A More Accurate Semantics for Distributed Erlang

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

Lars-Åke Fredlund *
Facultad de Informática, Universidad Politécnica de Madrid, Spain
fred@babel.ls.fi.upm.es

Abstract

In order to formally reason about distributed Erlang systems, it is necessary to have a formal semantics. In a previous paper we have proposed such a semantics for distributed Erlang. However, recent work with a model checker for Erlang revealed that the previous attempt was not good enough. In this paper we present a more accurate semantics for distributed Erlang. The more accurate semantics includes several modifications and additions to the semantics for distributed Erlang proposed by Claessen and Svensson in 2005, which in turn is an extension to Fredlund's formal single-node semantics for Erlang. The most distinct addition to the previous semantics is the possibility to correctly model disconnected nodes.

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

General Terms Languages, Theory, Verification

Keywords Erlang, semantics, distributed systems, verification, model checking

1. Introduction

Software systems written in Erlang are often running in a distributed environment, and are often highly concurrent and dynamic in nature. Something that has lately become even more emphasised by the introduction of multi-core and SMP[1] computers. And although Erlang with its *Concurrency Oriented Programming* paradigm is particularly suited for writing such applications, experience still shows that concurrent and fault-tolerant software is hard to write, test and verify. Because of this, several approaches have been proposed for testing [6, 7, 9, 21, 5] Erlang programs and also a lot of work has been done on formal verification of Erlang programs [19, 18, 3, 16].

Since the history of Erlang starts in industry, and not in a university, Erlang is very much defined in terms of its implementation. However, when working with formal verification a formal semantics is almost indispensable, and doing verification without a formal semantics is really hard. In 1999 Fredlund proposed a formal semantics for Erlang [14], and the semantics is described in detail in [15]. Fredlund's semantics is a small-step operational semantics. It is constructed in a simple, easy to understand layered fashion. The semantics has been used as a basis in several different verification projects, such as semi-formal verification of Erlang code [17] and model checking a resource manager [18]. Fredlund's semantics has also been a basis for the development of a theorem prover [18] and a translation of Erlang into a language that can be model checked [4].

In 2004 two previously unknown errors was discovered in an open source implementation in Erlang of a leader election algorithm [22, 5]. It turned out that both errors were caused by difficult to foresee chains of events related to the arrival order of messages in the distributed environment. It also turned out that the errors were specific to a *multi-node* setting. That is, the errors could only be found when different parts of the system run on different nodes. Thus, contrary to the Erlang idea that distribution should be transparent, there is a real behavioral difference between single-node and multi-node systems.

During the analysis of the errors in the leader election implementation, we realized that it is impossible to reason about this type of errors in Fredlund's semantics. The semantics does not contain the concept of nodes, and all processes are localized at the same run-time system. This also means that it is impossible to detect this kind of multi-node errors in a model checker based on Fredlund's single-node semantics.

Therefore we proposed an extension of the semantics into a distributed (multi-node) semantics for Erlang [11]. In that paper we added a distributed layer on top of the single node semantics, and were able to successfully model multi-node programs in the extended semantics. In the paper we also warned the community of potential pit-falls with testing and verification using a single-node semantics.

The semantics for distributed Erlang have since been used in the implementation of a model checker for Erlang (McErlang, [16]). From the work with the model checker we could observe that the proposed multi-node semantics was incomplete. (With incomplete we mean that there are possible behaviors in the Erlang run-time system, which can not be described by the semantics.) The incompleteness stems mainly from the Erlang behavior in the case of *node disconnect*. Two Erlang nodes can become disconnected from each other if the link between them fails, when this happens both involved nodes regard the other node as dead. This does have some interesting consequences when the nodes later re-connect. Such behavior can not be modeled in the multi-node semantics we proposed earlier. Since the publication of the semantics for distributed Erlang we have also discovered a few minor errors in the semantics as well as some rather embarrassing inelegancies in the presentation of the semantic rules. Therefore we have re-structured

* The author was supported by a Ramón y Cajal grant from the Spanish Ministerio de Educación y Ciencia, and the DESAFIOS (TIN2006-15660-C02-02) and PROMESAS (S-0505/TIC/0407) projects.

[1] Symmetric Multiprocessing

Erlang'07, October 5, 2007, Freiburg, Germany.

and extended the distributed layer and now propose a more accurate multi-node semantics.

