A Semantics for Distributed Erlang

Koen Claessen
Dept. of Computing Science
Chalmers University of Technology
Gothenburg, Sweden
koen@cs.chalmers.se

Hans Svensson
Dept. of Computing Science
Chalmers University of Technology
Gothenburg, Sweden
hanssv@cs.chalmers.se

Abstract

We propose an extension to Fredlund's formal semantics for Erlang that models the concept of nodes. The motivation is that there exist sequences of events that can occur in practice, but are impossible to describe using a single-node semantics, such as Fredlund's. The consequence is that some errors in distributed systems might not be detected by model checkers based on Fredlund's original semantics, or by other single-node verification techniques such as testing. Our extension is modest; it re-uses most of Fredlund's work but adds an extra layer at the top-level.

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

General Terms Languages, Theory, Verification

Keywords Erlang, semantics, distributed systems, verification

1. Introduction

Most software written in Erlang is running in distributed environments, and is often highly concurrent and dynamic in nature. Experience shows that such software is inherently hard to write, test and verify. Several approaches have been proposed for testing [6, 7, 8, 18, 5] and formally verifying [16, 15, 3] Erlang programs. One important aspect in the work with verification techniques such as model checking is a formal semantics for Erlang.

Fredlund [12] proposes a formal semantics for Erlang, which is described in more detail in [13]. Fredlund's semantics is a small-step operational semantics that is simple, easy to understand and constructed in a layered fashion. The semantics has been used as a basis in several different verfication projects, such as semi-formal verification of Erlang code [14] and model checking a resource manager [15]. Fredlund's semantics has also been a basis for the development of a theorem prover [15] and a translation of Erlang into a language that can be model checked [4].

Recently, we discovered two previously undiscovered errors in an open source Erlang implementation of a leader election algorithm [19, 5]. Both errors where caused by chains of events not foreseen by the designer, and were related to message arrival order in the distributed environment. As we later show with examples, using message passing in a distributed Erlang system requires at least some extra thought — it is not always obvious to know what the possible chains of events are in a situations where several processes are sending messages to each other. Especially, messages do not always arrive in the expected, intuitive order.

What could be more appropriate than to apply a model checker in this situation? A model checker would be able to prove that, even in unforeseen chains of events, the desired properties of the system still hold. However, such a model checker needs to be aware of what the possible chains of events actually are. For this to work properly, actual chains of events that can happen should be reflected by the underlying semantics that the model checker uses.

The errors we found in the leader election implementation were both specific to a *multi-node setting*. This means that the chains of events that exposed the errors can only occur when different parts of the system run on different nodes. Thus, contrary to the the Erlang idea that distribution should be transparent, there exist a real behavioural difference between systems where all processes run on the same node, and between systems where processes run on different nodes.

Unfortunately, Fredlund's Erlang semantics is also unable to foresee the chains of events that lead to the errors. The reason is that Fredlund's semantics does not contain the concept of nodes. All systems described in Fredlund's semantics are localized to the same runtime system, which is the reason why we will call his semantics a *single-node* semantics.

Message passing in Erlang is normally thought of to work similarily in a local and a distributed setting. However, Erlang message passing actually behaves slightly different in a local setting compared to the distributed setting. Since Fredlund's single-node semantics does not contain this distinction, it is impossible to faithfully model certain distributed systems.

Contributions The first contribution of this paper to the Erlang community is a warning: The behaviour of message passing between processes running on the same node is limited compared to processes on different nodes. This is moreover not just a theoreticl remark; we have found a real-life example where this actually matters. This leads to two conclusions: (1) When formally verifying Erlang systems (intended to run in a distributed setting) using a model checker based on a single-node semantics, there might be errors that slip through. But also, (2) when testing Erlang systems (intended to run in a distributed setting) using a single runtime system containing all processes, there might be errors that slip through.

The second contribution is a proposal to add one more layer on top of Fredlund's single-node semantics that formally descibes the concept of nodes, message passing between nodes, and the node linking mechanism. The hope is that this new semantics can be the formal basis for future model checkers for Erlang.

Erlang'05 September 25, 2005, Tallinn, Estonia.

Summary Sect. 2 contains an introduction to Fredlund's single-node semantics. In Sect. 3 we present some motivating examples, as well as a description of situations where Fredlund's Erlang semantics lacks expressive power. In Sect. 4 we provide an extension to Fredlund's semantics, where we add another layer on top of the existing semantics in order to introduce the full distributed behavior. Some of the design decisions in the extended semantics is discussed in Sect. 5, related approaches to the problem are described in Sect. 6 and we conclude in Sect. 7.

2. Original semantics

In [13], Fredlund gives a complete presentation of a small-step operational semantics for Erlang. Here we will highlight some of the most important aspects, with enough details to be able to understand the presentation of the extended semantics. Fredlund's single-node semantics is presented for a subset of Erlang, that is in short standard Erlang without: modules, nodes, floats, references, binaries, ports and the catch-expression. Some of the process' internal state has also been omitted: there are no process dictionaries, no group leader or processes groups and name-registration for processes is not included.

All definitions and rules presented in this section are taken from Fredlund's presentation of the semantics [13], with the exception that we in a few cases leave out details not relevant for this article in order to make the presentation clearer. Fredlund's semantics is separated into two parts; one functional part, with evaluation of expressions and one concurrent part where processes are spawned and messages are sent and delivered. Fredlund's single-node semantics is presented here in roughly the same order as in the original presentation [13], starting with *expression evaluation rules* then defining processes and finally stating *process evaluation rules*.

Definition 1. Erlang expressions are ranged over by $e \in \mathsf{erlangExpr}$; Erlang values (non-reducible expressions) are ranged over by $v \in \mathsf{erlangVal}$.

The semantics is provided in terms of transition rules on the format

$$\frac{t_1 \xrightarrow{\alpha_1} t_1' \ \ldots \ t_n \xrightarrow{\alpha_n} t_n' \qquad \varphi_1 \ \ldots \ \varphi_m}{t \xrightarrow{\alpha} t'}$$

where each φ_i is a logic side-condition that does not refer to any transition relation.

Definition 2. The expression actions, ranged over by $\alpha \in \mathsf{erlangExprAction}$, are:

$\gamma ::=$	τ	computation step
$\mid$	$pid\,!\,v$	output
$\mid$	$\mathsf{exiting}(v)$	exception
$\mid$	$\mathsf{read}(q, v)$	read from queue
$\mid$	$\ldots$	

Definition 3. The expression transition relation $\rightarrow$: $\mathsf{erlangExpr} \times \mathsf{erlangExprAction} \times \mathsf{erlangExpr}$, written $e_1 \xrightarrow{\alpha} e_2$ when $\langle e_1, \alpha, e_2 \rangle \in \rightarrow$, is the least relation satisfying the transition rules in [13]. In Fig. 1 we have listed Fredlund's rules for send and receive at expression level.

