# Actors with Multi-Headed Message Receive Patterns

Martin Sulzmann[1], Edmund S. L. Lam[1] and Peter Van Weert[2⋆]

[1] School of Computing, National University of Singapore
S16 Level 5, 3 Science Drive 2, Singapore 117543
{sulzmann,lamsoonl}@comp.nus.edu.sg
[2] Department of Computer Science, Katholieke Universiteit Leuven
Celestijnenlaan 200A, B-3001 Heverlee, Belgium
Peter.VanWeert@cs.kuleuven.be

**Abstract.** The actor model provides the programmer with high-level concurrency abstractions to coordinate simultaneous computations by sending and receiving messages. Languages implementing the actor model such as Erlang commonly only support single-headed pattern matching over received messages. We propose and design an extension of Erlang style actors with receive clauses containing multi-headed message patterns. Patterns may be non-linear and constrained by guards. We provide a number of examples to show the usefulness of the extension. We also explore the design space for multi-headed message matching semantics, for example first-match and rule priority-match semantics. The various semantics are inspired by multi-set constraint matching semantics found in Constraint Handling Rules which provides us with a formal model to study actors with multi-headed message receive patterns. The system can be implemented efficiently and we have built a prototype as a library-extension to Haskell.

## 1 Introduction

We all know by now that the free lunch is over. We must write concurrent programs to take advantage of the next generation of multi-core architectures. But writing correct concurrent programs using the traditional model of threads and locks is an inherently difficult and error-prone task. Message-based concurrency provides the programmer with the ability to exchange messages without relying on low-level locking and blocking mechanisms. A particular popular form of message-based concurrency is actor style concurrency [1] as implemented by the Erlang language [2].

In Erlang, an actor comes with an asynchronous message queue also known as mailbox. Erlang actors communicate by sending and receiving messages. Sending is a non-blocking (asynchronous) operation. Each sent message is placed in the actors mailbox and immediately returns to the sender. Messages are processed via receive clauses which resemble pattern matching clauses found in functional/logic languages. We select what message to receive by matching the

⋆ Research Assistant of the Research Foundation – Flanders (FWO-Vlaanderen)

message with the receive clauses in sequential order. The receive operation is blocking. If none of the receive clauses applies we suspend until a matching message is delivered.

Receive clauses in Erlang have the restriction that they consist of a *single-headed* message pattern. That is, each receive pattern matches at most one message, possibly constrained by a guard. There are situations where we wish to match against multiple messages. For example, for a seller we want to find a matching buyer. Via *multi-headed* message patterns we can give a direct encoding of such problems. But such patterns are not commonly supported in Erlang style languages. The programmer herself must therefore either explicitly keep track of the set of partial matches or resort to nested received clauses. But this leads to clumsy and error-prone code as we will see later in Section 2.

In this paper, we make the following contributions:

- We propose and design an extension of Erlang style actors with receive clauses containing multi-headed message patterns. Patterns may be non-linear (i.e. have multiple occurrences of the same pattern variable) and be constrained by guards. There are several possibly ways how to define multi-head message matching, for example either first-match or rule priority-match. We explore both alternatives in detail (Section 4).
- We have implemented a library-based prototype in Haskell (Section 5).

We draw our inspiration from prior work in the concurrent constraint logic programming community. Specifically, we adopt the various multi-set constraint matching semantics found in Constraint Handling Rules. Section 3 provides the necessary background information. We discuss related work in Section 6. Section 7 concludes and discusses some possible future work.

We assume that the reader has some basic familiarity with Erlang and functional languages such as Haskell. We will write example programs in Haskell syntax [16] extended with actors. The Haskell extension uses some minor syntactic sugar compared to our library-based extension described in Section 5. Throughout the paper whenever we refer to actors we mean Erlang style actors.

## 2 Motivating Example

We motivate multi-headed message receive patterns via a classic concurrency challenge, the Santa Claus problem [19]. We give the original specification as well as a variation to discuss alternative styles of message matching semantics. First, we describe the problem specification which is common to both.

**Common:** Santa repeatedly sleeps until wakened by either all of his nine reindeer, back from their holidays, or by a group of three of his ten elves. If awakened by the reindeer, he harnesses each of them to his sleigh, delivers toys with them and finally unharnesses them (allowing them to go off on holiday). If awakened by a group of elves, he shows each of the group into his study, consults with them on toy R&D and finally shows them each out (allowing them to go back to work).

The original specification additionally demands the following.

**Priority-Cond (Original):** Santa should give *priority* to the reindeer in the case that there is both a *matching* group of elves and a group of reindeer waiting.

In our variant, we substitute (Priority-Cond) as follows.
**First-Match-Cond (Variant):** Santa chooses the *first matching* group of either elves or reindeer waiting.

First, we consider the (First-Match-Cond) variant of the Santa Claus problem. We give a solution in Haskell extended with Erlang style actors. We omit some unimportant tasks such as "deliver toys" and "show study".

**Santa Variant.** We represent Santa via an actor which waits for either a deer or an elf message. Each message is represented as a constructor of Santa's message data type. Deer and elves may appear in random order in Santa's mailbox. Initially, we send eleven elves and nine deer. The critical task for Santa is to check for nine reindeer and three elves. Santa will pick the group whichever arrives first. We achieve this by accumulating the set of deer and elves received so far. See function `santa`. We are slightly more explicit compared to Erlang in that the `receive` primitive takes the actor as the first argument and then the receive clauses as second argument (similar to case statements).

