A Unified Semantics for Future Erlang

Hans Svensson
Department of Computer Science and Engineering,
Chalmers University of Technology,
Göteborg, Sweden
hanssv@chalmers.se

Lars-Åke Fredlund Clara Benac Earle
Facultad de Informática,
Universidad Politécnica de Madrid,
Spain
{fred,cbenac}@babel.ls.fi.upm.es

Abstract

The formal semantics of Erlang is a bit too complicated to be easily understandable. Much of this complication stems from the desire to accurately model the current implementations (Erlang/OTP R11-R14), which include features (and optimizations) developed during more than two decades. The result is a two-tier semantics where systems, and in particular messages, behave differently in a local and a distributed setting. With the introduction of multi-core hardware, multiple run-queues and efficient SMP support, the boundary between local and distributed is diffuse and should ultimately be removed. In this paper we develop a new, much cleaner semantics, for such future implementations of Erlang. We hope that this paper can stimulate some much needed debate regarding a number of poorly understood features of current and future implementations of Erlang.

Categories and Subject Descriptors D.3.1 [*Formal Definitions and Theory*]: Semantics

General Terms Languages, Verification

Keywords Erlang, Semantics

1. Introduction

In the past years quite a lot of effort has been spent on describing and defining the semantics of Erlang; from the early work of Petterson [5], via the formal definition of single node semantics by Fredlund [3] and the more high-level language philosophy description by Armstrong [1], to the definition of the distributed Erlang semantics by Claessen and Svensson [2] later refined by Svensson and Fredlund [6]. The end result is an accurate formal semantics of Erlang, that in detail describe how the current (Erlang/OTP R11-R14) implementation behave. The successful implementation of McErlang (by Fredlund and Svensson [4]) is further evidence that the semantics is actually useful and can be used for practical purposes.

Unfortunately, the formal semantics is a bit too complicated to be easily understandable. This is a bit of a nuisance, since the language design philosophy [1] is very clean and easy to grasp. When analyzing the current Erlang semantics we could see that much of the complexity stems from the strive to very closely follow the current implementation of the Erlang run-time system (ERTS). The result is a two-tier semantics where systems, and in particular messages, behave differently in a local and a distributed setting. Also, the Erlang language has evolved over the years, leaving some legacy constructions that could have been avoided if the language was re-constructed today. The most striking example is the overlap between *monitors* and *links*, where there is very little practical difference between a *trapped* link-message and a monitor-message.

With the current trend being the introduction of multi-core hardware, and the latest advances in the ERTS with multiple run queues and efficient SMP support; the boundary between local and distributed is diffuse and should ultimately be removed. In this paper we propose a new, much cleaner semantics for a future implementation of Erlang, where there is no boundary between local and remote processes and where the semantics does not in itself hinder parallelization.

It is not our intent to define the future semantics of Erlang, but we hope to initiate, and stimulate, the discussion and debate of this topic. We believe that it is vital to lift some of the current restrictions in order for Erlang to continue to scale to upcoming multi-core architectures. We are also well aware of all the complications involved in (drastically) changing the semantics of a mature programming language. The vast amount of legacy code, and backwards compatibility are large hurdles to overcome. Nevertheless, the earlier we start thinking about these improvements, the more time we have to make them before reality catches up!

Paper Organization The paper is organized as follows. In Sect. 2 we begin by giving a high-level intuitive overview of the new semantics, before we formally state definitions and rules of the new semantics in Sect. 3. In Sect. 4 we describe the inner workings of the node controller. Thereafter, in Sect. 5, we restrict the possible execution sequences by stating a rule for fairness. Thereafter, in Sect. 6 we make a few illustrative comparisons between the new and the current semantics. Finally, we conclude and add some future directions in Sect. 7.

2. Intuitive Semantics

Before we start listing definitions and semantic rules, we provide an informal but hopefully intuitive high-level overview of the new semantics.

In our setting, a complete distributed system consists of a number of *nodes*. The nodes are the top-level containers in such a system. A node contains a *node controller* (an addition in this new semantics), and a number of *processes*. One should note that a node is not equivalent to a physical machine, rather the opposite, a single machine can host many nodes. This high-level description of a distributed system is illustrated in Figure 1.

Erlang'10, September 30, 2010, Baltimore, Maryland, USA.

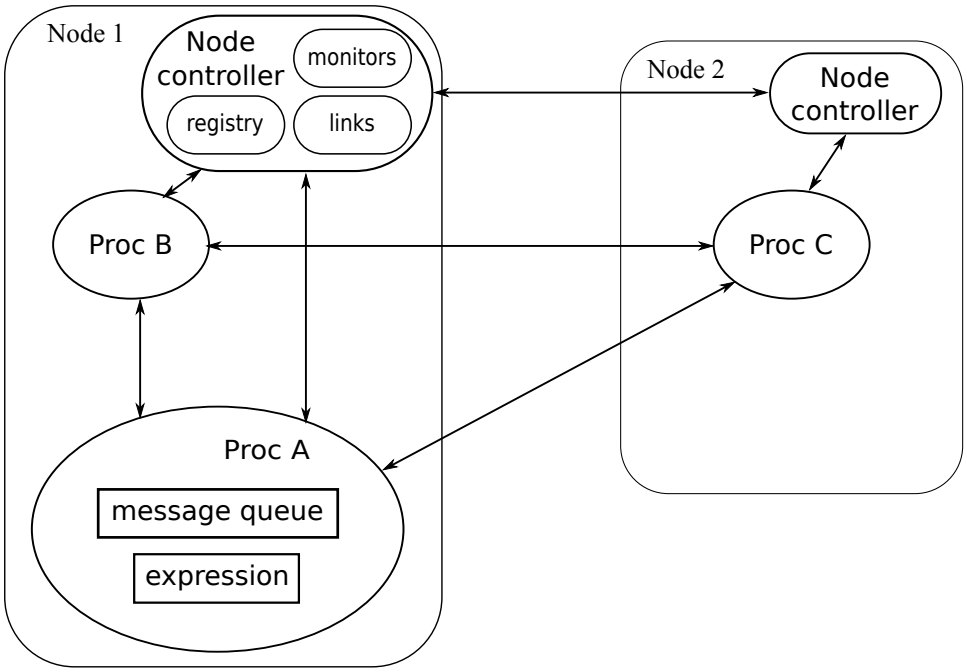

Figure 1. Distributed system - Nodes, node controllers, and processes

2.1 Everything is Distributed

As mentioned in the introduction, with the introduction of multi-core architectures, the boundary between local and remote is more diffuse. Therefore, the new semantics treats all messages (including messages sent to one-self!) equally, and all messages are sent through a (virtual) system message queue (the *ether*). In practice this means that every message-passing consists of two steps, sending and delivering. Thus, messages sent between different pairs of processes can be freely re-ordered.