In Fig. 1 we have listed the rules for evaluation of send and receive expressions. The send-rules are fairly straightforward, both terms are evaluated until finally a $pid\,!\,v$-action is generated. The receive-rule is more complicated, and won't be explained in detail. The intuition is that q is a prefix to the complete message queue, and none of the messages in that prefix matches any of the patterns

$$\textit{send}_0 \ \frac{}{pid\,!\,v \xrightarrow{pid!v} v}$$

$$\textit{send}_1 \ \frac{e_1 \xrightarrow{\alpha} e_1'}{e_1\,!\,e_2 \xrightarrow{\alpha} e_1'\,!\,e_2}$$

$$\textit{send}_2 \ \frac{e \xrightarrow{\alpha} e'}{v\,!\,e \xrightarrow{\alpha} v\,!\,e'}$$

$$\textit{receive} \ \frac{\begin{array}{c}\forall i.\neg(\mathsf{qmatches}\ q\ m_i) \\ \exists i.((\mathsf{result}\ v\ m_i\ e') \wedge \forall j.j < i \Rightarrow \neg(\mathsf{matches}\ v\ m_j))\end{array}}{\mathtt{receive}\ m\ \mathtt{end} \xrightarrow{\mathsf{read}(q,v)} e'}$$

Figure 1. Expression evaluation rules

in m. Also, there exist a pattern in m, such that it is the first one to match v, and when substituting v according to that pattern its corresponding expression become e'.

Next we need to formalize the notion of processes, which encapsulate Erlang expressions, and the notion of Erlang systems, which are collections of processes. Erlang processes, ranged over by $p \in \mathsf{erlangProcess}$, are either live or dead. The dead processes are introduced to make it easier to reason about the semantics of linked processes. Processes that are dead still perform some actions; they will eventually inform linked process about their termination, and they do respond to received link signals.

Definition 4. An Erlang mailbox is queue data structure, in theory unbound, thus it can store any number of messages. Mailboxes are ranged over by $q \in \mathsf{erlangQueue}$

Definition 5. A **live** *Erlang process* ($\mathsf{erlangLiveProcess} \subset \mathsf{erlangProcess}$), is a quintuple: $\mathsf{erlangExpr} \times \mathsf{erlangPid} \times \mathsf{erlangQueue} \times \mathcal{P}(\mathsf{erlangPid}) \times \mathsf{erlangBool}$, written $\langle$ *e, pid, q, pl, b* $\rangle$ such that

- *e* is an Erlang expression,
- *pid* is the process identifier of the process,
- *q* is a message queue,
- *pl* is a set of process identifiers (a set of links with other processes),
- *b* is a boolean determining how process exit notifications are handled.

Definition 6. A terminated (dead) Erlang process ($\mathsf{erlangDeadProcess} \subset \mathsf{erlangProcess}$) is a tuple:
$\mathsf{erlangPid} \times \mathcal{P}\ (\mathsf{erlangPid} \times \mathsf{erlangVal})$, written $\langle$ *pid, plm* $\rangle$, where

- *pid* is the process identifier of the process,
- *plm* is a set of tuples, combining process identifiers with a notification value that should be sent to the corresponding process.

Definition 7. An *Erlang system*, ranged over by s, is either a singleton process or a combination of systems s_1 and s_2, written as $s_1 \parallel s_2$.

Intuitively, the composition of processes into Erlang systems could be thought of as a set of processes. The $\parallel$ operator is commutative and associative. When there is no risk for confusion, we omit the linked processes parameter and the boolean flag from the live processes, that is they are written as $\langle$ *e, pid, q* $\rangle$. The *signals* are items of information transmitted between a sending and a receiving

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

$$\textit{output}_1\ \frac{e \xrightarrow{pid'\,!\,v} e' \qquad pid' \neq pid}{\langle e, pid, q, pl, b\rangle \xrightarrow{pid'\,!\,\mathsf{message}(v)} \langle e', pid, q, pl, b\rangle}$$

$$\textit{output}_2\ \frac{e \xrightarrow{pid\,!\,v} e'}{\langle e, pid, q, pl, b\rangle \xrightarrow{\tau} \langle e, pid, q \cdot v, pl, b\rangle}$$

$$\textit{input}\ \frac{}{\langle e, pid, q, pl, b\rangle \xrightarrow{pid\,?\mathsf{message}(v)} \langle e, pid, q \cdot v, pl, b\rangle}$$

$$\textit{link}\ \frac{}{\langle e, pid, q, pl, b\rangle \xrightarrow{pid\,?\,\mathsf{link}(pid')} \langle e, pid, q, pl \cup \{pid'\}, b\rangle}$$

$$\textit{term}\ \frac{}{\langle e, pid, q, pl, b\rangle \xrightarrow{\tau} \langle pid, \{\langle P, \mathsf{normal}\rangle | P \in pl\}\rangle}$$

Figure 2. Process evaluation rules

$$\textit{com}\ \frac{s_1 \xrightarrow{pid\,!\,sig} s_1' \qquad s_2 \xrightarrow{pid\,?\,sig} s_2'}{s_1 \parallel s_2 \xrightarrow{\tau} s_1' \parallel s_2'}$$

$$\textit{interleave}\ \frac{s_1 \xrightarrow{\tau} s_1' \qquad \mathsf{pids}(s_1') \cap \mathsf{pids}(s_2) = \emptyset}{s_1 \parallel s_2 \xrightarrow{\tau} s_1' \parallel s_2}$$

Figure 3. Process communication rules

process. A *system action*, committed by an Erlang system is either a silent action, an input action or an output action. We should also define the system transition relation.

Definition 8. The signals, ranged over by $sig \in \mathsf{erlangSignal}$ are:

$sig ::=$	$\mathsf{message}(v)$	message
$\mid$	$\mathsf{link}(pid)$	linking with process
$\mid$	$\mathsf{unlink}(pid)$	unlinking process
$\mid$	$\ldots$	

Definition 9. The system actions, ranged over by $\alpha \in \mathsf{erlangSysAction}$ are:

$\alpha ::=$	τ	silent action
$\mid$	$pid\,!\,sig$	output action
$\mid$	$pid\,?\,sig$	input action

Definition 10. The system transition relation $\rightarrow$: $\mathsf{erlangSystem} \times \mathsf{erlangSysAction} \times \mathsf{erlangSystem}$, written $s_1 \xrightarrow{\alpha} s_2$, is the least relation satisfying the transition rules in [13]. Some of those rules are listed here in Fig. 2 and Fig. 3.