The actual behavior is like in Erlang. Receive clauses are are tried from top to bottom, for one message at a time. If a message does not match any of the clauses we try the next message. In essence, this characterizes a *first-match semantics*. In our case, we first match the current message against the deer pattern. If the match fails, we check for an elf. If this match fails as well, we move on to the next message and the process repeats itself. If none of the messages match we block and wait for new messages to arrive. This case will not apply here because there are only deer or elf messages. The receive clauses are exhaustive.

```
data SantaMsg = Deer Int |  Elf Int

santa sanActor noOfDeer noOfElves DeerAcc ElvesAcc =
  receive sanActor of
     Deer x -> if length (Deer x:DeerAcc) == noOfDeer
               then ‘‘Deliver toys etc’’
               else santa sanActor noOfDeer noOfElves
                          (Deer x:DeerAcc) ElvesAcc
     Elf x ->  if length (Elf x:ElvesAcc) == noOfElves
               then ‘‘Show study etc’’
               else santa sanActor noOfDeer noOfElves
                          DeerAcc (Elf x:ElvesAcc)
```

Via multi-headed message patterns we can omit the accumulation of partial matches entirely. Here is a solution using our proposed multi-head extension.

```
santa2 sanActor =
  receive sanActor of
    Deer x1, Deer x2, Deer x3, Deer x4, Deer x5,
    Deer x6, Deer x7, Deer x8, Deer x9 -> ‘‘Deliver toys etc’’
    Elf x1, Elf x2, Elf x3 -> ‘‘Show study etc’’
```

We will explain the semantics for such an extension in terms of multi-set constraint matching semantics studied in the context of Constraint Handling Rules (CHR) [5]. CHR is a concurrent committed-choice constraint logic programming language to transform (rewrite) multi-sets of constraints into simpler

ones. Constraints correspond to messages, and the left-hand side of a CHR rule corresponds to the pattern of a receive clause. Concretely, we adopt the refined CHR semantics [3] which finds a match for the left-hand side of a CHR rule by processing constraints in sequential order and testing CHR rules from top to bottom. For the special case of single-headed CHR rules, this is essentially the first-match actor semantics. The refined CHR semantics provides for a formal basis to extend the first-match actor semantics with multi-headed message receive patterns involving guards.

Suppose not every group of three elves is compatible. For example, either only odd or even numbered elves are willing to work together. We can directly impose this condition in the multi-head solution by including a guard condition:

```
santa3 sanActor =
  receive sanActor of
    Deer x1, Deer x2, Deer x3, Deer x4, Deer x5,
    Deer x6, Deer x7, Deer x8, Deer x9 -> ‘‘Deliver toys etc’’
    Elf x1, Elf x2, Elf x3 when allOddorEven [x1,x2,x3] -> ‘‘Show study etc’’
 where
   allOddorEven xs = (and (map xs odd)) || (and (map xs even))
```

Further examples showing the usefulness of multi-headed message receive patterns in combination with guards and non-linear patterns are given in Section 4.

**Santa Original** Under a first-match semantics, our previous solutions `santa` and `santa2` do not obey the priority given to a group of deer. We must program the (Priority-Cond) condition explicitly via “otherwise” and nested receive statements.

```
santa4 sanActor =
  receive sanActor of
    Deer x1, Deer x2, Deer x3, Deer x4, Deer x5,
    Deer x6, Deer x7, Deer x8, Deer x9 -> ‘‘Deliver toys etc’’
    otherwise -> receive sanActor of
                   Elf x1, Elf x2, Elf x3 -> ‘‘Show study etc’’
                   otherwise -> santa4 sanActor
```

We first check if there are nine deer (waiting) in Santa’s mailbox. Otherwise, we call a nested receive statement to check for three elves. Otherwise, the process repeats itself. The statement `otherwise` corresponds to `after 0` in Erlang. This branch applies if none of the other branches could find a match after waiting for 0 seconds.

The above raises the issue whether for concurrency problems with priorities we need a different semantics in which receive clauses are executed in (textual) order. Such a rule priority-match semantics allows for a more declarative specification of the original Santa Claus problem. Incidentally, in the CHR literature such a semantics has been recently suggested [13, 14]. If adopted to the actor setting, function `santa2` then immediately solves the original Santa Claus problem with (Priority-Cond).

**Summary.** Thanks to multi-headed message receive patterns the programmer does not need to build the set of partial matches herself which can be non-trivial.

**Constraints and terms**

$$
\begin{array}{lll}
t & ::= F\ t...t & \text{Term function} \\
 & \mid\ x & \text{Term variable} \\
c & ::= K\ t...t & \text{Constraint} \\
cn & ::= c\#n & \text{Numbered constraint} \\
co & ::= c : j & \text{Occurrence constraint} \\
cno & ::= c\#n : j & \text{Active constraint}
\end{array}
$$

**Substitution**

$$\theta ::= [c_1/x_1, ..., c_n/x_n]$$

**Rule patterns**

$$
\begin{array}{lll}
H & ::= co \mid H \wedge H & \text{Head} \\
G & ::= e & \text{Guard} \\
RP & ::= H \text{ when } G & \text{Rule pattern} \\
 & \mid\ H & \\
\mathcal{RP} & ::= \{RP_1, ..., RP_n\} & \text{Set of rule patterns}
\end{array}
$$

**Executables**

$$
\begin{array}{l}
M ::= MN \mid [cno|MN] \\
MN ::= [] \mid [cn|MN]
\end{array}
$$

**Store**

$$St ::= [] \mid [cn|St]$$

**Matching States**

$$
\begin{array}{ll}
\langle M, St\rangle & \text{Intermediate} \\
\langle M, St, \theta, RP\rangle & \text{Successful} \\
\langle [], St\rangle & \text{Failure}
\end{array}
$$

**Fig. 1.** Constraint Handling Rules Essential Syntax

In combination with guards this leads to more concise and maintainable code. Erlang style actors follow the first-match semantics The refined CHR semantics is a conservative extension of such a semantics to the setting of multi-set matching involving guards. Certain concurrency problems are more naturally solved using a rule priority-match semantics which has also been explored in the CHR context.