We have also chosen to make (almost) all *side-effecting* actions (such as spawning a new process, linking to a process, etc.) asynchronous. To stress that many side effects have a similar impact on the system, they are treated in a uniform way by the introduction of a *node controller*. The node controller is responsible for all node-local administration.

As an example of a side effect, consider registration of a name for a process. This is done by sending a signal to the node controller. Some time later the node controller receives the signal, decides whether the name can be registered and sends a reply to the process doing the registration. This might seem a bit impractical from a user perspective, but nothing stops us from defining a higher-level function that sends the register-signal **and** waits for the reply.

2.2 Uni-directional Links Only

In the new semantics we provide both the concept of *links* and *monitors*. However, we do not have any functionality like *trapping exits*[1] that exists in the current semantics. This means that if a linked process terminates, the linking process is also terminated, no 'but's and 'if's. (The trap exit functionality is in practice the same as a monitor.) We have also opted for making links and monitors uni-directional. That is if process A is linking to process B, a failure of A does not affect B.

As a practical example let us consider how to re-implement the supervisor behavior using uni-directional links and monitors. Assuming that we want to supervise a child process, specified as the tuple `{M,F,A}`, so that when the child terminates the supervisor is informed, and if the supervisor terminates abnormally, the child terminates too, then the following supervisor code fragment suffices:

```
SupervisorPid = self(),
ChildPid =
  spawn(fun () ->
          link(SupervisorPid), apply(M,F,A)
        end),
MonitorRef = monitor(process, ChildPid),
```

2.3 A Built-in Registry

We have decided to make an addition to the semantics by including a process registry in the semantics. This inclusion is questionable, but we think that the ability to communicate with named processes is such a central concept in Erlang that it deserves a place in the semantics.

As in current Erlang, the basic operations supported are sending a message to a named process (`atom!msg`), sending a message to a named process on a remote node (`{atom,node}!msg`), and registering, unregistering and name lookup. However, for uniformity, in this semantics names can be registered for remote processes (i.e., `register(name,pid)` does not fail if `pid` is a remote process), and registering a local process at a remote node is supported too (using the operation `register(node,name,pid)`). As a consequence, when a message is sent to a remote node using the syntax `{atom,node}!msg` there is no guarantee that the process that should receive the message is located at `node`; thus it may be necessary to relay the message to a process on yet another node.

[1] In the current semantics, processes are able to trap exit-signals, and treat them as ordinary (information-)messages, thus avoiding termination upon receiving an exit-signal.

2.4 Message-Passing Guarantees

There are few message-passing guarantees in general in the new semantics, but for each pair of processes the order of messages is guaranteed. That is, if process A sends a stream of messages $M_1, M_2, M_3, \cdots$ to process B, then process B will receive the messages in exactly that order. (With the possibility of dropping messages at the end of the sequence because of a node disconnect.)

It should be noted that this guarantee matches exactly what is outlined by Armstrong [1] in his thesis. However note that as observed in Svensson and Fredlund [7], current Erlang implementations does not provide this guarantee, as when distributed processes communicate, messages may be lost.

Further note that in the semantics there are no guarantees regarding the ordering of messages delivered to a process if that process is addressed both directly (using its pid) and indirectly through a registered name. To exemplify we assume that a process P executes the code fragment `Q!msg1, q_name!msg` where `Q` is bound to the pid of a process Q also registered under the name `q_name`. In such a scenario there are no guarantees provided by the semantics regarding whether `msg1` or `msg2` is delivered first to the mailbox of Q.

3. Formal Semantics

In this section we present the semantics in a style similar to earlier Erlang formal semantics. We use this style since it is straightforward and easy to follow, while still being detailed enough to allow precise arguments about correctness. We make the necessary definitions before presenting the semantic rules.

Definition 1 A *process*, ranged over by $p \in \mathsf{Process}$, is a triplet: $\mathsf{Expression} \times \mathsf{ProcessIdentifier} \times \mathsf{MessageQueue}$, written $\langle e, pid, q\rangle$ such that

- e is an expression currently run by the process,
- pid is the process identifier of the process,
- q is a message queue.

The expression (e) in a process should be interpreted as a normal (Erlang) expression similar to the expressions defined in [3].

Definition 2 A *process group*, ranged over by $pg \in \mathsf{ProcessGroup}$, is either an empty process group $\emptyset$, a single process, or a combination of process groups pg_1 and pg_2, written as $pg_1 \parallel_P pg_2$.

Definition 3 A *node controller*, ranged over by $nc \in \mathsf{NodeController}$, is a triplet: $\mathcal{P}(\mathsf{ProcessIdentifier} \times \mathsf{ProcessIdentifier}) \times \mathcal{P}(\mathsf{MonitorReference} \times \mathsf{ProcessIdentifier} \times \mathsf{ProcessIdentifier}) \times \mathcal{P}(\mathsf{ProcessIdentifier} \times \mathsf{ProcessName})$, written $\langle lnks, mns, reg\rangle$ such that

- $lnks$ is a set of links, i.e. tuples ($link\_from$, $link\_to$),
- mns is a list of monitors, i.e. tuples (mon_name, mon_from, mon_to),
- reg is a set of registered names, i.e. tuples ($name$, pid).

Definition 4 A *System message queue*, also named an *ether*, ranged over by $eth \in \mathsf{SystemMessageQueue}$, consists of a finite sequence of triplets $\mathsf{Identifier} \times \mathsf{Identifier} \times \mathsf{Signal}$. Let ϵ denote the empty sequence, $(\cdot)$ is concatenation and $(\setminus)$ is deletion of the first matching triplet, e.g.:

$$\begin{aligned} eth &= (a_2, b_1, c_1)\cdot(a_1, b_2, c_1)\cdot(a_1, b_2, c_2)\cdot(a_1, b_2, c_1) \setminus (a_1, b_2, c_1) \\ &= (a_2, b_1, c_1)\cdot(a_1, b_2, c_2)\cdot(a_1, b_2, c_1) \end{aligned}$$

Definition 5 A *node*, ranged over by $n \in \mathsf{Node}$, is a triple: $\mathsf{ProcessGroup} \times \mathsf{NodeIdentifier} \times \mathsf{NodeController}$, it is written $[pg, nid, nc]$ such that

- pg is a group processes running at the node,
- nid is a unique identifier for the node,
- nc is a node controller.

Let $\mathsf{Identifier}$ be the union of process identifiers ($\mathsf{ProcessIdentifier}$) and node identifiers ($\mathsf{NodeIdentifier}$), and let identifiers be ranged over by $id \in \mathsf{Identifier}$.