The rules in Fig. 2 show how processes perform a computation step, terminates and sends and receives messages. Note that messages sent to the same process are delivered immediately (*output*$_2$). Also note that messages to other processes are transferred to the above layer by a visible ($pid'!\mathsf{message}(v)$) system action. The rules

```
procA() ->
    PidC = spawn(?NODE2,?MODULE,funC,[]),
    PidB = spawn(?NODE1,?MODULE,funB,[PidC]),
    PidC ! hello,
    PidB ! world.

procB(PidC) ->
    receive X ->
    PidC ! X
    end.

procC() ->
    receive X ->
    ok
    end.
```

Figure 4. Erlang program - Message reordering

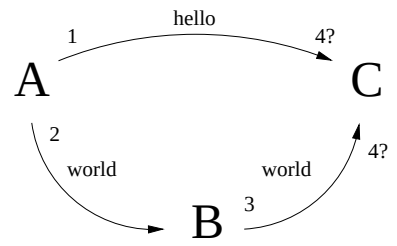

Figure 5. Message passing

in Fig. 3 show how processes communicate and how computations are interleaved, note that the communication rules also exist in a symmetric version where the roles of s_1 and s_2 are interchanged. This concludes the introduction to the original semantics, an example with Fredlund's single-node semantics in use is presented in Sect. 3.

3. Motivation

In our work with a leader election protocol [5], we saw several cases where problems arise due to unforseen order of events. Especially problematic were situations when messages arrived in what was thought to be an impossible order. To investigate this problem further, we constructed the Erlang program listed in Fig. 4. This Erlang program (process A) first spawns two processes (C and B, and passes the process identifier of C to B) and then sends a message, *hello*, directly to process C. Next the program sends another message, *world*, to process B. When process B receives a message, it is immediately re-sent to process C. Process C does only one thing, namely receives one message. Intuitively, process C will receive the message *hello*, since it is sent directly from A to C. However, in the fundamental ideas behind Erlang [1] the only thing said about message order is *'Message passing between a pair of processes is assumed to be ordered'*. This means that without violating this property *world* should be able to arrive before *hello*, since we have no guarantees for the relative message order when the messages are sent on different routes.

The program in Fig. 4 was executed in three different situations

1. A,B and C where executed in the same runtime system.
2. A,B and C where executed on the same physical machine, but in separate runtime systems.
3. A,B and C where executed on three different physical machines connected via a 100 MBit Ethernet network, thus running in separate runtime systems.

The results where somewhat surprising. If the execution follows the intuition that it is faster to deliver a message directly, *hello* should always arrive first; if the Erlang ideas where implemented faithfully we should see both *hello* and *world* arriving first in all three situations. However, in situation (1) *hello* always arrive first, in situations (2) and (3) we see both *hello* and *world* arriving first. In situation (2), *world* would arrive first 10-15 times out of 10 000 and in situation (3) *world* would arrive first somewhat more frequently, 20-25 times out of 10 000. (Still those numbers vary from experiment to experiment, due to different network load and different machine load.) That is, the Erlang runtime system implementation behave differently in a local setting as compared to in a distributed setting. This partly explain why errors such as those found in [5] appear to be common.

Another reason is that Erlang programmers often think of their system in an event-based way: "First this process dies, then that process sends a message, then that message is sent...". In other words they have a conceptual model of the many possible orders in which the events can be generated. The semantics adds additional possibilities in the form of the possible orders in which the events actually arrive. This extra complexity may be hard to deal with and the speed with which messages are delivered allows programmers to often only think in forms of generated events. Thus, if one does not think carefully enough, it is easy to be misled and overlook something.

Message reordering in Fredlund's semantics

What happens if we try to analyze the program in Fig. 4 with Fredlund's single-node semantics? Since Fredlund chooses not to include nodes in his semantics, it is not too surprising that the program will behave as in situation (1) above, as we can see in Fig. 6. The desire to also get the behavior in situations (2) and (3) serves as the motivation for extending Fredlund's semantics to be able to fully reason about distributed Erlang systems. It is especially important in case we use the semantics to produce a model, if certain situations are not present in the model, errors may be overlooked, and thus giving false confidence.

In the example in Fig. 6, we use P_i as a short hand notation for the process with identifier p_i, we also use the short process notation leving out the linked processes parameter and the boolean flag.

4. Multi-node Semantics

We extend Fredlund's single-node semantics to a distributed setting in a layered fashion, i.e. by adding another layer on top of the existing semantics to deal with all aspects of distributed Erlang systems. One implication of this is that everything defined in terms of the original semantics is still valid in the extended semantics under the restriction that the system is local, i.e. running on the same node. To be fully operational we need to make some restrictions to the extended semantics, namely we need to ensure *fairness*. Finally we demonstrate that the extended semantics work as intended, by studying the same example as in Sect. 3 in the extended semantics.

Here, we present the definitions and semantic rules needed to extend Fredlund's semantics to also include nodes and distributed execution. Firstly, we add the possibility to *spawn* processes on other nodes. To be able to do this, we have to extend the concept of *Erlang systems* to *Erlang Runtime systems*, i.e. a single node, and also *Erlang Multi-node systems* which are collections of nodes forming complete distributed systems. Secondly, we need new rules for communication between processes on different nodes (i.e. different runtime systems). These communication rules should have the properties described in Sect. 3, and thus enable certain message reordering. Thirdly, we add the concept of nodes that die and get restarted again, together with a linking mechanism to send a warning when a node dies.

4.1 Messages

The message ordering induced by a single-node semantics is too static; certain message reorderings are not considered. We achieve the distributed ordering by introducing one message queue *per node*, holding all messages 'in transit' to that node.

We first present the definitions introducing the new concepts and slight changes to the underlying semantics and then present the evaluation rules for the additional objects.

Definition 11. Let the function node(erlangPid) return the node for a given process id (pid) and node(erlangSystem) returns the node for an Erlang system.

The node identifier could be any unique identifier. For the sake of simplicity, we can assume that they are integers.