In the up-coming section, we provide background information on the first-match and rule priority-match CHR semantics. In Section 4, we formalize an extension of actors with multi-headed message patterns which can be constrained by guards. The extension is parametric in terms of the underlying message match semantics for receive clauses. In case we adopt a first-match CHR style semantics for message patterns, we obtain a conservative extension of Erlang style actors.

## 3 Constraint Handling Rules Matching

We review the essentials of the multi-set constraint matching semantics of Constraint Handling Rules (CHR). The actual CHR framework is much richer than presented here. CHR also supports constraint propagation and built-in constraints such as unification constraints. We ignore these additional features.

Figure 1 introduces some basic syntactic categories. Constraints carry a distinct number to distinguish multiple appearances of a constraint $c$. The signifi-

**Matching reduction:** $\langle M, St\rangle \longrightarrow_{First-\mathcal{RP}} \langle M, St\rangle$ and $\langle M, St\rangle \longrightarrow_{First-\mathcal{RP}} \langle M, St, \theta, RP\rangle$

$$\text{(Activate)} \qquad \langle [c\#n|M], St\rangle \longrightarrow_{First-\mathcal{RP}} \langle [c\#n:1|M], St\rangle$$

$$\text{(Match)} \quad \frac{\begin{array}{c} H_1, c' : j, H_2 \text{ when } G \in \mathcal{RP} \\ \theta(G) \text{ evaluates to } \mathit{True} \quad St_1 +\!\!+ St_2 +\!\!+ St' =_{set} St \\ \theta(c') = c \quad \theta(H_1) = St_1 \quad \theta(H_2) = St_2 \quad \text{for some } \theta \end{array}}{\langle [c\#n : j, M], St\rangle \longrightarrow_{First-\mathcal{RP}} \langle M, St', \theta, H_1 \wedge c' : j \wedge H_2 \text{ when } G\rangle}$$

$$\text{(Continue)} \quad \frac{j < maxOccur(\mathcal{RP})}{\langle [c\#n : j|M], St\rangle \longrightarrow_{First-\mathcal{RP}} \langle [c\#n : j+1|M], St\rangle}$$

$$\text{(Deactivate)} \quad \frac{j \geq maxOccur(\mathcal{RP})}{\langle [c\#n : j|M], St\rangle \longrightarrow_{First-\mathcal{RP}} \langle M, St +\!\!+ [c\#n]\rangle}$$

$$\text{(Step1)} \quad \frac{\langle M, St\rangle \longrightarrow_{First-\mathcal{RP}} \langle M', St'\rangle}{\langle M, St\rangle \longrightarrow^*_{First-\mathcal{RP}} \langle M', St'\rangle}$$

$$\text{(Step2)} \quad \frac{\langle M, St\rangle \longrightarrow_{First-\mathcal{RP}} \langle M', St', \theta, RP\rangle}{\langle M, St\rangle \longrightarrow^*_{First-\mathcal{RP}} \langle M', St', \theta, RP\rangle}$$

$$\text{(Trans)} \quad \frac{\langle M_1, St_1\rangle \longrightarrow^*_{First-\mathcal{RP}} \langle M_2, St_2\rangle \quad \langle M_2, St_2\rangle \longrightarrow^*_{First-\mathcal{RP}} \langle M_3, St_3\rangle}{\langle M_1, St_1\rangle \longrightarrow^*_{First-\mathcal{RP}} \langle M_3, St_3\rangle}$$

**Fig. 2.** CHR Multi-Set First Match Semantics

cance of occurrence and active constraints becomes clear shortly when we discuss two particular matching strategies: first-match and rule priority-match. Initially, all (numbered) constraint are stored in a list $M$. We use Prolog syntax to denote a list $[x|xs]$ with first element $x$ and tail $xs$. The symbol [] denotes the empty list and $+\!\!+$ denotes list concatenation. In the CHR context, $M$ is referred to as the execution stack. Here, it is more appropriate to view $M$ as a list corresponding to the actors mailbox. Constraints in $M$ are executed in sequential order to find a match with a rule pattern. Constraints not contributing to a match are put into the store. In CHR speak we say they are deactivated. Rule patterns contain a guard component which must be an expression evaluating to a Boolean value. A CHR rule also consists of a rule body which we ignore here. We are only interested in the multi-set constraint match semantics of CHR and not in CHR execution. In the actor context, a rule pattern corresponds to a receive pattern and a rule body corresponds to the body of a receive clause.

**First-Match Semantics.** Our presentation largely follows the CHR description [13, 14], which we adapt to our specialized setting. Figure 2 describes the CHR multi-set first-match semantics as a transition system $\longrightarrow^*_{First-\mathcal{RP}}$ among states $\langle M, St\rangle$ where $M$ represents the constraints to be executed and

**Matching reduction:** $\langle M, St\rangle \longrightarrow^*_{Priority-\mathcal{RP}} \langle M, St\rangle$ and $\langle M, St\rangle \longrightarrow^*_{Priority-\mathcal{RP}} \langle M, St, \theta, RP\rangle$

$$\text{(Succ)}\ \frac{\begin{array}{c}\mathcal{RP} = \{RP_1, ..., RP_n\} \\ \forall 1 \leq j < i\ \langle M, St\rangle \longrightarrow^*_{First-\{RP_j\}} \langle [], St'_j\rangle \\ \langle M, St\rangle \longrightarrow^*_{First-\{RP_i\}} \langle M', St', \theta, RP_i\rangle\end{array}}{\langle M, St\rangle \longrightarrow^*_{Priority-\mathcal{RP}} \langle M', St', \theta, RP_i\rangle}$$

$$\text{(Fail)}\ \frac{\begin{array}{c}\mathcal{RP} = \{RP_1, ..., RP_n\} \\ \forall 1 \leq j \leq n\ \langle M, St\rangle \longrightarrow^*_{First-\{RP_j\}} \langle [], St'_j\rangle\end{array}}{\langle M, St\rangle \longrightarrow^*_{Priority-\mathcal{RP}} \langle [], St_n\rangle}$$