Definition 6 A *node system*, ranged over by $ns \in \mathcal{P}\ \mathsf{Node}$, is either an empty node system $\emptyset$, a single node, or a combination of node systems ns_1 and ns_2, written as $ns_1 \parallel_N ns_2$.

Definition 7 A *system*, ranged over by $s \in \mathsf{System}$, is a tuple: $\mathcal{P}\ \mathsf{Node} \times \mathsf{SystemMessageQueue}$, written $[\![ns, eth]\!]$ such that

- ns is a node system, and
- eth is an ether (system message queue).

Intuitively, the composition of processes into process groups and nodes into node systems should be thought of as a set of processes (nodes). We will take care to define the semantics in such a way as to ensure that the operators $\parallel_P$ and $\parallel_N$ are commutative and associative.

To keep the semantic rules reasonably short and readable we use some supportive functions to abbreviate some lengthy (and often repeated) constructions.

Definition 8 Let the function $\mathsf{isNid}(i)$ (where $i \in \mathsf{Identifier}$) return true if the identifier i represents a node identifier, and false if it represents a process identifier.

Definition 9 Let the function $\mathsf{node}(p)$ (where $p \in \mathsf{ProcessIdentifier}$) return the node identifier for a given process identifier.

Definition 10 Let the function $\mathsf{destNid}(sig)$ return the node identifier of the remote node involved in the signal sig (note that the return value is `undefined` for some signals), e.g.

$$\begin{aligned} \mathsf{destNid}(\mathbf{link}(pid)) &\longrightarrow \mathsf{node}(pid), \text{ and} \\ \mathsf{destNid}(\mathbf{spawn}(e, ref)) &\longrightarrow \texttt{undefined} \end{aligned}$$

Definition 11 $\mathsf{ethMatch}(eth, to, from)$, is a function that given a system message queue, a sender identity ($from$), and a receiver identity (to) returns the first message in the queue sent by $from$ to to, e.g.

$$\begin{aligned} eth \quad &= \quad (a_2, b_1, c_1)\cdot(a_1, b_2, c_1)\cdot(a_1, b_2, c_2)\cdot(a_1, b_2, c_1) \\ &\Longrightarrow \quad \mathsf{ethMatch}(eth, a_1, b_2) = c_1 \end{aligned}$$

Definition 12 Let the functions $\mathsf{pids}(pg)$ and $\mathsf{nids}(ns)$ return the set of process identifiers belonging to processes in the process group pg and the set of node identifiers in the node system ns respectively. Further let a process group and a system be *well-formed* if its process identifiers and node identifiers are unique, i.e., a process identifier belongs to at most one process and a node identifier to at most one node, and an identifier is either a node identifier or a process identifier.

In the following we assume that all (Erlang) node systems are well-formed and that they only contain well-formed process groups.

In the semantics, *signals* are items of information transmitted between a sending and a receiving process (or node controller). A *process action*, committed by a process, group of processes, or node, is either a silent action, an input action, an output action, a node termination, or a node disconnect.

Definition 13 (Process signals) The process signals, ranged over by $sig \in \mathsf{Signal}$ are:

sig ::=	$\mathbf{message}(v)$	message
|	$\mathbf{link}(pid)$	linking with process
|	$\mathbf{unlink}(pid)$	unlinking process
|	$\mathbf{monitor}(pid, ref)$	monitor process
|	$\mathbf{unmonitor}(ref)$	unmonitor process
|	$\mathbf{monitor\_node}(nid)$	monitor node
|	$\mathbf{unmonitor\_node}(ref)$	unmonitor node
|	$\mathbf{whereis}(name)$	lookup pid for name
|	$\mathbf{register}(name, pid)$	register name for pid
|	$\mathbf{spawn}(\mathrm{e}, ref)$	spawn a process
|	$\mathbf{spawn\_node}()$	spawn a node
|	$\mathbf{nsend}(name, v)$	named send
|	$\mathbf{exit}(v)$	external termination signal
|	$\mathbf{died}(id, v)$	termination signal

Definition 14 (Process actions) The process actions, ranged over by $\alpha \in \mathsf{Action}$ are:

α	::=	τ	silent action
	|	$pid\ !_{from}\ sig$	output action
	|	$pid\ ?_{from}\ sig$	input action
	|	$\mathbf{die}(nid)$	node termination
	|	$\mathbf{disconnect}(nid_1, nid_2)$	node disconnect

In the following we define formally the possible computation steps of an Erlang system. In this definition we assume the existence of a set of transition rules for expressions, a subset of $\mathcal{P}(\mathsf{Expression} \times \mathsf{exprAction} \times \mathsf{Expression})$, which can be found in [3]. For completeness we repeat below the definition of the expression actions:

Definition 15 (Expression actions) The expression actions, ranged over by $\alpha \in \mathsf{exprAction}$, are:

α	::=	τ	computation step
	|	$pid\ !\ v$	output
	|	$\mathbf{exiting}(v)$	exception
	|	$\mathbf{read}(q, v)$	read from queue
	|	$\mathbf{test}(q)$	checking queue contents
	|	$f(v_1, \ldots, v_n) \rightsquigarrow v$	built-in function call

Intuitively, $f(v_1, \ldots, v_n) \rightsquigarrow v$ corresponds to the call of a built-in function f which returns the value v.

Definition 16 Let the function $\mathsf{mkAction}(msgs)$ be defined as follows:

$$\mathsf{mkAction}(\epsilon) \longrightarrow \tau$$
$$\mathsf{mkAction}((to, from, sig)\cdot msgs) \longrightarrow to\ !_{from}\ sig\ ;\mathsf{mkAction}(msgs)$$

Definition 17 The system transition relation is the least relation satisfying the transition rules in Tables 1 - 9

It should be pointed out that some rules should be combined together to achieve the desired effect. The most obvious examples are perhaps the output-rules, where the message is sent in the *side_eff*-rule in Table 2 and delivered in the *output*-rule in Table 5.

Most of the semantic rules are self explaining, but we go through each table below and point out some subtleties and explain the more complicated rules.