In order to make this paper self contained and the presentation as easy as possible to follow, some material from the previous paper by Claessen and Svensson ([11]) is presented here as well. This especially concerns Sect. 2 with the single-node semantics, which naturally has not changed, and Sect. 3 where the motivation for the multi-node semantics is still at least partly the same.

Contributions The contribution of the paper is a clear and self contained presentation of a more accurate distributed (multi-node) semantics for Erlang. In addition to previous attempts to present a semantics for distributed Erlang, this presentation includes together with some other modifications and additions novel semantic rules to correctly express node disconnects. We also present and informally argue for desirable properties of the proposed semantics. The multi-node semantics has already proved to be useful, when used as the basis for a model checker for Erlang (McErlang).

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 single-node semantics in order to introduce the full distributed behavior. In Sect. 5 we specify desirable properties of the multi-node semantics, and argue why they are fulfilled by the presented semantics. Some of the design decisions in the extended semantics are further discussed in Sect. 6, related approaches to the problem are described in Sect. 7 and finally, we conclude in Sect. 8.

2. Original semantics

In [15], 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 also not included.

All definitions and rules presented in this section are taken from Fredlund's presentation of the semantics [15], 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 [15], 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.

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

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

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

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

Figure 1. Expression evaluation rules

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

$\gamma ::=$	τ	computation step
$\mid$	$\textit{pid}\,!\,v$	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 [15].

In Fig. 1 we have listed Fredlund's rules for evaluation of send and receive at the expression level. 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 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 a queue data structure, in theory unbound, thus it can store any number of messages. Mailboxes are ranged over by $q \in \mathsf{erlangQueue}$, and ϵ denotes the empty queue.

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 (erlangDeadProcess $\subset$ erlangProcess) is a tuple:
erlangPid $\times$ $\mathcal{P}$ (erlangPid $\times$ 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 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 $sig \in$ erlangSignal are:

$$\begin{array}{rcll} sig ::= & \mathsf{message}(v) & \text{message} \\ | & \mathsf{link}(pid) & \text{linking with process} \\ | & \mathsf{unlink}(pid) & \text{unlinking process} \\ | & \ldots & \end{array}$$

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

$$\begin{array}{rcll} \alpha ::= & \tau & \text{silent action} \\ | & pid\,!\,sig & \text{output action} \\ | & pid\,?\,sig & \text{input action} \end{array}$$

Definition 10 The system transition relation
$\rightarrow$: erlangSystem $\times$ erlangSysAction $\times$ erlangSystem, written $s_1 \xrightarrow{\alpha} s_2$, is the least relation satisfying the transition rules in [15]. 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*$'$!message(v)) system action. The rules 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. The function pids used in the *interleave*-rule, simply returns all process identifiers in the Erlang system. 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

The motivation for creating a multi-node semantics for Erlang comes from observations made during research projects. There were some cases when we did not understand the behavior of our Erlang programs and other cases when we were just curious about how the run-time system is implemented. When we had figured out how things actually worked, we realised that the existing single-node semantics was not expressive enough to describe the problematic situations. Below we describe two motivating examples, where we have quite ordinary situations in which the single-node semantics is not expressive enough.

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

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

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

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

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

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

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

3.1 Message reordering

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

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

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

Figure 4. Erlang program - Message reordering

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. This understanding of possible message orderings is further confirmed in the natural language semantics for Erlang (draft) by Barklund and Virding [8] (Sect. 10.5.4): *'It is assured (through the rules of signals, cf. §10.6.2) that if a process P_1 dispatches two messages M_1 and M_2 to the same process P_2, in that order, then message M_1 will never arrive after M_2 at the message queue of P_2. Note that this does not guarantee anything about in which order messages arrive when a process sends messages to two different processes...'*

The possible executions are depicted in Fig. 5.