**Fig. 3.** CHR Multi-Set Rule Priority Match Semantics

$St$ holds the already processed constraints. The set $\mathcal{RP}$ holds the rule patterns. Initially, we start in the state $\langle M, []\rangle$. The goal is to reach a successful state $\langle M', St, \theta, RP\rangle$ where $RP$ is the (first) rule pattern matched by a (sequentially processed) sequence of constraints in $M$, $\theta$ is the matching substitution, $St$ holds the already processed constraints which did not contribute to the match and $M'$ are the remaining constraints from $M$. State $\langle [], St\rangle$ indicates failure. None of the constraints in the initial $M$ could trigger a rule pattern.

Constraints in rule heads have distinct, increasing occurrences with respect to their textual order in a program. For example, the rule heads derived from the `santa3` function in the previous section are $Deer\ x_1 : 1 \wedge ... \wedge Deer\ x_9 : 9$ and $Elf\ x_1 : 10 \wedge Elf\ x_2 : 11 \wedge Elf\ x_3 : 12$. Hence, the search for a match is performed by activating the leading constraint in $M$ by assigning it the occurrence number 1. See rule (Activate). Rule (Match) checks whether the active constraint matches a constraint in the head of a rule pattern at the respective position. We consult the store to find constraints $St_1$ and $St_2$ which match the remaining constraints $H_1$ and $H_2$ in the head. The symbol $=_{set}$ denotes set equality among lists. $St_1 =_{set} St_2$ holds if each element in $St_1$ appears in $St_2$ and vice versa. We assume that the equality test among constraints ignores numbering of constraints and occurrences. If the guard can be satisfied as well, we report the successfully found match. Otherwise, we continue our search by incrementing the occurrence number of the active constraint. See rule (Continue). This is only sensible if the maximum occurrence in any constraint in $\mathcal{RP}$, computed via function $maxOccur(\cdot)$, is smaller than the current occurrence number. Otherwise, we deactivate the constraint by putting it into the store. See rule (Deactivate). The order among messages is retained. That is, for any initial state $\langle M, []\rangle$ and intermediate state $\langle M', St\rangle$ we have that $M = St +\!\!+ M'$. We keep repeatedly applying rules (Activate), (Match), (Continue) and (Deactivate), in that order, until we either reach a successful or failure state. To summarize, the first-match semantics finds a match by processing constraints in sequential order and checking for a matching rule pattern from top to bottom (in the textual order).

**Priority-Match Semantics.** We consider a rule priority-match semantics which guarantees that rule patterns are executed in (textual) order. Figure 3

**Receive clause:**

```
receive act of
    A,A -> "RP1"      -- RP1 = A:1 ∧ A:2
    B   -> "RP2"      -- RP1 = B:3
```

$$\mathcal{RP} \;=\; \{RP_1, RP_2\}$$

**First-Match reduction:**

$$\begin{array}{ll} & \langle [A\#1, B\#2, A\#3], [] \rangle \\ \text{(Act-Cont-Deact)} & \longrightarrow_{First-\mathcal{RP}} \langle [B\#2, A\#3], [A\#1] \rangle \\ \text{(Activate)} & \longrightarrow_{First-\mathcal{RP}} \langle [B\#2:1, A\#3], [A\#1] \rangle \\ \text{(Continue } \times 2) & \longrightarrow_{First-\mathcal{RP}} \langle [B\#2:3, A\#3], [A\#1] \rangle \\ \text{(Match)} & \longrightarrow_{First-\mathcal{RP}} \langle [A\#3], [A\#1], \mathit{identSubst}, RP_2 \rangle \end{array}$$

**Priority-Match reduction:**

$$\begin{array}{ll} & \langle [A\#1, B\#2, A\#3], [] \rangle \\ \text{(Act-Cont-Deact)} & \longrightarrow_{First-\{RP_1\}} \langle [B\#2, A\#3], [A\#1] \rangle \\ \text{(Act-Cont-Deact)} & \longrightarrow_{First-\{RP_1\}} \langle [A\#3], [A\#1, B\#2] \rangle \\ \text{(Activate)} & \longrightarrow_{First-\{RP_1\}} \langle [A\#3:1], [A\#1, B\#2] \rangle \\ \text{(Match)} & \longrightarrow_{First-\{RP_1\}} \langle [], [B\#2], \mathit{identSubst}, RP_1 \rangle \end{array}$$

**Fig. 4.** Example 1

contains the details. We apply the first-match semantics on each rule pattern and select the first successful match in textual order. We assume that $RP_j$ appears before $RP_{j+1}$ in the program which can be specified via occurrences associated to head constraints.

Next, we consider some examples to illustrate the differences between both semantics.