Expression evaluation in process context Table 1 contains expression evaluation that is local to a process. Note that all these actions take place in a larger context (a system), but since the rules do not depend on anything outside the process, the context is not visible in the rules. (The rules are lifted to the system level by the *internal*-rule defined in Table 5.) In the rule *silent*, if the expression e has a transition $e \xrightarrow{\tau} e'$ (a normal computation step) then the process (and in the greater scheme the whole system) $\langle e, pid, q\rangle$ has a transition labelled by the silent action τ to the process $\langle e', pid, q\rangle$.

$$\textit{silent}\ \frac{e \xrightarrow{\tau} e'}{\langle e, pid, q\rangle \xrightarrow{\tau} \langle e', pid, q\rangle}$$

$$\textit{read}\ \frac{e \xrightarrow{\mathbf{read}(q_1, v)} e'}{\langle e, pid, q_1 \cdot v \cdot q_2\rangle \xrightarrow{\tau} \langle e', pid, q_1 \cdot q_2\rangle}$$

$$\textit{test}\ \frac{e \xrightarrow{\mathbf{test}(q)} e'}{\langle e, pid, q\rangle \xrightarrow{\tau} \langle e', pid, q\rangle}$$

$$\textit{self}\ \frac{e \xrightarrow{\mathtt{self}() \rightsquigarrow pid} e'}{\langle e, pid, q\rangle \xrightarrow{\tau} \langle e', pid, q\rangle}$$

Table 1. Rules for process-local expression evaluation

$$\textit{side\_eff}\ \frac{e \xrightarrow{side\_eff\,(args) \rightsquigarrow res} e' \qquad (res, id, sig) = \mathsf{mkSig}(\mathsf{node}(pid), side\_eff, args)}{\langle e, pid, q\rangle \xrightarrow{id\ !_{pid}\ sig} \langle e', pid, q\rangle}$$

Table 2. Rules for side effecting expression evaluation

$$\begin{array}{ll}
\mathsf{mkSig}(nid, !, [pid, v]) & \longrightarrow (v, pid, \mathbf{message}(v)) \\
\mathsf{mkSig}(nid, !, [\{name, nid'\}, v]) & \longrightarrow (v, nid', \mathbf{nsend}(name, v)) \\
\mathsf{mkSig}(nid, !, [name, v]) & \longrightarrow (v, nid, \mathbf{nsend}(name, v)) \\
\mathsf{mkSig}(nid, \mathtt{exit}, [pid, v]) & \longrightarrow (\mathtt{ok}, pid, \mathbf{exit}(v)) \\
\mathsf{mkSig}(nid, \mathtt{link}, [pid]) & \longrightarrow (\mathtt{ok}, nid, \mathbf{link}(pid)) \\
\mathsf{mkSig}(nid, \mathtt{unlink}, [pid]) & \longrightarrow (\mathtt{ok}, nid, \mathbf{unlink}(pid)) \\
\mathsf{mkSig}(nid, \mathtt{spawn}, [e, ref]) & \longrightarrow (ref, nid, \mathbf{spawn}(e, ref)) \\
\mathsf{mkSig}(nid, \mathtt{spawn}, [nid', e, ref]) & \longrightarrow (ref, nid', \mathbf{spawn}(e, ref))
\end{array}$$

Table 3. Definition of mkSig()

In the *read*-rule, a transition from $\langle e, pid, q_1 \cdot v \cdot q_2\rangle$ to the target process $\langle e', pid, q_1 \cdot q_2\rangle$ is enabled whenever the process mailbox (queue) can be split into three parts $q_1 \cdot v \cdot q_2$, and the expression transition *receive*: $e \xrightarrow{\mathbf{read}(q_1, v)} e'$ is derivable at the expression level. (See [3] for the exact definition of the *receive*-rule.) Thus, the rules *read* and *receive* together ensure the intuitive semantics of the `receive` construct. The side effecting actions are handled by the rule in Table 2.

Side-effecting (node controller) expression evaluation Table 2 contains the rule for side effecting actions. Many of these actions require involvement of the node controller. The *side_eff* rule is generic, and include the actions: *send, exit, link, unlink, monitor, unmonitor, spawn, register, whereis, monitor_node, and unmonitor_node*. All rules in the table are *asynchronous*, e.g. `whereis`() returns immediately with `ok`, whereas the "real" result (for example for *whereis* and *unlink*) arrive later (as a normal receivable signal). The asynchronous nature of these operations results in deceptively simple semantic rules. In Table 7 the node controller side of these rules is presented, and in Sect. 4 the more complex internals of the node controller is explained. The *side_eff*-rule uses the translation

$$\textit{node}\ \frac{e \xrightarrow{\texttt{node}()\rightsquigarrow nid} e'}{[\![[\langle e, pid, q\rangle \|_P pg, nid, nc] \|_N ns, eth]\!] \xrightarrow{\tau} [\![[\langle e', pid, q\rangle \|_P pg, nid, nc] \|_N ns, eth]\!]}$$

$$\textit{termination}\ \frac{sig = \mathbf{died}(pid, \texttt{normal})}{[\![[\langle v, pid, q\rangle \|_P pg, nid, nc] \|_N ns, eth]\!] \xrightarrow{nid\ !_{pid}\ sig} [\![[pg, nid, nc] \|_N ns, eth \cdot (nid, pid, sig)]\!]}$$

$$\textit{exiting}\ \frac{e \xrightarrow{\mathbf{exiting}(v)} e' \qquad sig = \mathbf{died}(pid, v)}{[\![[\langle e, pid, q\rangle \|_P pg, nid, nc] \|_N ns, eth]\!] \xrightarrow{nid\ !_{pid}\ sig} [\![[pg, nid, nc] \|_N ns, eth \cdot (nid, pid, sig)]\!]}$$

$$\textit{spawn}_{node}\ \frac{e \xrightarrow{\texttt{spawn\_node}()\rightsquigarrow nid'} e' \qquad nid' = \mathsf{fresh}()}{[\![[\langle e, pid, q\rangle \|_P pg, nid, nc] ns, eth]\!] \xrightarrow{\tau} [\![[\langle e', pid, q\rangle \|_P pg, nid, nc] \|_N [\emptyset, nid', \langle\emptyset, \emptyset, \emptyset\rangle] \|_N ns, eth]\!]}$$

Table 4. Process rules for expression evaluation at system level

function mkSig to construct an appropriate *signal*, this function is defined in Table 3.

Expression evaluation in node context Table 4 contains the simple rule for evaluation of node() in the system context. The function returns immediately (i.e. it is not asynchronous). Because the result is depending on the node context, the rule is separated from the rules in Table 1.

The *termination*- and the *exiting*-rule are also evaluated at the system level. Terminated processes are removed from the system as seen in Table 4.

The last rule in Table 4 handle the slightly odd action of spawning a new *node*. This action is an odd bird in the semantics, and the obvious place for such a rule would be in the general *side_eff*-rule. However, since the result is the *creation* of a whole node it does not fit into the general pattern. Moreover, making the pattern even more general was not too appealing, instead the rule is listed on its own here. The rule makes sure that a new node is created, with a fresh node identifier.

Node level input- and output-rules Table 5 and Table 6 contains input and output rules. The *output*-rule deliver messages to the system message queue (the *ether*), while the *internal*-rule simply lifts non-output expression evaluations to the system level (as discussed above).