Everything defined in the original semantics will work in the extended semantics, with one exception. We have to change the *com*-rule (Fig. 3) slightly so that it only applies in the correct situation where both processes are running on the same node. Further we also need to do a small modification in order to export extra information about the sender of a message to the additional layer we are adding. This is done by replacing the sending operator ! with a tagged version $!_{from}$. This change is straightforward and is applied to all the send operators in the original semantics. One example can be seen in the new *com*-rule presented in Fig. 7, where we have added a side condition such that the rule only applies when the sender and the receiver are running in the same Erlang system.

$$com\ \frac{s_1 \xrightarrow{pid!_{from} sig} s_1' \quad s_2 \xrightarrow{pid?sig} s_2' \quad \mathsf{node}(pid) = \mathsf{node}(from)}{s_1 \parallel s_2 \xrightarrow{\tau} s_1' \parallel s_2'}$$

Figure 7. New com-rule

Definition 12. An *Erlang node message queue*, ranged over by $nq \in$ erlangNodeQueue, is a finite sequence of triplets, $v_1 = (from_1, to_1, sig_1) \cdot v_2 = (from_2, to_2, sig_2) \cdot \ldots \cdot v_n$, where ϵ is the empty sequence, $(\cdot)$ is concatenation and $(\setminus)$ is deletion of the first matching triplet, e.g.

$$nq = (a_2, b_1, c_1) \cdot (a_1, b_2, c_1) \cdot (a_1, b_2, c_2) \cdot (a_1, b_2, c_1) \setminus (a_1, b_2, c_1)$$
$$= (a_2, b_1, c_1) \cdot (a_1, b_2, c_2) \cdot (a_1, b_2, c_1)$$

4.2 Runtime systems

Next, we define the concepts of alive and dead runtime systems.

Definition 13. A **live** *Erlang Runtime system (ERTS)*, ranged over by $r \in$ erlangRuntimeSystem is a triplet:
erlangSystem $\times$ erlangNodeName $\times$ erlangNodeQueue, written $\langle$ s, node, nq $\rangle$ such that:

- *s* is the Erlang system at node *node*.
- *node* is the node identifier (name).
- *nq* is a node message queue.

Definition 14. A **stopped** *Erlang Runtime system* is a triplet: erlangNodeName $\times$ $\mathcal{P}$(erlangPid) $\times$ $\mathcal{P}$(erlangPid $\times$ erlangPid), written $[\![$ *node, npl, nlm* $]\!]$. Where *npl* is a set process identifiers (of all processes on *node*), and *nlm* is a set of tuples combining the process identifier of a terminated process with the process identifier for a process that is to be notified of the termination. In addition we also introduce a new way of writing a *terminated Erlang process*, previously written $\langle$ *pid, plm* $\rangle$ and instead write this as $[\![$ *pid, plm* $]\!]$.

An example of a stopped Erlang Runtime system is: $[\![n_1, \{p_1, p_5, p_{13}\}, \{(p_1, p_7), (p_{13}, p_7), (p_{13}, p_{24})\}]\!]$ where the node

Initial system:

$$\langle \texttt{PidC} = \texttt{spawn}(\texttt{procC}, []) \ \ldots, p_0, [] \rangle$$

After spawns:

$$\underbrace{\langle \texttt{PidC} \,!\, \texttt{hello} \ldots, p_0, [] \rangle \parallel \langle \texttt{receive}\ X \rightarrow \texttt{ok end}, p_1, [] \rangle \parallel \langle \texttt{receive}\ X \rightarrow \texttt{PidC} \,!\, X\ \texttt{end}, p_2, [] \rangle}_{s_1}$$

Only p_0 can make progress:

$$s_1 \xrightarrow{p_1 ! \mathsf{message}(\textit{hello})} \underbrace{\langle \texttt{PidB} \,!\, \texttt{world}, p_0, [] \rangle \parallel \langle \texttt{receive}\ X \rightarrow \texttt{ok end}, p_1, [\texttt{hello}] \rangle \parallel P_2}_{s_2}$$

Here s_2 can either proceed by p_1 reading *hello* from the message-queue or by p_0 sending a message, the first alternative will clearly result in receiving *hello* first, so let us see what happens in the second case:

$$s_2 \xrightarrow{p_2 ! \mathsf{message}(\textit{world})} \underbrace{\langle p_0, [] \rangle \parallel P_1 \parallel \langle \texttt{receive}\ X \rightarrow \texttt{PidC} \,!\, X\ \texttt{end}, p_2, [\texttt{world}] \rangle}_{s_3}$$

Yet again we have a choice, s_3 can either proceed by p_1 reading hello or by p_2 receiving and sending a messge, still it is the second case that is interesting here:

$$s_3 \xrightarrow{\tau} \underbrace{P_0 \parallel P_1 \parallel \langle \texttt{PidC} \,!\, \texttt{world}, p_2, [] \rangle}_{s_4}$$

and further:

$$s_4 \xrightarrow{p_1 ! \mathsf{message}(\textit{world})} P_0 \parallel \langle \texttt{receive}\ X \rightarrow \texttt{ok end}, p_1, [\texttt{hello}, \texttt{world}] \rangle \parallel \langle p_2, [] \rangle$$

And also in this case there is no possibility to receive *world* before *hello*, with this little example we have shown that in the original semantics, no message reordering is possible.

Figure 6. Execution sequence in Fredlund's single-node semantics

identifier is n_1, the processes that has executed on n_1 are p_1, p_5 and p_{13} and there are links between p_1 and p_7 etc. Note also that $\mathsf{node}(p_5) = n_1$.

Definition 15. An *Erlang Multi-node system* (EMNS) is either a singleton ERTS or a composition of Erlang Multi-node systems n_1 and n_2, written as $n_1 \parallel n_2$.

Note that here we have chosen to use the same notation ($\parallel$) for composition of Erlang Multi-node systems as for the composition of Erlang systems in the original semantics. This is to illustrate that they are similar in behavior. Moreover, there is little risk for confusion.

4.3 Transitions

When multi-node systems make transitions, they are labelled by actions. The actions that can occur at the level of nodes are defined below.

Definition 16. The Multi-node system actions, ranged over by $\gamma \in$ erlangMultiNodeSysAction are:

$$\begin{array}{rcll} \gamma ::= & \tau & & \text{silent action} \\ | & pid!_{\textit{from}} sig & & \text{output action} \\ | & pid?_{\textit{from}} sig & & \text{input action} \\ | & \mathsf{die}(node) & & \text{node failure} \end{array}$$

That is, the actions visible in erlangMultiNodeSysAction are only the node-to-node communication and node failure. Messages sent between processes executing on the same node is not visible at this level. Note also that at this level the input actions ($?$) is tagged with a *from*. This is not necessary from a functionality point of view, but as we see below, *fairness* can be expressed in a simple and elegant way in the presence of the tagged input action.

Definition 17. The Multi-node system transition relation, $\rightarrow$: erlangMultiNodeSystem $\times$ erlangMultiNodeSysAction $\times$ erlangMultiNodeSystem, written $n_1 \xrightarrow{\gamma} n_2$, is the least relation satisfying the rules in Fig. 8 – Fig. 12.

Definition 18. The function links(erlangNodeQueue) traverses an Erlang node message queue and picks out all pending link-request from this queue. The function init() is an initialization process which is started on a new node, what this process does is not further specified in the semantics. Finally the function gpl(erlangPid) return the *pl*-list (list of linked nodes) for a process, given the process identifier of that process.