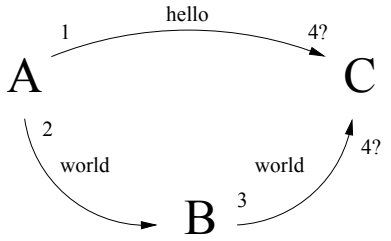

Figure 5. Message passing

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

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

The results were somewhat surprising. If the execution would follow the intuition, *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 and in situations (2) and (3) we could see both *hello* and *world* arriving first. The conclusion is that the Erlang run-time system implementation behaves differently in a local setting as compared to in a distributed setting (and also differently from the Erlang specifications, this is further disscussed in Sect. 6). This partly explain why errors such as those found in the leader election implementation [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 terms 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 the single-node semantics does not include nodes, 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 be able to describe also the behavior in situations (2) and (3) serves as the motivation for extending Fredlund's single-node semantics to be able to fully reason about distributed Erlang systems. This is especially important in case we use the semantics

1. Initial system:
   $P_0 = \langle \texttt{PidC} = \texttt{spawn}(\texttt{procC}, []) \ \ldots, p_0, \epsilon \rangle$
2. The only scheduler option is to spawn `procC`. After that, the only option is to spawn `procB` since the `receive` in `procC` (P_2) blocks. This results in three processes:
   $P_0 = \langle \texttt{PidC} \,!\, \texttt{hello} \ldots, p_0, \epsilon \rangle$
   $P_1 = \langle \texttt{receive}\ X \rightarrow \texttt{PidC} \,!\, X\ \texttt{end}, p_1, \epsilon \rangle$
   $P_2 = \langle \texttt{receive}\ X \rightarrow \texttt{ok}\ \texttt{end}, p_2, \epsilon \rangle$
3. Only P_0 can make progress, since P_1 and P_2 are blocked on a `recieve` statement:
   $P_0 = \langle \texttt{PidB} \,!\, \texttt{world}, p_0, \epsilon \rangle$
   $P_1 = \langle \texttt{receive}\ X \rightarrow \texttt{PidC} \,!\, X\ \texttt{end}, p_1, \epsilon \rangle$
   $P_2 = \langle \texttt{receive}\ X \rightarrow \texttt{ok}\ \texttt{end}, p_2, \epsilon \cdot \texttt{hello} \rangle$
4. Here we see that there is no way that `procC` can receive `world` before `hello`. This is because we have a 'match all' pattern (the single unbound variable `X`) in `procC` and any later arriving message is put last in the mailbox. Therefore, the next application of the receive-rule (Fig. 1) must read `hello` from the mailbox.

Figure 6. Hello World - Single-Node Execution

to produce a model, if certain situations are not present in the model, errors may be overlooked, and thus giving false confidence.

3.2 Disconnected nodes

Another interesting and potentially dangerous behavior of the Erlang run-time system occurs when *nodes disconnect* from each other. In the simple situation two nodes (it could be generalized to many nodes) become disconnected because one of them dies. This of course means that all processes on the failing node dies, and processes on the surviving node will be notified of this via the link-mechanism. This situation is not dangerous, and it could be simulated in the single-node semantics by grouping processes together at a meta level and then kill off a whole group of processes.

The potentially dangerous situation is when two processes become disconnected because the link (in the ordinary case, the network connection) between them breaks down. In that case both of the nodes continue to execute, and both nodes consider the other node to have failed(!). Thus the processes are informed of the failure of processes on the other node via the link-mechanism. Things then become really interesting when the nodes *re-connect*, because this happens without any notice to the running processes. This should be considered harmful since messages can be dropped silently (if the link mechanism is not used), which breaks the common assumptions about TCP/IP like communication in Erlang.

From a programmer's point of view this requires some extra caution, and as long as this behavior is taken into consideration it should not cause too much trouble. However if one is careless, and for example has a system with two processes running on different nodes (A and B), where A sends a stream of messages to B and only occasionally gets a reply from B. Then if neither A or B uses the link mechanism it could be the case that A sends a lot of messages that B never receives because the nodes are disconnected, and later the nodes are re-connected before A expects an answer. This behavior obviously can not be described in the single-node semantics.

It should be noted that this phenomenon is also acknowledged in Barklund and Virdings natural language semantics for Erlang [8] (Sect. 10.6.2): *'It is guaranteed that if a process P_1 dispatches two signals s_1 and s_2 to the same process P_2, in that order, then signal s_1 will never arrive after s_2 at P_2. It is ensured that whenever possible, a signal dispatched to a process should eventually arrive at it. There are situations when it is not reasonable to require that all signals arrive at their destination, in particular when a signal is*

sent to a process on a different node and communication between the nodes is temporarily lost.'. In this context a message is a signal. That is, there are no promises regarding safe delivery (except no reordering), especially during temporary communication failures.