For the *input*-rules we should note that the rules can be applied in an arbitrary order for pairs of a sender and a receiver. This means that messages (from different senders and receivers) can possibly be reordered. However, at the same time this introduces a problem, namely that a certain (sender,receiver)-pair is never considered. That means that the delivery of some messages could potentially be delayed forever. Many properties can not be proved for such a non-fair situation, to deal with this problem we have to state a fairness rule (in Sect. 5). Also, since nothing stops processes from sending messages to dead processes, we need rules (the *missing*-rules) to eventually remove such messages. The $\textit{missing}_{node}$-rule is also responsible for sending some node controller reply-messages, such as link-replies (noproc) and spawn-replies (a useless pid), by using the ncEffect()-function. The ncEffect()-function is defined in Sect. 4.

The *exit*-rule handles external abnormal termination of processes, the reason for termination could be a triggered link or an explicit call to exit(). The receiving process is terminated and the node controller is (eventually) informed.

The presented message-passing mechanism is totally asynchronous, even messages sent to oneself are delivered through the system message queue. Finally, note that messages to node controllers are dealt with in Table 7.

Node controller (meta-)rules Table 7 contains meta-rules for the *node controller*. The rules describe how node controller signals are treated. There are two cases, in the normal case (the *nc*-rule) the signal is at its final destination and needs only to be handled at this node controller. In the second case (the $\textit{nc}_{forward}$-rule) we handle a signal involving a remote pid at the sender side; i.e. the signal should be passed on after doing local actions. Specifying these actions and replies is the meat of these rules. The actions (i.e. the definitions of the function ncEffect) are presented in Sect. 4. To distinguish local signals from remote signals the function destNid() is used, it is defined in Table 8.

It should be noted that the node controller does not have a separate message queue. Signals sent to the node controller are consumed immediately upon delivery. Thus, there is no selective receive for node controllers. The *nc*-rule and the $\textit{nc}_{forward}$-rule are also special in the sense that they (potentially) have more than one action attached to their transitions, both an input action and output action(s). The interpretation of multiple actions for one transition is straightforward; the actions are ordered, just as if there had been several consecutive transitions with a single action. The function mkAction() is used to create multiple actions from a sequence of messages.

$$\begin{array}{ll}
\mathsf{destNid}(\mathbf{link}(pid)) & \longrightarrow \mathsf{node}(pid) \\
\mathsf{destNid}(\mathbf{unlink}(pid)) & \longrightarrow \mathsf{node}(pid) \\
\mathsf{destNid}(\mathbf{spawn}(e, ref)) & \longrightarrow \texttt{undefined} \\
\mathsf{destNid}(\mathbf{nsend}(name, v)) & \longrightarrow \texttt{undefined}
\end{array}$$

Table 8. Definition of destNid()

Node failure rules Table 9 contains rules for failing nodes, either when a single node crash or when two nodes are disconnected. In both cases the node controllers are informed. Also here we see how deferring most of the work to the node controller saves us from complex semantic rules. The corresponding rules in the distributed Erlang semantics fills the majority of a page. Nevertheless, the complexity does not fully disappear, we still have to deal with some bookkeeping inside the node controller.

$$\textit{internal}\ \frac{p \xrightarrow{\tau} p'}{[\![[p \|_P pg, nid, nc] \|_N ns, eth]\!] \xrightarrow{\tau} [\![[p' \|_P pg, nid, nc] \|_N ns, eth]\!]}$$

$$\textit{output}\ \frac{p \xrightarrow{to\ !_{from}\ sig} p'}{[\![[p \|_P pg, nid, nc] \|_N ns, eth]\!] \xrightarrow{to\ !_{from}\ sig} [\![[p' \|_P pg, nid, nc] \|_N ns, eth \cdot (to, from, sig)]\!]}$$

Table 5. System output rules

$$\textit{input}\ \frac{\mathsf{ethMatch}(eth, to, from) = sig = \mathbf{message}(v)}{[\![[\langle e, to, q\rangle \|_P pg, nid, nc] \|_N ns, eth]\!] \xrightarrow{to\ ?_{from}\ sig} [\![[\langle e, to, q \cdot sig\rangle \|_P pg, nid, nc] \|_N ns, eth \setminus (to, from, sig)]\!]}$$

$$\textit{exit}\ \frac{\mathsf{ethMatch}(eth, pid, from) = \mathbf{exit}(v)}{[\![[\langle e, pid, q\rangle \|_P pg, nid, nc] \|_N ns, eth]\!] \xrightarrow{pid\ ?_{from}\ \mathbf{exit}(v)\ ;\ nid\ !_{pid}\ \mathbf{died}(pid,v)} [\![[pg, nid, nc] \|_N ns, eth \setminus (pid, from, \mathbf{exit}(v)) \cdot (nid, pid, \mathbf{died}(pid, v))]\!]}$$

$$\textit{missing}_{process}\ \frac{\mathsf{ethMatch}(eth, to, from) = sig \qquad \neg\mathsf{isNid}(to) \qquad (\mathsf{node}(to) = nid\ \wedge\ to \notin \mathsf{pids}(pg))\ \vee\ \mathsf{node}(to) \notin \mathsf{nids}(ns)}{[\![[pg, nid, nc] \|_N ns, eth]\!] \xrightarrow{to\ ?_{from}\ sig} [\![[pg, nid, nc] \|_N ns, eth \setminus (to, from, sig)]\!]}$$

$$\textit{missing}_{node}\ \frac{\mathsf{ethMatch}(eth, to, from) = sig \qquad \mathsf{isNid}(to) \qquad to \notin \mathsf{nids}(ns) \qquad (\_, msgs) \longleftarrow \mathsf{ncEffect}([\emptyset, to, \langle\emptyset, \emptyset, \emptyset\rangle], from, sig)}{[\![ns, eth]\!] \xrightarrow{to\ ?_{from}\ sig} [\![ns, eth \setminus (to, from, sig) \cdot msgs]\!]}$$