**Examples.** The first example is given in Figure 4. We assume that $A$ and $B$ are constant messages. Therefore, each (Match) reductions make use of the identity (matching) substitutions *identSubst*. Each reduction step is annotated with the corresponding reduction rule. For brevity, we shorten reduction steps. For example, we write

$$\begin{array}{ll} & \langle [A\#1, B\#2, A\#3], [] \rangle \\ \text{(Act-Cont-Deact)} & \longrightarrow_{First-\mathcal{RP}} \langle [B\#2, A\#3], [A\#1] \rangle \end{array}$$

as a short-hand for

$$\begin{array}{ll} & \langle [A\#1, B\#2, A\#3], [] \rangle \\ \text{(Activate)} & \longrightarrow_{First-\mathcal{RP}} \langle [A\#1:1, B\#2, A\#3], [] \rangle \\ \text{(Continue } \times 3) & \longrightarrow_{First-\mathcal{RP}} \langle [A\#1:4, B\#2, A\#3], [] \rangle \\ \text{(Deactivate)} & \longrightarrow_{First-\mathcal{RP}} \langle [B\#2, A\#3], [A\#1] \rangle \end{array}$$

The first-match reduction applies $RP_2$. We sequentially process constraints, searching for the first match for a rule pattern from top to bottom. Starting with the initial list of executables $[A\#1, B\#2, A\#3]$, we find that $B\#2$ form the first match for rule pattern $RP_2$. On the other hand the priority-match reduction applies $RP_1$. We strictly apply rule patterns in (textual) order. Based on the priority of rules, $A\#1, A\#3$ form a match for the first rule pattern $RP_1$.

**Receive clause:**

```
receive act of
    A,A -> "RP1"      -- RP1 = A:1 ∧ A:2
    B   -> "RP2"      -- RP1 = B:3
```

$$\mathcal{RP} \;=\; \{RP_1, RP_2\}$$

**First-Match reduction:**

$$\begin{array}{ll} & \langle [A\#1, B\#2], [] \rangle \\ \text{(Act-Cont-Deact)} & \longrightarrow_{First-\{RP_1,RP_2\}} \langle [B\#2], [A\#1] \rangle \\ \text{(Activate)} & \longrightarrow_{First-\{RP_1,RP_2\}} \langle [B\#2:1], [A\#1] \rangle \\ \text{(Continue } \times 2) & \longrightarrow_{First-\{RP_1,RP_2\}} \langle [B\#2:3], [A\#1] \rangle \\ \text{(Match)} & \longrightarrow_{First-\{RP_1,RP_2\}} \langle [], [A\#1], \mathit{identSubst}, RP_2 \rangle \end{array}$$

**Priority-Match reductions:**

$$\begin{array}{ll} & \langle [A\#1, B\#2], [] \rangle \\ \text{(Act-Cont-Deact)} & \longrightarrow_{First-\{RP_1\}} \langle [B\#2], [A\#1] \rangle \\ \text{(Act-Cont-Deact)} & \longrightarrow_{First-\{RP_1\}} \langle [], [A\#1, B\#2] \rangle \end{array}$$

$$\begin{array}{ll} & \langle [A\#1, B\#2], [] \rangle \\ \text{(Act-Cont-Deact)} & \longrightarrow_{First-\{RP_2\}} \langle [B\#2], [A\#1] \rangle \\ \text{(Activate)} & \longrightarrow_{First-\{RP_2\}} \langle [B\#2:1], [A\#1] \rangle \\ \text{(Continue } \times 2) & \longrightarrow_{First-\{RP_2\}} \langle [B\#2:3], [A\#1] \rangle \\ \text{(Match)} & \longrightarrow_{First-\{RP_2\}} \langle [], [A\#1], \mathit{identSubst}, RP_2 \rangle \end{array}$$

**Fig. 5.** Example 2

In the second example in Figure 5, we apply the first-match and priority-match on the initial list of executables $[A\#1, B\#2]$. Clearly, in both cases only rule pattern $RP_2$ applies. The first-match reduction is almost identical to Example 1 where we additionally find constraint $A\#3$ in the initial $M$. But this constraint does not contribute to the first match. In case of the priority-match reduction we first try $RP_1$ which fails and then we try $RP_2$ which leads to success.

## 4 Actors with Multi-Headed Message Patterns

Figure 6 introduces the syntax and Figures 7 and 8 introduce the semantics of an elementary actor language which supports multi-headed message patterns. In example programs, we will use "," (comma) to separate multi-headed message patterns whereas in our (internal) syntax we use $\wedge$. We define the semantics in terms of a small-step Wright/Felleisen style semantics [21]. We assume that we have a fixed set of actors each of which is identified by a unique actor identification number, *aid* for short. Each actor has a mailbox $M$ and the actor's behavior is specified by an expression $e$. We execute actors in random order. See rule (Schedule) in Figure 8. We simply evaluate the actor expression $k$ number of steps. Evaluation will affect the actors mailbox and has as a side effect the sending of messages. We append sent messages to the appropriate mailboxes.

Evaluation of expressions is described in Figure 7. Rule (Send) yields a don't care expression but has the side effect of sending a message. Side effects are

**Expressions**

| | | |
|---|---|---|
| $e$ | $::= x$ | Variable |
| | $\mid K\ e...e$ | Message |
| | $\mid \lambda x.e \mid e\ e$ | Function and application |
| | $\mid \mathsf{receive}\ [p_i\ \mathsf{when}\ e'_i \rightarrow e_i]_{i \in I}$ | Message receive |
| | $\mid \mathsf{send}\ aid\ e$ | Message send |
| | $\mid ()$ | Don't care |
| $p$ | $::= p' \mid p \wedge p$ | Single-head and multi-head pattern |
| $p'$ | $::= x$ | Variable pattern |
| | $\mid K\ p'...p'$ | Message pattern |

**Actor**

$a ::= (aid, M, e)$
$aid$ Actor identification
$M$ Mailbox
$e$ Behavior

**Fig. 6.** MiniActor Language

collected in a multi-set of constraints. We may send the same message twice to the same actor. The symbol $\uplus$ denotes multi-set union. We do not care much about the order of sent messages which may be random. Evaluation of receiving of messages is parametric in terms of the match semantics described earlier. We first describe the general receive rules in terms of a generic-match reduction $\longrightarrow^*_{X-\mathcal{RP}}$ before we consider the impact of a specific matching policy.

