join and go to step 2. If the current join is the last one and there are still some messages left in the queue, we update the message pointer to the next message and go to step 1, otherwise we stall until a new message arrives to the process.[3]

In JErlang with the modified VM, step 1 uses the **`search`** construct. In the non-VM version we use a standard **`receive`** construct that matches any message and puts it into the "internal" library queue for analysis.

Joins that exist in the **`gen_joins`** behaviour offer more optimisation possibilities because unlike in **`receive`**, they are 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 running of the test function is independent of other factors. Another challenge introduced by **`gen_joins`** is the addition of the *status* variable[4]. Since the execution of a join may have side-effects on its value, joins that previously couldn't be fired may now have become successful. Our algorithm takes into account the possibility of a chain of join actions that does not involve analysis of any new messages (similar to [10]).

[2] `http://uthash.sourceforge.net`

[3] *Timeout* is treated as in the description of **`receive`**.

[4] The status variable allows for internal storage in the client-server pattern.

5 Optimisations

For performance reasons we incorporated ideas from the Rete algorithm [11], used for efficient solving of *Production Rule Systems*. Rete reduces the amount of redundant operations through partial evaluation, thus allowing for steady building of knowledge. RETE uses so called alpha- and beta-reductions to build an efficient network of information nodes representing knowledge. The former focuses on testing independent nodes, irrespective of any connections they can have, whereas the latter gradually, from the left-hand side, tries to find a satisfying connection. As a simplified example assume the existence of a statement consisting of *A, B, C* and *D* predicates. The alpha-reduction will correspond to testing *A*, *B*, *C* and *D* individually, and beta-reduction will incrementally check *A and B*, *A and B and C* and *A and B and C and D*.
We implemented the algorithm sequentially because:

- We still can profit from *First-Match* semantics.
- The order of the messages in the mailbox is preserved.
- JErlang has to preserve the no-shared-memory principle between processes.

JErlang's alpha-reduction is performed by having local test functions for each of the patterns of the joins. Beta-reduction has to be performed by having multiple test functions that perform partial checks of the joins. Checking for consistency is performed through the matching of the headers. Joins of length 1 would have a single beta function and for joins with *n* patterns, we produce *n - 1* beta functions. Listing 10 presents a **handle_join** function with 4 patterns and the corresponding beta-functions for the Rete algorithm.

```
handle_join( {operation, Id, Op} and {num, Id, A} and {num, Id, B}
                and {num, Id, C}, State) when (A > B) ->
  Res = Op([A,B,C]),
 {[{reply, {ok, Res}}, noreply, noreply, noreply], [Res | State]}.
%% ------------------------------
[fun([{operation, Id, _}, {num, Id, _}], _) -> true end,
 fun([{operation, Id, _}, {num, Id, _}, {num, Id, _}], _)
                                                  -> true end,
 fun([{operation, Id, _}, {num, Id, A}, {num, Id, B},
       {num, Id, _}], Status) when (A > B)    -> true end]
```

Listing 10. Multi-pattern join in **gen_joins** and the corresponding beta-reduction tests

In **gen_joins**, the internal state can be changed during the execution of the body of the join so care is taken to preserve possible knowledge instead of repeating all the partial tests when not necessary.

To improve the efficiency of the algorithm we prune branches of the search space that cannot be satisfied or that are satisfiable by some previous branches in the search order. Hence sequences of messages that only permute equal messages are pruned. This radically increased the performance of the Santa Claus problem solution (see Section 6). In example 10 guards and the status variable are applied only to the last test function. We believe that this (typical) construction insufficiently uses knowledge about the patterns, because variables **`A`** and **`B`** in line 7 already provide information necessary to use the guard **`A`** > **`B`**. Therefore we check the earliest beta-function to which we can apply guards and additional variables, so that the filtering of invalid message combinations is done as soon as possible. This feature is especially interesting for the case when JErlang has to handle very large mailboxes, an Achilles' heel of Erlang.

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. It is important to remember that each set of messages that satisfies a partial test from the joins increases the time to solve it. Therefore it is crucial to abort any incorrect sequence of messages as soon as it is possible. The analysis of the structure of the joins assigns a rank to each pattern, which depends on the number of variables that it shares with other patterns, taking into account the occurrence of the variables inside guards and the status parameter (if available). Using this information we can reorder the patterns in the join during compile time, without actually losing the deterministic ordering guarantee, gaining reasonable speed-ups for joins with multiple dependencies. This feature worked well along with the guard optimisations described above.