Table 6. System input rules

$$\textit{nc}\ \frac{\mathsf{ethMatch}(eth, nid, from) = sig \qquad \mathsf{destNid}(sig) = nid\ \ \vee\ \ \mathsf{destNid}(sig) = \texttt{undefined} \qquad ([pg', nid, nc'], msgs) \longleftarrow \mathsf{ncEffect}([pg, nid, nc], from, sig)}{[\![[pg, nid, nc] \|_N ns, eth]\!] \xrightarrow{nid\ ?_{from}\ sig\ ;\ \mathsf{mkAction}(msgs)} [\![[pg', nid, nc'] \|_N ns, eth \setminus (nid, from, sig) \cdot msgs]\!]}$$

$$\textit{nc}_{forward}\ \frac{\mathsf{ethMatch}(eth, nid, from) = sig \qquad \mathsf{destNid}(sig) = nid' \qquad nid \neq nid' \qquad ([pg', nid, nc'], msgs) \longleftarrow \mathsf{ncEffect}([pg, nid, nc], from, sig) \qquad msgs' = (nid', from, sig) \cdot msgs}{[\![[pg, nid, nc] \|_N ns, eth]\!] \xrightarrow{nid\ ?_{from}\ sig\ ;\ \mathsf{mkAction}(msgs')} [\![[pg', nid, nc'] \|_N ns, eth \setminus (nid, from, sig) \cdot msgs']\!]}$$

Table 7. Meta-rules for node controller

$$\textit{node}_{failure}\ \frac{msgs = \{(nid', nid, \mathbf{died}(nid, \texttt{nodedown})) \mid nid' \in \mathsf{nids}(n)\}}{[\![[pg, nid, nc] \|_N ns, eth]\!] \xrightarrow{\mathbf{die}(nid)\ ;\ \mathsf{mkAction}(msgs)} [\![ns, eth \cdot msgs]\!]}$$

$$\textit{node}_{disconnect}\ \frac{}{[\![[pg_1, nid_1, nc_1] \|_N [pg_2, nid_2, nc_2] \|_N ns, eth]\!] \xrightarrow{\mathbf{disconnect}(nid_1, nid_2)} [\![[pg_1, nid_1, nc_1] \|_N [pg_2, nid_2, nc_2] \|_N ns, eth \cdot (nid_1, nid_2, \mathbf{died}(nid_2, \texttt{disconnect})) \cdot (nid_2, nid_1, \mathbf{died}(nid_1, \texttt{disconnect}))]\!]}$$

Table 9. Node failure rules

4. Node Controller

In this section we thoroughly describe the node controller, and define how the different signals are handled. Looking at the variety of signals the node controller handle, it is a fairly complex construction. However, as we see below, much of the complexity is imaginary, the handling of each signal alone is quite straightforward; it is mostly a matter of bookkeeping.

For brevity we refrain from introducing all node controller definitions and rules. Many rules are very similar to rules presented here; for example monitor and monitor_node behave very similar to link (with the addition of *references*), register is quite similar to named send, etc. First we need to define yet another couple of functions, then we look at in turn handling of: **link**, **unlink**, **dead**, **spawn**, and **nsend** signals.

Definition 18 Let the function deleteDead(pid,nc) be defined in the obvious way; deleting all occurrences of pid from the node controller structure nc. In the case when the pid represents a node (i.e. it is in fact a nid), the nc should be cleared of all *processes* at that node as well.

***Node controller* – link/unlink** The rules for **unlink** in Table 11 are more simple than the rules for **link** in Table 10 since they do not depend on the unlinked process being alive or not. The first unlink-rule is the local instance, where the unlink is treated at the local node controller. In this case the node controller acknowledges the removal of the link. In the remote case (second unlink-rule) the link is silently removed. Combined with the rules for link one can see that as soon as the link is removed locally, there is no risk of getting an **exit**-signal later.

There are three rules for **link**, the first rule handle the case when the link is successful (either local or remote), and the last two handle the case when the to-be-linked process does not exist. In the second case, if it was a remote link (the to-be-linked process was on a different node than the linking process) the linking process is informed via a message to its node controller (the last rule), if it was a local link an **exit**-signal is constructed directly (the second last rule). In the first rule, we simply record the link in the $lnks$-set in the node controller, if it is a remote link, there might be a quick reply from the remote node controller with a **died**-signal, but that is handled by the rules for **died** (described in the next section).

At first it might seem a bit strange that there is no acknowledgement (i.e. no **message**($\{$`reply`, ...$\}$)-signal) for links, but since it is not possible to trap exits in this semantic there is no real difference between an acknowledgement followed by an **exit**-signal, or just an **exit**-signal.[2]

***Node controller* – died** The two rules in Table 12 capture all possible combinations of links and monitors to both processes and nodes. The rule is in fact too general for some cases, but the result of being too general is simply empty message-sets.

To illustrate how the rules works in practice, we consider two different situations. In the first situation a local process has terminated; this triggers the first rule. The node controller needs to send exit and monitor signals to the local processes linked to and monitoring the terminated process. The node controller also needs to communicate the termination of the process to all other node controllers where a process is linked to or monitoring the terminated process. Lastly, the node controller should remove all links, monitors and registered names for the terminated process, since these are not active anymore, this is done by the function deleteDead. It should be noted that $local\_mons$ is a list, and that $remote\_nodes$ and $local\_links$ are true sets. I.e. duplicates are possible in the $local\_mons$, but not in the others.

In the second situation a remote node has crashed (or disconnected), resulting in a **died**-signal with a node identifier being sent to the node controller. This triggers the second rule. This situation involves a bit more work for the node controller, not only should it notify local process monitoring the crashed node and then remove these monitors from the mns-set. The node controller also needs to find all links and monitors for *processes* located at the crashed node, and construct appropriate messages for them. However, in the second rule there is no need to inform remote nodes, they are already informed. Finally, in this situation as well, we clean up the node controller structure by applying the function deleteDead.

***Node controller* – spawn** In Table 13 the rule for **spawn** is presented. The handling of spawn is straightforward, a new process is created with a *fresh* pid, and the new pid is communicated back to the spawning process.

***Node controller* – nsend** In Table 14 the rules for **nsend** are presented. The two rules handle the different lookup cases. In the first rule there is a process registered for $name$ and we proceed by sending the signal to that process. In the second rule, the lookup-call returns `undefined` and the node controller simply drops the message.

5. Fairness

As we noted above, the input-rules, i.e. the rules in Table 6, can be applied in such a way that some messages are never delivered. I.e. the rules themselves does not ensure that messages are delivered in a fair manner. This is generally a bad thing, since many properties can not be proved in a non-fair system. Therefore we need to define a fairness rule that excludes certain unwanted behavior of the system. Fairness is defined in terms of permissible *execution sequences*.

Definition 19 An *execution sequence* is a sequence of node systems ns_i, together with corresponding system actions α_i written:

$$ns_0 \xrightarrow{\alpha_0} ns_1 \xrightarrow{\alpha_1} ns_2 \xrightarrow{\alpha_2} \ldots.$$

Definition 20 [**Fairness for execution sequences**] It should hold for all execution sequences, $(\vec{ns}, \vec{\alpha})$:

$$\forall i.\Big\{ ns_i \xrightarrow{pid\ !_{from}\ sig} ns_{i+1} \Rightarrow \exists j > i.\Big(ns_j \xrightarrow{pid\ ?_{from}\ sig} ns_{j+1} \Big)\Big\}$$

I.e. every message sent is eventually delivered.