Matching starts in the initial state $\langle M, \emptyset \rangle$ where $M$ is the actor's current mailbox. In rule (Receive) we have found a successful match. From the successful state $\langle M', St, \theta, p_j\ \mathsf{when}\ e'_j \rightarrow e_j \rangle$ we collect the list $St$ of already processed messages which have not been involved in the matching. We put these messages back into the actor's mailbox in their original order (see also rule (Deactivate) in Figure 2). We then continue executing the successful receive body $\theta(e_j)$. There is no rule for covering failure which means that evaluation of a receive clause will block until a successful match is found.

In case we instantiate $\longrightarrow^*_{X-\mathcal{RP}}$ with the first-match reduction relation from Section 3, we obtain a conservative extension of Erlang-style actors with multi-headed message receive patterns. The first-match semantics guarantees the following:

**Monotonicity Property:**

**If** $\langle M, [] \rangle \longrightarrow^*_{First-\mathcal{RP}} \langle M', St, \theta, p_j\ \mathsf{when}\ e'_j \rightarrow e_j \rangle$
**then** $\langle M +\!\!+ M'', [] \rangle \longrightarrow^*_{First-\mathcal{RP}} \langle M' +\!\!+ M'', St, \theta, p_j\ \mathsf{when}\ e'_j \rightarrow e_j \rangle$

The above says that any successful match remains valid if further messages arrive in the actor's mailbox. This is a fairly important property and shows that we can treat the actor's mailbox as a "lazy" structure. That is, the mailbox represents a stream of incoming messages.

We cannot give similar guarantees in case we employ the priority-match semantics from Section 3. Newly arrived messages may invalidate earlier (match)

**Values**

$$v ::= \lambda x.e \mid K\ v_1 ... v_n \mid ()$$

**Send effects**

$$S ::= \emptyset \mid \{\mathsf{send}\ aid\ K\ v_1 ... v_n\} \mid S \uplus S$$

**Evaluation contexts:**

$$E ::= [\,] \mid E\ v \mid K\ E ... E \mid \mathsf{receive}\ [p_i\ \mathsf{when}\ E \to e_i]_{i \in I} \mid \mathsf{send}\ aid\ E$$

**Expression reduction:** $\langle M, e\rangle \xrightarrow{S} \langle M, e\rangle$

(Beta) $\quad \langle M, (\lambda x.e)\ v\rangle \xrightarrow{\emptyset} \langle M, [v/x]e\rangle$

(Send) $\quad \dfrac{S = \{\mathsf{send}\ aid\ K\ v_1 ... v_n\}}{\langle M, \mathsf{send}\ aid\ K\ v_1 ... v_n\rangle \xrightarrow{S} \langle M, ()\rangle}$

(Receive) $\quad \dfrac{\begin{array}{c}\mathcal{RP} = \{p_1\ \mathsf{when}\ e_1' \to e_1, ..., p_n\ \mathsf{when}\ e_n' \to e_n\} \\ \langle M, []\rangle \longrightarrow^*_{X - \mathcal{RP}} \langle M', St, \theta, p_j\ \mathsf{when}\ e_j' \to e_j\rangle \quad \text{for some } j \in \{1, ..., n\} \\ M'' = St +\!\!+ M'\end{array}}{\langle M, \mathsf{receive}\ [p_i\ \mathsf{when}\ e_i' \to e_i]_{i \in \{1,...,n\}}\rangle \xrightarrow{\emptyset} \langle M'', \theta(e_j)\rangle}$

(Context) $\dfrac{\langle M, e\rangle \xrightarrow{S} \langle M', e'\rangle}{\langle M, E[e]\rangle \xrightarrow{S} \langle M', E[e']\rangle}$ (Step) $\dfrac{\langle M, e\rangle \xrightarrow{S} \langle M', e'\rangle}{\langle M, e\rangle \xrightarrow{S}{}^* \langle M', e'\rangle}$

($k$-Step) $\dfrac{\begin{array}{c}\langle M_1, e_1\rangle \xrightarrow{S_1} \langle M_2, e_2\rangle .... \langle M_{k-1}, e_{k-1}\rangle \xrightarrow{S_{k-1}} \langle M_k, e_k\rangle \\ S_k = S_1 \uplus ... \uplus S_{k-1}\end{array}}{\langle M_1, e_1\rangle \xrightarrow{S_k}{}^k \langle M_k, e_k\rangle}$

(Trans) $\dfrac{\langle M_1, e_1\rangle \xrightarrow{S_1}{}^* \langle M_2, e_2\rangle \quad \langle M_2, e_2\rangle \xrightarrow{S_2}{}^* \langle M_3, e_3\rangle}{\langle M_1, e_1\rangle \xrightarrow{S_1 \uplus S_2}{}^* \langle M_3, e_3\rangle}$

**Fig. 7.** Expression Semantics

choices. For instance, consider the priority-match reduction in Figure 5. Suppose that at some later stage the message $A\#5$ arrives (already attached with a unique number). The priority-match reduction in Figure 4 shows that we now may apply a different rule pattern.

**Examples.** We consider some examples to demonstrate the usefulness of multi-headed message patterns. We also consider cases where a priority-match semantics has advantages over a first-match semantics and vice versa.