6 Evaluation

With JErlang we aimed to increase the expressiveness of Erlang for handling concurrency problems while keeping negative effects on performance as small as possible. One of the problems that drove the development of JErlang is the *Santa Claus* problem first defined by Trono [12]. In this synchronisation problem, *Santa* sleeps at the North Pole waiting to be awakened by nine *reindeer* or three *elves* and then performs work with them. However the waiting group of the former has higher priority if both full groups gather at the same time.

```
receive
    {reindeer, Pid1} and {reindeer, Pid2} and {reindeer, Pid3}
      and {reindeer, Pid4} and {reindeer, Pid5} and {reindeer, Pid6}
      and {reindeer, Pid7} and {reindeer, Pid8} and {reindeer, Pid9} ->
        io:format("Ho, ho, ho! Let's deliver presents!~n"),
        [Pid1, Pid2, Pid3, Pid4, Pid5, Pid6, Pid7, Pid8, Pid9];
    {elf, Pid1} and {elf, Pid2} and {elf, Pid3} ->
        io:format("Ho, ho, ho! Let's discuss R&D possibilities!~n"),
        [Pid1, Pid2, Pid3]
end
```

Listing 11. Santa Claus solution in JErlang

Many solutions were proposed, using semaphores (or similar), but since the advent of the Join-Calculus a more elegant solution is possible. We compare our JErlang solution with one provided by Richard A. O'Keefe [5] written in Erlang. Listing 11 presents an extract from our solution (the pids allow us to reply to the processes representing the *reindeer* and *elves*). With JErlang we are able to say: "Synchronise on 9 *reindeer* or 3 *elves*, with the priority given to the former". Typically the priority remains the hardest part to solve but with our *First-Match* semantics we get it for free. An Erlang solution contains multiple nested receive statements therefore it is hard to understand immediately what is the aim of the code and how the priority is resolved. The JErlang version of *Santa Claus* is half the size of the original Erlang version.

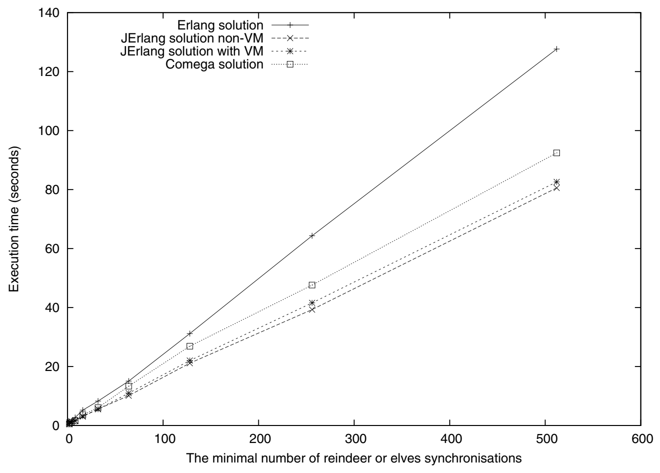

Fig. 1. Execution of the *Santa Claus* problem in **gen_joins**, implemented in Erlang as well as JErlang with and without VM support. The problem artificially limits the synchronisation to measure the correct time of execution.

Our tests have shown that VM-supported JErlang gives better results but only for larger mailboxes. The overhead that we introduce is too big for small mailboxes. Figure 1 presents an average time of execution of the *Santa Claus* problem in Erlang, Cϖ(using implementation from [4]), JErlang with and without the VM support. The difference between the two JErlang versions is minimal because consumers work at a similar rate as producers. Unsurprisingly, the time of execution of the simulation increases linearly with the number of required synchronisations. It is interesting that the optimised JErlang is faster than the one implemented using Erlang, as a result of our optimisations, as one would expect a manual, specific and overhead-free implementation to be more efficient.