6. Discussion

In this section we make a few illustrative comparisons between the current and the new semantics. We also identify some practical consequences of changing Erlang to follow the new semantics.

6.1 Everything is Distributed

The biggest, and in our eyes the most important, difference between the distributed semantics of Erlang [6] and the semantics we define in this paper is the changes made to message passing. In the current semantics (describing the current Erlang/OTP implementation) messages behave differently in a local and a distributed setting, i.e. there are different guarantees for message ordering, and

[2] Here is a distinct difference between links and monitors, since we are allowed to have more than one monitor for a pair of processes, each monitor needs to be identified by a unique reference. Thus there is an acknowledgement signal for monitor, containing this reference. Apart from this, monitors are handled similarly to links.

$$\frac{pid \in \mathsf{pids}(pg) \ \vee \ nid \neq \mathsf{node}(pid)}{\mathsf{ncEffect}([pg, nid, \langle lnks, mns, reg\rangle], from, \mathbf{link}(pid)) \longrightarrow ([pg, nid, \langle lnks \cdot (from, pid), mns, reg\rangle], \epsilon)}$$

$$\frac{pid \notin \mathsf{pids}(pg) \ \wedge \ nid = \mathsf{node}(pid) \qquad nid = \mathsf{node}(from)}{\mathsf{ncEffect}([pg, nid, nc], from, \mathbf{link}(pid)) \longrightarrow ([pg, nid, nc], (from, pid, \mathbf{exit}(\mathtt{noproc})))}$$

$$\frac{pid \notin \mathsf{pids}(pg) \ \wedge \ nid = \mathsf{node}(pid) \qquad nid \neq \mathsf{node}(from)}{\mathsf{ncEffect}([pg, nid, nc], from, \mathbf{link}(pid)) \longrightarrow ([pg, nid, nc], (\mathsf{node}(from), nid, \mathbf{died}(pid, \mathtt{noproc})))}$$

Table 10. Node controller effect and reply for **link**

$$\frac{nid = \mathsf{node}(from)}{\begin{array}{l}\mathsf{ncEffect}([pg, nid, \langle lnks, mns, reg\rangle], from, \mathbf{unlink}(pid)) \longrightarrow \\ \quad ([pg, nid, \langle lnks \setminus (from, pid), mns, reg\rangle], (from, nid, \mathbf{message}(\{\mathtt{reply}, \mathtt{unlink}, pid\})))\end{array}}$$

$$\frac{nid \neq \mathsf{node}(from)}{\mathsf{ncEffect}([pg, nid, \langle lnks, mns, reg\rangle], from, \mathbf{unlink}(pid)) \longrightarrow ([pg, nid, \langle lnks \setminus (from, pid), mns, reg\rangle], \epsilon)}$$

Table 11. Node controller effect and reply for **unlink**

$$\frac{\begin{array}{rcl} & & \neg\mathsf{isNid}(pid) \\ local\_links & = & \{(pid_1, pid_2, \mathbf{exit}(v)) \mid (pid_1, pid_2) \leftarrow lnks \ \wedge \ \mathsf{node}(pid_1) = nid \ \wedge \ pid_2 = pid\} \\ remote\_nodes & = & \{(\mathsf{node}(pid_1), nid, \mathbf{died}(pid, v)) \mid (pid_1, pid_2) \in lnks \cup mns \ \wedge \ \mathsf{node}(pid_1) \neq nid \ \wedge \ pid_2 = pid\} \\ local\_mons & = & [(pid_1, pid_2, \mathbf{message}(\{\mathtt{reply}, \mathtt{monitor}, \{ref, pid, v\}\})) \\ & & \ \| \ (ref, pid_1, pid_2) \leftarrow mns \ \wedge \ \mathsf{node}(pid_1) = nid \ \wedge \ pid_2 = pid]\end{array}}{\begin{array}{l}\mathsf{ncEffect}([pg, nid, \langle lnks, mns, reg\rangle], from, \mathbf{died}(pid, v)) \longrightarrow \\ \quad ([pg, nid, \mathsf{deleteDead}(pid, \langle lnks, mns, reg\rangle)], local\_links \cdot local\_mons \cdot remote\_nodes)\end{array}}$$

$$\frac{\begin{array}{rcl} & & \mathsf{isNid}(nid') \\ local\_links & = & \{(pid_1, pid_2, \mathbf{exit}(v)) \mid (pid_1, pid_2) \leftarrow lnks \ \wedge \ \mathsf{node}(pid_1) = nid \ \wedge \ \mathsf{node}(pid_2) = nid'\} \\ local\_mons & = & [(pid_1, pid_2, \mathbf{message}(\{\mathtt{reply}, \mathtt{monitor}, \{ref, pid, v\}\})) \\ & & \ \| \ (ref, pid_1, pid_2) \leftarrow mns \ \wedge \ \mathsf{node}(pid_1) = nid \ \wedge \ (pid_2 = pid \ \vee \ \mathsf{node}(pid_2) = nid')]\end{array}}{\begin{array}{l}\mathsf{ncEffect}([pg, nid, \langle lnks, mns, reg\rangle], from, \mathbf{died}(nid', v)) \longrightarrow \\ \quad ([pg, nid, \mathsf{deleteDead}(nid', \langle lnks, mns, reg\rangle)], local\_links \cdot local\_mons)\end{array}}$$

Table 12. Node controller effect and reply for **died**

$$\frac{pid' = \mathsf{fresh}()}{\mathsf{ncEffect}([pg, nid, nc], from, \mathbf{spawn}(e, ref)) \longrightarrow ([\langle e, pid', \epsilon\rangle \|_{_P} pg, nid, nc], (from, nid, \mathbf{message}(\{\mathtt{reply}, ref, pid'\})))}$$

Table 13. Node controller effect and reply for **spawn**

$$\frac{\mathsf{lookup}(name, reg) = pid}{\mathsf{ncEffect}([pg, nid, \langle lnks, mns, reg\rangle], from, \mathbf{nsend}(name, v)) \longrightarrow ([pg, nid, \langle lnks, mns, reg\rangle], (pid, from, \mathbf{message}(v)))}$$

$$\frac{\mathsf{lookup}(name, reg) = \mathtt{undefined}}{\mathsf{ncEffect}([pg, nid, \langle lnks, mns, reg\rangle], from, \mathbf{nsend}(name, v)) \longrightarrow ([pg, nid, \langle lnks, mns, reg\rangle], \epsilon)}$$

Table 14. Node controller effect and reply for **nsend**

message delivery depending on whether the receiving process is local or remote. The change to a fully asynchronous message passing should enable a more efficient implementation of the run-time system. However, there is also a downside with the more permissive semantics. Less restrictions results in more possible interleavings, and in effect a larger state space to explore during verification.