These three functions are related to failing/restarting nodes. The init-function should be thought of as any reasonable, and changeable from the outside, starting action for an Erlang node. For example starting a certain set of processes, or initiate some other chain of events. The links and gpl are mere bookkeeping-functions defined to express what happens in case of node failure in a comprehensible way.

Definition 19. The function nmatch(erlangNodeQueue, erlangPid, erlangPid), is a function that given an Erlang node message queue, a sender process id (*from*) and a receiver process id (*to*) returns the first message in the queue sent by *from* to *to*, e.g.

$$\begin{array}{rcl} nq & = & (a_2, b_1, c_1) \cdot (a_1, b_2, c_1) \cdot (a_1, b_2, c_2) \cdot (a_1, b_2, c_1) \\ & \Rightarrow & \mathsf{nmatch}(nq, a_1, b_2) = (a_1, b_2, c_1) \end{array}$$

The function nmatch is later used to retreive a message from the node message queue in a non-deterministic fashion.

4.4 Operational rules

In Fig. 8 we see the *node-failure*-rule. It states that when a node dies, a *stopped ERTS* with the same node identifier is created. The stopped node contains both a list of all the processes previously running on the node and a collected list of link-notifications to be sent. Observe that the link-notifications are collected both from the individual processes and the node message queue *nq*. The intuition

$$\textit{node-failure}\ \frac{\textit{nlm} = \{(pid, pid') | pid' \in \mathsf{gpl}(pid),\ pid \in \mathsf{pids}(s)\} \cup \mathsf{links}(nq)}{\langle s, node, nq\rangle \xrightarrow{\mathsf{die}(\textit{node})} [\![node, \mathsf{pids}(s), nlm]\!]}$$

Figure 8. Node-failure rule

$$\textit{node-start}\ \frac{}{[\![node, \textit{npl}, \{\}]\!] \xrightarrow{\tau} \langle \textit{init}() \parallel \{[\![pid, \{\}]\!] \mid pid \in \textit{npl}\}, node, \epsilon\rangle}$$

Figure 9. Start node rule

$$\textit{output}_{\textit{node}}\ \frac{s_1 \xrightarrow{\textit{pid}\ !_{\textit{from}}\textit{sig}} s_1' \qquad \mathsf{node}(\textit{from}) \neq \mathsf{node}(\textit{pid})}{\begin{array}{c}\langle s_1, \mathsf{node}(\textit{from}), nq_1\rangle \parallel \langle s_2, \mathsf{node}(\textit{pid}), nq_2\rangle \xrightarrow{\textit{pid}\ !_{\textit{from}}\textit{sig}} \\ \langle s_1', \mathsf{node}(\textit{from}), nq_1\rangle \parallel \langle s_2, \mathsf{node}(\textit{pid}), nq_2 \cdot (\textit{from},\textit{pid},\textit{sig})\rangle\end{array}}$$

$$\textit{output2}_{\textit{node}}\ \frac{s_1 \xrightarrow{\textit{pid}\ !_{\textit{from}}\mathsf{link}(\textit{from})} s_1' \qquad (pid, \textit{from}) \notin \textit{nlm}}{\begin{array}{c}\langle s_1, \mathsf{node}(\textit{from}), nq\rangle \parallel [\![\mathsf{node}(\textit{pid}), \textit{npl}, \textit{nlm}]\!] \xrightarrow{\textit{from}\ !_{pid}\mathsf{exited}(\textit{pid},\mathtt{noconnection})} \\ \langle s_1', \mathsf{node}(\textit{from}), nq \cdot (\textit{pid},\textit{from},\mathsf{exited}(\textit{pid},\mathtt{noconnection}))\rangle \parallel [\![\mathsf{node}(\textit{pid}), \textit{npl}, \textit{nlm}]\!]\end{array}}$$

$$\textit{output3}_{\textit{node}}\ \frac{s_1 \xrightarrow{\textit{pid}\ !_{\textit{from}}\textit{sig}} s_1' \qquad \begin{array}{c}(pid, \textit{from}) \in \textit{nlm}\ \vee \\ \neg\exists Pid' : \mathsf{erlangPid}.(sig = \mathsf{link}(Pid'))\end{array}}{\langle s_1, \mathsf{node}(\textit{from}), nq_1\rangle \parallel [\![\mathsf{node}(\textit{pid}), \textit{npl}, \textit{nlm}]\!] \xrightarrow{\tau} \langle s_1', [\mathsf{node}(\textit{from})], nq_1\rangle \parallel [\![\mathsf{node}(\textit{pid}), \textit{npl}, \textit{nlm}]\!]}$$

Figure 10. Intra-node communication – Output rules

$$\textit{input}_{\textit{node}}\ \frac{s \xrightarrow{\textit{pid}\ ?\textit{sig}} s' \qquad \mathsf{nmatch}(\textit{nq},\textit{from},\textit{pid}) = \textit{sig}}{\langle s, node, nq\rangle \xrightarrow{\textit{pid}\ ?_{\textit{from}}\textit{sig}} \langle s', node, nq \setminus (\textit{from},\textit{pid},\textit{sig})\rangle}$$

$$\textit{input2}_{\textit{node}}\ \frac{s \xrightarrow{\textit{pid}'?\mathsf{exited}(\textit{pid},\mathtt{noconnection})} s' \qquad (pid, pid') \in \textit{nlm}}{\langle s, \mathsf{node}(\textit{pid}'), nq\rangle \parallel [\![\mathsf{node}(\textit{pid}), \textit{npl}, \textit{nlm}]\!] \xrightarrow{\tau} \langle s', \mathsf{node}(\textit{pid}'), nq\rangle \parallel [\![\mathsf{node}(\textit{pid}), \textit{npl}, \textit{nlm} \setminus (pid, pid')]\!]}$$

$$\textit{input3}_{\textit{node}}\ \frac{(pid, pid') \in \textit{nlm}_1}{\begin{array}{c}[\![\mathsf{node}(\textit{pid}), \textit{npl}_1, \textit{nlm}_1]\!] \parallel [\![\mathsf{node}(\textit{pid}'), \textit{npl}_2, \textit{nlm}_2]\!] \xrightarrow{\tau} \\ [\![\mathsf{node}(\textit{pid}), \textit{npl}_1, \textit{nlm}_1 \setminus (pid, pid')]\!] \parallel [\![\mathsf{node}(\textit{pid}'), \textit{npl}_2, \textit{nlm}_2]\!]\end{array}}$$

$$\textit{silent}_{\textit{node}}\ \frac{s \xrightarrow{\tau} s'}{\langle s, node, nq\rangle \xrightarrow{\tau} \langle s', node, nq\rangle}$$

Figure 11. Intra-node communication – Input-rules

behind this is that as soon a process on another node has sent a link request, the sending process believes that it has a working link to the linked process. In order to maintain this image, we need to retreive these messages from the message queue. This is further discussed in Sect. 5.