[5] http://www.cs.otago.ac.nz/staffpriv/ok/santa

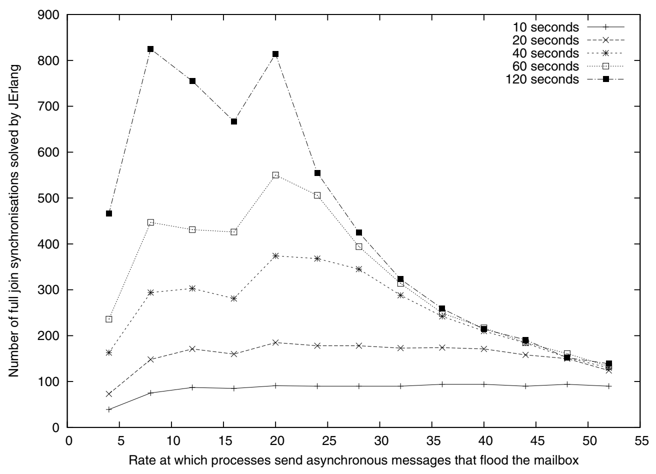

Fig. 2. VM supported JErlang benchmark. The effect of increasing the rate at which the processes produce asynchronous messages on the number of synchronisations that the join solver is able to analyse in a given quantity of time (different lines). We introduced random delays to clearly mark the tendency in the performance and therefore the real numbers of synchronisations are higher.

To experiment with the performance of joins we developed tests that create competitive, heavy-load client-server scenarios. Our aim was to develop situations where the numerous producers contributed to a sudden increase in the size of the mailbox. Additionally, we generated messages that do not necessarily match the specific patterns or won't match the full join, which is much closer to real systems. Figure 2 shows the drop in the number of synchronisations that our joins-solver was able to find, as we kept increasing the rate at which the messages were produced (exponentially), with the rate of messages that can actually perform a join remaining constant. The empirical results have shown that the difference between using the non-VM JErlang and a hash-map data structure inside the VM is negligible for small mailboxes, but for this benchmark we were seeing at least double boost in the performance for VM supported JErlang (omitted on the graphs). Additionally for simple benchmarking we compared different implementations of a single cell buffer (using the example from [4]) and the performance of JErlang was better from its Cϖ's equivalent (roughly 20%) but worse than the Scala version (10%), due to its speed for simple matching.

7 Related Work

The Join-Calculus has inspired many implementations, the first being JoCaml[3], an extension of OCaml. Compilation of patterns is described in [10] in the JoCaml implementation. In comparison to our work, JoCaml requires the uniqueness of patterns in the joins and forces an awkward syntax when using the same patterns in different join definitions. Guards are disallowed in JoCaml.

Cϖ (previously POLYPHONIC C#) [4] was introduced as an extension of the popular C# language where it introduces an object-oriented version of joins, known as *chords*. JERLANG's **gen_joins** is similar if we want to synchronise on multiple function calls but Cϖ allows for at most one synchronous method call in the whole *chord* and suffers from similar feature limitations as JOCAML. A Cϖ variant has been implemented as a library and can be used on the .NET platform with languages like VISUAL BASIC or C# itself.

In HASKELLJOINRULES[6] project the authors designed efficient implementation of *Constraint Handling Rules*(CHR), concepts which were shown in [13] to be comparable to JOIN-CALCULUS. HASKELLJOINRULES was one of the first to propose guards along with the join patterns. It also inspired us to include a propagation feature (included in CHR), however JERLANG, unlike HASKELLJOINRULES, proposes a more efficient approach instead of primitive re-send semantics. Sulzmann and Lam proposed a join construct for ERLANG in [6], but the semantics of their approach was vague and no prototype was built. For their HASKELL extension they focused on the implementation of a parallel solver for their *Constraints*[14,8] a possible approach for future development of JERLANG. This way though they lose some of the important guarantee like deterministic behaviour.