Further, the choice of making all side effects (with the exception of `spawn_node`) purely asynchronous was motivated two-fold. (1) it made the resulting semantics a lot shorter, and therefore easier to understand, and (2) it should make the operations easier to parallelize in a future implementation of the semantics. These bare asynchronous behaviors would in many situations be rather inconvenient for the programmer to use, for example when spawning a process or registering a named process. But nothing stops us from defining (as a BIF or equivalent) the well-known synchronous variants that are used in Erlang today, by simply encapsulating the side-effecting action and the waiting for confirmation.

6.2 Uni-directional Links Only

Another part of the current semantics that we have found non-intuitive is the bi-directional links. We find it a bit strange that a process can affect the behavior of another process behind its back by linking to it. (E.g. in the situation where process A first links to process B and then exits abnormally, resulting in that B, unless it is trapping exits, suddenly crashes.) Also the functional overlap of monitors and trapped exits (links) makes for extra complications in the semantics. Instead we propose to only have uni-directional links (and monitors, but they are already uni-directional), and to remove the whole exit-trapping functionality since a monitor serves the same purpose.

The practical consequences of this change are not too dramatic, but it becomes a bit more complicated to build the important *supervision trees*. However, since supervision structures are most of the time built using the OTP-behavior (`supervisor`), it should be enough to change the supervisor implementation to compensate. The `spawn_link` construction is also easily mimicked by a slight change to its definition.

6.3 A Built-in Registry

The built-in registry is another difference between the current and the new semantics. However, its workings should be easily made equivalent to the standard process registry implementation in Erlang. Also, because of the introduction of a general treatment of side-effecting actions, the inclusion of a registry affects the size of the semantics marginally. The size argument is important, since the main priority of the new semantics is to keep it reasonably small.

6.4 Message-Passing Guarantees

The message-passing guarantees has changed dramatically in the new semantics. There are no longer any difference between local and distributed message-passing, and only the order of messages between a pair of processes is guaranteed.

7. Conclusions and Future Directions

We have presented a proposal for a future semantics of Erlang. However, the observant reader might have noticed that there are very few references to Erlang in the description of the semantics. That is, perhaps the language for this semantics is not Erlang, maybe there is another language waiting around the corner?

Nevertheless, we hope that the presented semantics is easy enough to follow that it can serve as a discussion starter when it comes to improving and changing Erlang in the future. It is our strong belief that the current message-passing guarantees must be removed in order to fully take advantage of an extreme (32+) multi-core architecture.

On the other hand, perhaps we can continue to work with bi-directional links, although they are slightly unintuitive they have after all worked fairly well for many years. It might not be worth trying to make the change.

Another side to the discussion is the implications for model checking. From a model checking perspective it makes perfect sense to impose stronger guarantees for message-passing, as we have in the current semantics. The result is a smaller state space, a crucial detail when model checking. Therefore, from a model checking perspective, the new, more permissive, semantics is potentially going to cause problems. Everything is not lost, it should be possible to find reduction techniques to compensate for the enlarged state space, but it will require a bit of thought.

Future Directions Since we have access to an experiment platform, McErlang [4], the obvious next step is to implement this semantics there. Having such an implementation is going to make it easier to analyze semantic design decisions in more detail.

Since, as mentioned above, the new semantics results in larger state spaces during model checking we should also put some effort into partial order reduction techniques for the new semantics. Otherwise, it will become hard to perform model checking for systems under the more asynchronous semantics.

Acknowledgments

This research was sponsored by EU FP7 Collaborative project *ProTest*, grant number 215868. Many thanks to the anonymous reviewers, whose comments significantly improved the presentation of the semantics. Thanks also to Koen Claessen for stimulating discussions and helpful comments.

References

[1] J. Armstrong. *Making reliable distributed systems in the presence of software errors*. PhD thesis, Royal Institute of Technology, Stockholm, Sweden, December 2003.

[2] K. Claessen and H. Svensson. A semantics for distributed Erlang. In *Proc. of the ACM SIGPLAN workshop on Erlang*, pages 78–87, New York, NY, USA, 2005. ACM Press.

[3] L-Å. Fredlund. *A Framework for Reasoning about Erlang Code*. PhD thesis, Royal Institute of Technology, Stockholm, Sweden, 2001.

[4] L-Å. Fredlund and H. Svensson. McErlang: a model checker for a distributed functional programming language. In *Proceeding of the 12th ACM SIGPLAN Int. Conf. on Functional Programming (ICFP)*, pages 125–136, Freiburg, Germany, 2007. ACM.

[5] M. Petterson. A definition of Erlang (draft). Manuscript, Department of Computer and Information Science, Linköping University, 1996.

[6] H. Svensson and L-Å. Fredlund. A more accurate semantics for distributed Erlang. In *Proc. of the SIGPLAN workshop on Erlang*, pages 43–54, New York, USA, 2007. ACM.

[7] H. Svensson and L-Å. Fredlund. Programming distributed Erlang applications: pitfalls and recipes. In *Proc. of the SIGPLAN workshop on Erlang*, pages 37–42, New York, USA, 2007. ACM.

References

[1] J. Armstrong. *Making reliable distributed systems in the presence of software errors*. PhD thesis, Royal Institute of Technology, Stockholm, Sweden, December 2003.

[2] K. Claessen and H. Svensson. A semantics for distributed Erlang. In *Proc. of the ACM SIGPLAN workshop on Erlang*, pages 78–87, New York, NY, USA, 2005. ACM Press.

[3] L-Å. Fredlund. *A Framework for Reasoning about Erlang Code*. PhD thesis, Royal Institute of Technology, Stockholm, Sweden, 2001.

[4] L-Å. Fredlund and H. Svensson. McErlang: a model checker for a distributed functional programming language. In *Proceeding of the 12th ACM SIGPLAN Int. Conf. on Functional Programming (ICFP)*, pages 125–136, Freiburg, Germany, 2007. ACM.

[5] M. Petterson. A definition of Erlang (draft). Manuscript, Department of Computer and Information Science, Linköping University, 1996.

[6] H. Svensson and L-Å. Fredlund. A more accurate semantics for distributed Erlang. In *Proc. of the SIGPLAN workshop on Erlang*, pages 43–54, New York, USA, 2007. ACM.

[7] H. Svensson and L-Å. Fredlund. Programming distributed Erlang applications: pitfalls and recipes. In *Proc. of the SIGPLAN workshop on Erlang*, pages 37–42, New York, USA, 2007. ACM.