In Fig. 9 a node is (re-)started. The interesting thing to notice here is that we create an erlangDeadProcess for each pid that has previously been running on the node. Because of this also future link-requests sent to these processes do get the correct respose, without having to state further rules in Fig. 11. Note especially that a node can only be restarted if the *plm*-list is empty, this is further discussed in Sect. 5.

In Fig. 10 the first rule, $\textit{output}_{\textit{node}}$ is the normal output rule, where a message is sent to a process on a live ERTS, the message is then appended to the node message queue *nq*. The message is later delivered by an input rule. The $\textit{output2}_{\textit{node}}$-rule generates an appropriate reply to a link-request made to a process on a dead node. A reply is only generated if there is no previous link present (i.e. a notification in *nlm* or a message waiting for delivery in nq_1) for that particular process identifier. The last output rule

$$\textit{interleave}_{\textit{node}}\ \frac{n_1 \xrightarrow{\gamma} n_1' \qquad \mathsf{pids}(n_1') \cap \mathsf{pids}(n_2) = \emptyset}{n_1 \parallel n_2 \xrightarrow{\gamma} n_1' \parallel n_2}$$

Figure 12. Node interleaving (symmetrical rule omitted)

$$\textit{spawn}_{0a}\ \frac{e \xrightarrow{\mathsf{spawn}(n,f,[v_1,\ldots,v_m]) \rightsquigarrow \{result,pid'\}} e' \qquad pid \neq pid' \qquad \mathsf{node}(pid) = n}{\langle e, pid, q, pl, b\rangle \xrightarrow{\tau} \langle e', pid, q, pl, b\rangle \parallel \langle f(v_1,\ldots,v_m), pid', \epsilon, \{pid\}, false\rangle}$$

$$\textit{spawn}_{0b}\ \frac{e \xrightarrow{\mathsf{spawn}(n,f,[v_1,\ldots,v_m]) \rightsquigarrow \{result,pid'\}} e' \qquad pid \neq pid' \qquad \mathsf{node}(pid) \neq n}{\langle\langle e, pid, q, pl, b\rangle \parallel s_1, \mathsf{node}(\textit{pid}), nq_1\rangle \parallel \langle s_2, n, nq_2\rangle \xrightarrow{\tau}}\ .$$
$$\langle\langle e', pid, q, pl, b\rangle \parallel s_1, \mathsf{node}(\textit{pid}), nq_1\rangle \parallel \langle\langle f(v_1,\ldots,v_m), pid', \epsilon, \{pid\}, false\rangle \parallel s_2, n, nq_2\rangle$$

Figure 13. New spawn-rules

output3$_{node}$ make sure that all other messages to processes on a dead node are ignored. In Fig. 11 there are three input-rules, which deliver messages from the node message queue and the notification-list respectively. In Fig. 11 there is also the *silent*-rule, applied for everything except communication happening at system/process level. Finally, note that messages sent between processes executing on the same node does not end up in the *node message queue*, since they are handled by the modified *com*-rule in Fig. 7.

In Fig. 11 we should note that the rule *input*$_{node}$ can be applied in an arbitrary order for a pair of sender and receiver. This means that messages can (possibly) be reordered. But at the same time this rule introduces another problem, namely that a certain (sender,receiver)-pair is never considered. In this situation the delivery of some messages are delayed infinitely, and therefore we have to state a fairness rule. We have more or less the same situation for the *input2*$_{node}$-rule, which does also require a fairness rule.

To be able to spawn new processes in the multi-node setting, we also need to slightly refine spawn (and similarily spawn_link). Refined spawn-rules are listed in Table 13.

We should also take a closer look at what happens with the node-to-node communication when the receiving process terminates. When a process terminates, its message queue q disappears. That is all messages which have already been delivered to the process are deleted. If the rule *input*$_{node}$ is applied for a terminated process, i.e. if we deliver a message to a terminated process, this is handled by the rules in Table 3.17 in Fredlunds semantics [13]. That is, the underlying semantics properly destroy messages and reply to link-requests.

4.5 Fairness

As we noted above, the input-rules, i.e. the rules in Fig. 11, can be applied in such a way that some messages are never delivered. That is 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 fairness-rules which will exclude certain unwanted behavior of the system. Fairness is defined in terms of permissable *execution sequences*.

Definition 20. An *execution sequence* is a sequence of Erlang Multi-node Systems n_i, together with corresponding Erlang Multi-node system actions γ_i written:

$$n_0 \xrightarrow{\gamma_0} n_1 \xrightarrow{\gamma_1} n_2 \xrightarrow{\gamma_2} \ldots.$$

Definition 21. [**Fairness for intra-node messages**]
It should hold for all execution sequences, $(\vec{n}, \vec{\gamma})$:

$$\forall i. \Big\{ \quad n_i \xrightarrow{pid!_{from} sig} n_{i+1} \Rightarrow$$
$$\exists j > i. \left(n_j \xrightarrow{pid?_{from} sig} n_{j+1} \ \vee \ n_j \xrightarrow{die(\mathsf{node}(pid))} n_{j+1} \right) \Big\}$$

That is, Definition 21 state that every sent message is eventually delivered or the node where the receiving process is executed dies.

Definition 22. [**Fairness for** `noconnection`**-messages**]
It should hold for all execution sequences, $(\vec{n}, \vec{\gamma})$:

$$\forall i. \left(n_i \xrightarrow{die(node)} n_{i+1} \Rightarrow \exists j \geq i. \left(n_j \xrightarrow{\gamma_j} [\![node, \textit{npl}, \{\}]\!] \right) \right)$$

That is, Definition 22 state that eventually the list of notifications to send is empty. Both fairness definitions are written in such a way that they can easily be expressed as LTL expressions.

4.6 Message reordering

The motivation for extending Fredlund's single-node semantics was to capture the distributed behavior where messages were reordered. Therefore, we conclude the presentation of the extended semantics with an example of such a reordering. The example is presented in Fig. 14, where we execute the program in Fig. 4 in the extended semantics. Here we use N_i as an abbreviation for the ERTS with node identifier n_i.

5. Discussion

The fundamental characteristics of Erlang is described by Armstrong in his thesis [1]. Armstrong describes how the concept of *concurrency oriented programming* led to the development of Erlang. The original thoughts on distribution are further described by Wikström [20]. In a concurrency oriented programming language the following is specified for *message passing*: "Message passing between a pair of processes is assumed to be ordered."