In our first example, we make use of multi-head messages to find a match between a seller and a (potential) buyer. We use Haskell syntax and receive statements expect the actor as the first argument.

```
matchmaker act = receive act of
                   Sell x, Buy x -> ‘‘match found’’
```

**Actor pool**

$$AP ::= \emptyset \mid \{a\} \mid AP \cup AP$$

**Actor send:** $S@a$ and $S@AP$

$$\emptyset@(aid, M, e) = (aid, M, e)$$

$$\frac{aid \neq aid'}{\{\mathsf{send}\ aid\ K\ v_1 ... v_n\} \uplus S@(aid', M, e) = S@(aid', M, e)}$$

$$\frac{aid = aid' \quad \text{unique number } m}{\{\mathsf{send}\ aid\ K\ v_1 ... v_n\} \uplus S@(aid', M, e) = S@(aid', M +\!\!+ [K\ v_1 ... v_n \# m], e)}$$

$$S@\{a_1, ..., a_n\} = \{S@a_1, ..., S@a_n\}$$

**Actor reduction:** $AP \longrightarrow AP$

$$\text{(Schedule)}\ \frac{\begin{array}{c} AP = \{(aid, M, e)\} \cup AP' \\ \langle M, e\rangle \overset{S}{\longrightarrow}^{k} \langle M', e'\rangle \\ AP'' = \{S@(aid, M', e')\} \cup S@AP' \end{array}}{AP \longrightarrow AP''}$$

$$\text{(Step)}\ \frac{AP \longrightarrow AP'}{AP \longrightarrow^{*} AP'} \quad \text{(Trans)}\ \frac{AP_1 \longrightarrow^{*} AP_2 \quad AP_2 \longrightarrow^{*} AP_3}{AP_1 \longrightarrow^{*} AP_3}$$

**Fig. 8.** MiniActor Semantics

The matchmaker waits for a seller and a buyer to arrive. We employ non-linear patterns to test that the object sold is the same as the one bought. Strictly speaking, we can always replace non-linear patterns by a guard, for example `Sell x, Buy y when x == y -> ‘‘match found’’`. However, non-linear patterns allow for a concise specification of the dependency between the seller and buyer object.

In the above, the receive statement will simply block until a match is found. What we would like is to first check for the presence of a seller and buyer. Then, we check for the case that the seller has no matching buyer. Finally, we catch the case that there is only a buyer. It is tempting to specify the above priorities as follows.

```
matchmaker2 act =
    receive act of
      Sell x, Buy x -> ‘‘match found’’
      Sell x -> ‘‘inform seller no match found’’
      Buy x -> ‘‘inform  buyer ...’’
```

Unfortunately, this is not a valid solution under a first-match semantics. Each incoming seller and buyer will immediately trigger either the second and third clause which overlaps with the first clause. Under a first-match semantics, we must resort to “otherwise” to impose priorities. We have not formally included

"otherwise" in the syntax and formal description in Figure 7 but the development is straightforward.

```
matchmaker3 act =
    receive act of
      Sell x, Buy x -> ``match found''
      otherwise -> receive act
                     Sell x -> ``inform seller no match found''
                     otherwise -> receive act of
                                    Buy x -> ``inform  buyer ...''
                                    otherwise -> matchmaker3 act
```

We first check for a matching seller and buyer. Otherwise, we check for a seller which does not have a matching buyer, followed by checking for a buyer with no matching seller which means that at this stage there are only buyers in the actor's mailbox. If the mailbox is empty we recursively call `matchmaker3`. This is important because else the actor would block indefinitely if a seller arrives. On the other hand, under a priority-match semantics `matchmaker2` is a valid solution.

In summary, there is no clear winner. The first-match and priority-match semantics have their pros and cons. Under a priority-match semantics we can read off the priorities directly from the receive clauses. Under a first-match semantics, we need to explicitly program priorities via "otherwise" and nested receive statements. But the first-match semantics works better in a setting where we process a continuous stream of messages because it enjoys a monotonicity property. We believe that both semantics represent interesting, alternative design choices for an actor language and it will depend on the application which semantics is the better choice.

## 5 Implementation

We have implemented a prototype as a library extension in Haskell using the Glasgow Haskell Compiler [6]. GHC supports light-weight threads. Therefore, our implementation scales well to many actors. The latest version including examples can be downloaded via [9].

We briefly highlight the main features of our implementation. We support strongly typed actors in the sense that an actor's mailbox can only holds messages of a certain (data) type. The actual mailbox consists of two parts. A buffer $B$ for recently sent messages represented as a transacted channel to manage conflicts among multiple writers. A transacted channel is a linked list in shared memory where access is protected by Software Transactional Memory. In the future we plan to support distributed channels to support sending of messages across the network. The second part of the mailbox is a linked list $L$ to process $\langle M, St\rangle$ by a single reader. We use pointers to indicate the start of $M$ and $St$.

Our implementation applies the first-match scheme outlined in Section 3. We process messages in $M$ in sequential order. If $M$ is empty we check the buffer and transfer any recently sent messages to $M$. If the buffer is empty, we wait for new messages to arrive. To improve the search for matching messages, we adopt common CHR optimizations [17, 10]. Our current version does not support all

of the CHR optimizations but we plan to gradually integrate them into later versions.

One optimization is message indexing. For example, consider the earlier rule pattern `Sell x,Buy x`. Suppose the active message `Sell SomeObject` in $M$ matches part of the rule head. We yet need to find the matching partner `Buy SomeObject` in $St$. We use (hash)-indexing for a fast lookup of `Buy SomeObject` (if exists). We build hash-indexes whenever we deactivate a constraint.

Another optimization is the early scheduling of guards. For example, consider the rule pattern `Foo x,Bar y,Erk z when x > y`. Suppose that `Foo 1` is our active message and what remains is to find matching partners `Bar y` and `Erk z` for some `y` and `z` such that `x>y`. As soon as we have found a possible candidate `Bar y` we should schedule (i.e. test) the guard `1>y` to reduce the search space.