Eugster and Jayaram presented an extension of *Java* for creating efficient event-based systems. *EventJava* [15] allows for powerful expressions in guards, which enables it to build complex relations between asynchronous-only messages. In JERLANG, guard conditions can also span over multiple messages, but we are limited by ERLANG functionality to disallow new variables or control constructs in them. In *EventJava* the messages are transformed into a format acceptable by the off-the-shelf RETE framework. In our implementation we have focused on introducing optimisations which specifically target joins resolutions, because of better control and a lack of existing solutions in ERLANG. The *Java* extension has the option of streams yet no event can appear in more than one correlation pattern, which is reasonable for solving the problems presented in their work, but opposite to our implementation. Finally *EventJava* allows for assigning timeouts to specific events unlike JERLANG, which forces timeouts on whole joins on the receiver side. We believe that the former option would be more suitable to message queueing systems rather than JERLANG. Nevertheless the synchronous calls in **gen_joins** modules can specify timouts, therefore avoiding stalling for long running operations.

8 Conclusions and Future Work

This paper presents JERLANG, a JOIN-CALCULUS inspired extension of ERLANG. A number of examples have shown typical actor synchronisation problems that programmers encounter while writing concurrent applications, and how they can be solved using JERLANG primitives. Unlike other JOIN-CALCULUS implementations we tightly integrated joins semantics, by adding guards, timeouts, non-linear patterns and propagation to the original idea. The implementation of

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

these features allowed us to explore the role of *First-Match* semantics in JERLANG programs.

We intend to experiment further with the various join techniques to explore the performance optimisations possibilities. It would be interesting to compare the limits of the sequential algorithms to the parallel, out-of-order joins solver and find a reasonable trade-off between JERLANG's performance and expressiveness. More in-depth static analysis of the patterns, as well as better data structures for mailboxes, promises further optimisations.

Acknowledgements. We would like to thank our shepherd, Patrick Eugster, and the reviewers for their helpful comments.

References

1. Armstrong, J.: Programming Erlang: Software for a Concurrent World. Pragmatic Bookshelf (July 2007)
2. Gonthier, G., Rocquencourt, I.: The reflexive CHAM and the Join-Calculus. In: Proceedings of the 23rd ACM Symposium on Principles of Programming Languages, pp. 372–385. ACM Press, New York (1996)
3. Mandel, L., Maranget, L.: JoCaml Documentation and Manual (Release 3.11). INRA (2008)
4. Benton, N., Cardelli, L., Polyphonic, C.: Modern Concurrency Abstractions for C#. ACM Trans. Program. Lang. Syst., 415–440 (2002)
5. Haller, P., Van Cutsem, T.: Implementing Joins using Extensible Pattern Matching. In: Lea, D., Zavattaro, G. (eds.) COORDINATION 2008. LNCS, vol. 5052, pp. 135–152. Springer, Heidelberg (2008)
6. Sulzmann, M., Lam, E.S.L., Weert, P.V.: Actors with multi-headed message receive patterns. In: Lea, D., Zavattaro, G. (eds.) COORDINATION 2008. LNCS, vol. 5052, pp. 315–330. Springer, Heidelberg (2008)
7. Sulzmann, M., Lam, E.S.: Haskell - join - rules. In: Chitil, O., Horváth, Z., Zsók, V. (eds.) IFL 2007. LNCS, vol. 5083, pp. 195–210. Springer, Heidelberg (2008)
8. Lam, E.S., Sulzmann, M.: Parallel join patterns with guards and propagation (2009)
9. Nielson, F., Nielson, H.R., Hankin, C.: Principles of Program Analysis. Springer, New York (1999)
10. Maranget, L., Fessant, F.L.: Compiling join-patterns. Electronic Notes in Computer Science. Elsevier Science Publishers, Amsterdam (1998)
11. Forgy, C.: RETE: a fast algorithm for the many pattern/many object pattern match problem. Artificial Intelligence 19, 17–37 (1982)
12. Trono, J.A.: A new exercise in concurrency. SIGCSE Bull. 26(3), 8–10 (1994)
13. Lam, E., Sulzmann, M.: Finally, a comparison between Constraint Handling Rules and Join-Calculus. In: Fifth Workshop on Constraint Handling Rules, CHR 2008 (2008)
14. Sulzmann, M., Lam, E.S.: Compiling Constraint Handling Rules with lazy and concurrent search techniques. In: CHR 2007, pp. 139–149 (2007)
15. Eugster, P., Jayaram, K.R.: Eventjava: An extension of java for event correlation. In: Drossopoulou, S. (ed.) ECOOP 2009. LNCS, vol. 5653, pp. 570–594. Springer, Heidelberg (2009)