In Fredlund's presentation [13] it is stated that this is indeed true for the semantics, but due to the construction of the *com*-rule (in Fig. 3), even stronger properties hold in his semantics. In Fredlund's semantics the delivery of a message is instantaneous, meaning that **all** messages are delivered in exactly the order they are sent. Now, this is actually true for processes running on the same node,

Initial system:

$$\langle\langle \texttt{PidC} = \texttt{spawn}(n_2, \texttt{funC}, [\,]) \ \ldots, p_0, [\,]\rangle, n_0, [\,]\rangle$$

After spawns:

$$\underbrace{\begin{array}{c}\langle\langle \texttt{PidC ! hello} \ \ldots, p_1, [\,]\rangle, n_0, [\,]\rangle \parallel \\ \langle\langle \texttt{receive X} \rightarrow \texttt{PidC ! X end}, p_{11}, [\,]\rangle, n_1, [\,]\rangle \parallel \\ \langle\langle \texttt{receive X} \rightarrow \texttt{ok end}, p_{21}, [\,]\rangle, n_2, [\,]\rangle\end{array}}_{m_1}$$

Only p_1 at n_0 can make progress:

$$m_1 \xrightarrow{p_{21}!_{p_1}\mathsf{message}\textit{(hello)}} \underbrace{\begin{array}{l}\langle\langle \texttt{PidB ! world}, p_1, [\,]\rangle, n_0, [\,]\rangle \parallel N_1 \parallel \\ \langle\langle \texttt{receive X} \rightarrow \texttt{ok end}, p_{21}, [\,]\rangle, n_2, [(p_1, p_{21}, \texttt{hello})]\rangle\end{array}}_{m_2}$$

Since we are only aiming to show that messages can be reordered we now proceed by letting p_1 at n_0 send yet another message:

$$m_2 \xrightarrow{p_{11}!_{p_1}\mathsf{message}\textit{(world)}} \underbrace{\begin{array}{l}\langle[\![p_1, [\,]]\!], n_0, [\,]\rangle \parallel N_2 \parallel \\ \langle\langle \texttt{receive X} \rightarrow \texttt{PidC ! X end}, p_{11}, [\,]\rangle, n_1, [(p_1, p_{11}, \texttt{world})]\rangle\end{array}}_{m_3}$$

Next apply the *input*$_{node}$-rule for node n_1:

$$m_3 \xrightarrow{\mathrm{p}_{11}?_{p_1}\mathsf{message}\textit{(world)}} \underbrace{N_0 \parallel \langle\langle \text{receive X} \rightarrow \texttt{PidC ! X end}, p_{11}, [\texttt{world}]\rangle, n_1, [\,]\rangle \parallel N_2}_{m_4}$$

Proceed by letting p_{11} at n_1 read `world` from its queue and send this on to p_{21} at n_2:

$$m_4 \xrightarrow{p_{21}!_{p_{11}}\mathsf{message}\textit{(world)}} \underbrace{\begin{array}{l}N_0 \parallel \langle[\![p_{11}, [\,]]\!], n_1, [\,]\rangle \parallel \\ \langle\langle \text{receive X} \rightarrow \texttt{ok end}, p_{21}, [\,]\rangle, n_2, \\ [(p_1, p_{21}, \texttt{hello}), (p_{11}, p_{21}, \texttt{world})]\rangle\end{array}}_{m_5}$$

Now to acheive a message reordering we can apply the *input*$_{node}$-rule for n_2 and let nmatch return the `world`-message:

$$m_5 \xrightarrow{p_{21}?_{p_{11}}\mathsf{message}\textit{(world)}} \begin{array}{l}N_0 \parallel N_1 \parallel \\ \langle\langle \text{receive X} \rightarrow \texttt{ok end}, p_{21}, [\texttt{world}]\rangle, n_2, \\ [(p_1, p_{21}, \texttt{hello})]\rangle\end{array}$$

And clearly we have shown a possible execution sequence where the *world*-message arrives before the *hello*-message.

Figure 14. Execution sequence in **extended** semantics

due to how the Erlang runtime system is implemented. It is, however, not true in general for a concurrency oriented programming language, and specifically not in a distributed setting with several different Erlang nodes.

Nevertheless, since Fredlund's single-node semantics faithfully describes what actually happens inside the Erlang runtime system, we argue that it is a good thing to keep the underlying semantics as it is. Fredlund's intra-node message passing is not only more faithful, but also simpler than our inter-node message passing. Thus, using a special version of local message passing makes certain (local) systems easier to reason about. An alternative is to only use the kind of message passing rules that we have in the node-to-node communication. The consequence of this is an overall simpler semantics, which would be less restrictive for a local system. However, this could be problematic in a model checking context, since it might result in a bigger state space. Another problem is the introduction of false negatives, because a local system might fail due to an order of event not possible in reality.

When developing the extended semantics, we have made several distinctive choices. In some places it is not entirely clear how the Erlang system is working or even how it is supposed to work, and in such places we have made some simplifying assumptions. Perhaps the most interesting such place is in the *node-start*-rule (Fig. 9) where we allow a node to restart only if the set of notifications-to-be-sent (*nlm*) is empty. In Erlang, these notifications are handled inside the runtime system and since we do not model the runtime system it is impossible to capture the full behavior. Instead, since a node restart is regarded as a *slow* event it seems reasonable to do the approximation that a node can not be restarted before all notifications are sent. This also make matters simple when we are defining *fairness* for the notification messages. So in all this is an approximation of the real world, but a very reasonable approximation. Another choice we made was to introduce one message queue for 'messages in transit' per node. There is no functional motivation behind this choice, we could just as well have settled for one single global message queue, but in the end we thought it to be more aesthetic to have one queue per node.

Quite many of the rules presented in Section 4 handle the link-messages. The link mechanism is a very useful construction and many distributed implementations rely on this functionality. The Erlang concept of *monitors* can be implemented in terms of links. It is important to observe that we must treat links differently from ordinary messages in order to faithfully describe Erlang programs. For example, take a look at the Erlang program in Fig. 15. If we run `procA` it should be possible to sometimes trap the exit message (i.e. get an `{'EXIT',pid,kill}` from `procB` and sometimes just get a `{'EXIT',pid,noproc}` back, indicating that process B had already terminated. This behavior can be observed by running the program repeatedly. Although the result is heavily dependent on machine load and network load, with 1000 runs, almost everytime

both behaviors could be observed. This means that it would be incorrect to treat the link-message as ordinary message, since the message order between a pair of processes is respected and then an order of events such as getting {'EXIT',pid,kill} from process B would be impossible.

In Fredlund's single-node semantics, (and here seen in Fig. 2) a separation is made between link-messages and other messages, which ensures the correct behavior. However, when dead nodes are involved, some special care is needed, which results in special link-rules as seen in Fig. 10 and Fig. 11.