## 6 Related Work

In their foundational work, Kahn and Saraswat [12] establish connections between the actor programming model and concurrent constraint logic programming. Our work follows their footstep by providing a formal model for multi-headed message receive patterns with guards based on CHR style multi-set constraint matching semantics.

There are a number of works which integrate Erlang style actors into object-oriented and functional languages. For examples, consider [11, 20, 8]. These works focus mainly on language integration and implementation aspects of Erlang style actors whereas we enrich the Erlang actor model with multi-headed message receive patterns and also explore several alternative matching semantics.

Closest to our work is some recent work by Haller and Van Cutsem [7]. Their main focus of attention is the implementation of join patterns [4] by means of extensible pattern matching. There are close connections between the join and actor model and their system has support for join-style actors (what we call multi-headed message receive patterns). However, their approach appears to be more limiting. They can only support a limited form of guards, it is unclear whether they can support our non-linear patterns at all. Furthermore, they do not give a semantic specification of their system.

## 7 Conclusion

We have studied an extension of Erlang style multi-headed message receive patterns with guards. Such an extension is useful as supported by a number of examples. We have explored two possible semantics by adapting previously studied CHR multi-set matching semantics. The first-match semantics gives us a conservative extension of Erlang style actors to the setting of multi-headed message receive patterns with guards. We have also explored a priority-match semantics. For certain applications (where priority constraints need to be enforced) this semantics is the better choice. Both semantics can be implemented efficiently as shown by previous work [3, 17, 14]. Our library-based prototype exploits some of these methods. We plan to integrate further optimizations and conduct more experimentations in future work.

In another line of work, we plan to enrich join patterns with CHR style guards and non-linear patterns. In the join context, patterns can be be executed

concurrently. We wish to parallelize the concurrent execution of join patterns. Contrast this to actor receive patterns which are executed sequentially (either following the first-match or priority-match semantics). We have already started some work how to parallelize CHR [15] and explored an CHR style enriched join pattern language [18]. We plan to report more detailed results in the future.

## References


1. G. Agha. *ACTORS: A Model of Concurrent Computation in Distributed Systems*. MIT Press, 1986.
2. J. Armstrong, R. Virding, C. Wikström, and M. Williams. *Concurrent Programming in Erlang, Second Edition*. Prentice-Hall, 1996.
3. G. J. Duck, P. J. Stuckey, M. J. García de la Banda, and C. Holzbaur. The refined operational semantics of Constraint Handling Rules. In *Proc. of ICLP'04*, volume 3132 of *LNCS*, pages 90–104. Springer-Verlag, 2004.
4. C. Fournet and G. Gonthier. The join calculus: A language for distributed mobile programming. In *Applied Semantics, International Summer School, APPSEM 2000*, pages 268–332, Caminha, Portugal, 2002. Springer-Verlag.
5. Thom Frühwirth. Theory and practice of Constraint Handling Rules. *J. Logic Programming, Special Issue on Constraint Logic Programming*, 37(1–3):95–138, 1998.
6. Glasgow haskell compiler home page. `http://www.haskell.org/ghc/`.
7. P. Haller and T. Van Cutsem. Implementing Joins using Extensible Pattern Matching. Technical report, EPFL, 2007. To appear in Proc. of DAMP'08.
8. P. Haller and M. Odersky. Actors that unify threads and events. In *Proc. of COORDINATION'07*, volume 4467 of *LNCS*, pages 171–190. Springer-Verlang, 2007.
9. HaskellActor. `http://www.comp.nus.edu.sg/~sulzmann`.
10. C. Holzbaur, M. J. García de la Banda, P. J. Stuckey, and G. J. Duck. Optimizing compilation of Constraint Handling Rules in HAL. *TPLP*, 5(4-5):503–531, 2005.
11. F. Huch. Erlang-style distributed Haskell, 1999. In Draft Proc. of IFL'99.
12. K. Kahn and Vijay A. Saraswat. Actors as a special case of concurrent constraint (logic) programming. In *Proc. of OOPSLA/ECOOP*, pages 57–66. ACM, 1990.
13. L. De Koninck, T. Schrijvers, and B. Demoen. User-definable rule priorities for CHR. In *Proc. of PPDP'07*, pages 25–36. ACM, 2007.
14. L. De Koninck, P.J. Stuckey, and G.J. Duck. Optimizing compilation of CHR with rule priorities. In *To appear in Proc. of FLOPS'08*, 2008.
15. E. S. L. Lam and M. Sulzmann. A concurrent Constraint Handling Rules implementation in Haskell with software transactional memory. In *Proc. of DAMP'07*, pages 19–24. ACM Press, 2007.
16. S. Peyton Jones, editor. *Haskell 98 Language and Libraries: The Revised Report*. Cambridge University Press, 2003.
17. Tom Schrijvers. *Analyses, Optimizations and Extensions of Constraint Handling Rules*. PhD thesis, K.U.Leuven, Leuven, Belgium, June 2005.
18. M. Sulzmann and E. S. L. Lam. Haskell – Join – Rules. In Draft Proc. of IFL'07, September 2007.
19. J. A. Trono. A new exercise in concurrency. *SIGCSE Bull.*, 26(3):8–10, 1994.
20. C. Varela and G. Agha. Programming dynamically reconfigurable open systems with SALSA. *SIGPLAN Not.*, 36(12):20–34, 2001.
21. Andrew K. Wright and Matthias Felleisen. A syntactic approach to type soundness. *Information and Computation*, 115(1):38–94, 1994.