Another part of the linking mechanism is the somewhat complicated *node-failure* rule (Fig. 8), where we have to collect link messages from the node message queue. This is because we are modelling the link mechanism in a different way from the actual Erlang implementation. In the Erlang implementation, the run-time system keeps track of links via a timeout construction. Here instead, we do the book keeping (so to say) at the other end. Therefore, we have to take extra care when a node fails since the messages in nq are otherwise lost.

```
procA() ->
    PidB = spawn(?ANOTHERNODE,?MODULE,procB,[]),
    PidB ! a,
    process_flag(trap_exit,true),
    link(PidB),
    ...

procB() ->
    receive a ->
    exit(kill)
    end.
```

Figure 15. Erlang program - Linking

6. Related Work

The semantics for Erlang is informally described in [2]. A first, not completed, attempt to formally specify the semantics of Erlang was made by Petterson [17]. Petterson used Natural Semantics, and was inspired by similar work with Standard ML and Relational ML. Following this, the Formal Design Techniques group at the Swedish Institute of Computing Science (SICS) developed a number of formal (operational) semantics for different subsets of Erlang, for example [10] and [11]. These attempts, compared to the semantics presented by Fredlund in [13], are not as direct and lacks the clear separation between the functional and the concurrent part of the semantics. A completely different approach is taken by Huch in [16]. Huch present a semantics for (a smaller subset) of Erlang, which is more direct and relies heavily on contextual information. All these approaches except Petterson's consider systems which are not fully distributed since they do not deal with nodes.

Both [10] and [16] make use of subsets of Erlang referred to as core fragments of Erlang. These references should not be confused with the Core Erlang project [9], which defines a complete (with respect to representing all possible Erlang programs) core fragment of Erlang. Core Erlang is in the Erlang compiler used as the intermediate format where optimizations and transformations are applied, therefore its use is mostly syntactic. For Core Erlang the semantics is given in a structured but also informal way, and does not directly speak about nodes or message delivery.

7. Conclusions and Future Work

After Fredlund proposed his single-node semantics, it was at least thought "morally OK" to use this semantics to reason about and model distributed systems. However, messages actually can and do arrive in different orders in a distributed setting as compared to a local setting. A simple experiment involving 3 nodes already shows this, even when the nodes are implemented as 3 run-time systems running on the same workstation! Moreover, we discovered that the above was not merely a theoretical anomaly, but an actual problem with a real-life implementation.

Along the same lines, many Erlang developers think it morally OK to test their distributed system on a single node. For the same reasons as mentioned above, errors might slip through.

We have augmented Fredlund's single-node semantics with another top-level layer describing nodes. We claim that our semantics is intuitive and models the actual behaviour of message passing between nodes.

Future Work We plan to use our semantics as a basis for a translation from Erlang into a process calculus, with the goal of being able to automatically model check Erlang systems.

Acknowledgments

We thank Thomas Arts for his valuable comments on earlier versions of this paper.

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] J. Armstrong, M. Williams, C. Wikstrom, and R. Virding. *Concurrent Programming in Erlang*. Prentice-Hall, Englewood Cliffs, New Jersey, USA, second edition, 1996.

[3] T. Arts, C. Benac Earle, and J. Derrick. Development of a verified Erlang program for resource locking. *Int. J. on Software Tools for Technology Transfer*, 5(2-3):205–220, 2004.

[4] T. Arts, C. Benac Earle, and J. J. Sánchez Penas. Translating erlang to mcrl. In *Fourth International Conference on Application of Concurrency to System Design*, pages 135–144, Hamilton (Ontario), Canada, June 2004. IEEE computer society.

[5] T. Arts, K. Claessen, and H. Svensson. Semi-formal development of a fault-tolerant leader election protocol in Erlang. In *Lecture Notes in Computer Science*, volume 3395, pages 140 – 154. Springer, Feb 2005.

[6] T. Arts and L.-Å. Fredlund. Trace analysis of Erlang programs. In *Proceedings of the 2002 ACM SIGPLAN workshop on Erlang*, pages 16–23. ACM Press, 2002.

[7] T. Arts and J. Hughes. Erlang/quickcheck. In *Ninth International Erlang/OTP User Conference*, Nov. 2003.

[8] J. Blom and B. Jonsson. Automated test generation for industrial Erlang applications. In *ERLANG '03: Proceedings of the 2003 ACM SIGPLAN workshop on Erlang*, pages 8–14, New York, NY, USA, 2003. ACM Press.

[9] R. Carlsson, B. Gustavsson, E. Johansson, T. Lindgren, S. Nystrm, M. Petterson, and R. Virding. Core Erlang 1.0 language specification. Technical Report 2000-30, Department of Information Technology, Uppsala University, November 2000.

[10] M. Dam and L.-Å. Fredlund. On the verification of open distributed systems. In *SAC '98: Proceedings of the 1998 ACM symposium on Applied Computing*, pages 532–540, New York, NY, USA, 1998. ACM Press.

[11] M. Dam, L.-Å. Fredlund, and D. Gurov. Toward parametric verification of open distributed systems. In *COMPOS'97: Revised Lectures from the International Symposium on Compositionality: The Significant Difference*, pages 150–185, London, UK, 1998. Springer-Verlag.

[12] L.-Å. Fredlund. Towards a semantics for Erlang. In *Foundations of Mobile Computation: A Post-Conference Satellite Workshop of FST & TCS 99*, Institute of Mathematical Sciences, Chennai, India, Dec 1999.

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

[14] L.-Å. Fredlund, D. Gurov, and T. Noll. Semi-automated verification of Erlang code. In *ASE '01: Proceedings of the 16th IEEE International Conference on Automated Software Engineering*, page 319, Washington, DC, USA, 2001. IEEE Computer Society.

[15] L.-Å. Fredlund, D. Gurov, T. Noll, M. Dam, T. Arts, and G. Chugunov. A verification tool for ERLANG. *International Journal on Software Tools for Technology Transfer (STTT)*, 4(4):405 – 420, Aug 2003.

[16] F. Huch. Verification of Erlang programs using abstract interpretation and model checking. In *ICFP '99: Proceedings of the fourth ACM SIGPLAN international conference on Functional programming*, pages 261–272, New York, NY, USA, 1999. ACM Press.

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

[18] M. Widera. Flow graphs for testing sequential erlang programs. In *ERLANG '04: Proceedings of the 2004 ACM SIGPLAN workshop on Erlang*, pages 48–53, New York, NY, USA, 2004. ACM Press.

[19] U. Wiger. Fault tolerant leader election. http://www.erlang.org/.

[20] C. Wikstrom. Distributed programming in Erlang. In *PASCO'94, First International Symposium on Parallel Symbolic Computation*, Linz, Austria, Dec 1994.