4. Distributed (Multi-Node) Semantics

In this section Fredlund's single-node semantics is extended, by adding a new layer of semantic rules, to a distributed (multi-node) semantics. By adding another layer on top of the existing semantics we can deal with all aspects of distribution without making more than a few marginal changes to the single-node semantics. One important implication of this is that everything that is defined in terms of the single-node semantics is still valid in the distributed semantics under the restriction that the system is local, i.e. running on the same node.

The distributed layer of the semantics is presented in three steps; 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 Run-Time systems*, i.e. a single node, and also *Erlang Multi-node systems* which are collections of nodes forming complete distributed systems. We also need to make some minor changes to the single-node semantics. Secondly, we need new rules for communication between processes on different nodes (i.e. different run-time systems). These communication rules should have the properties described in Sect. 3, and thus enable certain message reordering as well as introduce the possibility to drop messages in the case of node disconnect. Thirdly, we add the concept of nodes that die and get restarted. We also need to extend the linking mechanism in order for it to work also in the distributed semantics.

4.1 Nodes

Before we can define semantic rules for multi-node Erlang systems we have to introduce *node identifiers*. The node identifier could be any unique identifier. For the sake of simplicity, we can assume that they are integers. We also need two functions that returns node identifiers:

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

4.2 Node message queues

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

Definition 12 An *Erlang node message queue*, ranged over by $nq \in$ erlangNodeQueue, consists of a finite sequence of triplets $v_x = (\mathit{from}_x, \mathit{to}_x, \mathit{sig}_x)$: $v_1 \cdot v_2 \cdot \ldots \cdot v_n$, where ϵ is the empty sequence, $(\cdot)$ is concatenation and $(\setminus)$ is deletion of the first matching triplet, e.g.

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

4.3 Run-Time systems

Next, we define the concepts of live and dead run-time systems.

Definition 13 A **live** *Erlang Run-Time system (ERTS)*, ranged over by $r \in$ erlangRuntimeSystem is a triplet: erlangSystem $\times$ erlangNodeName $\times$ erlangNodeQueue, written [s, node, nq], where:

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

Definition 14 A **dead** *Erlang Run-Time system* is a tuple: erlangNodeName $\times$ $\mathcal{P}$(erlangPid), written $[\![node, npl]\!]$, where:

- *node* is the node identifier.
- *npl* is a set of process identifiers (consisting of all processes on the node *node*).

An example of a dead ERTS is: $[\![n_1, \{p_1, p_5, p_{13}\}]\!]$ where the node identifier is n_1, and the processes that has executed on n_1 are p_1, p_5 and p_{13}. Note also that 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 ($\|$) 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.

4.4 Changes to the single-node semantics

Everything defined in the original semantics works in the extended semantics, with a few exceptions. We have to change the *com*-rule (Fig. 3) slightly, so that it only applies in the situation where both processes are running on the same node. Further, we need to do a small (non-functional) modification in order to export extra information about the sender of a message to the layer we are adding. This is done by replacing the sending operator (!) with a tagged version ($!_{\mathit{from}}$), where *from* is the sender of the message. This change is straightforward and is applied to all the send operators in the single-node semantics. An example of the tagged send operator can be seen in the new *com*-rule presented in Fig. 7. The new *com*-rule has an added side condition, which restricts its application to the case when the sender and the receiver are running in the same Erlang system. Finally, to be able to spawn new processes in the multi-node setting, we need to refine the existing spawn-rule (we add a side condition, assuring that it is a local spawn) and add a distributed spawn-rule. (And similar changes for spawn_link.) The new ($spawn_{dist}$) and modified ($spawn_{local}$) spawn-rules are presented in Fig. 8. We should note that the distributed spawn rule is atomic, that is, the new process is created (but does not necessarily start executing) immediately at the remote node.

In addition to these changes, we also introduce a new way of writing an erlangDeadProcess (previously written $\langle \mathit{pid}, \mathit{plm} \rangle$), namely: $\langle\!\langle \mathit{pid}, \mathit{plm} \rangle\!\rangle$.

4.5 Transitions

Multi-node systems transitions are labeled 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:

$\gamma ::=$	τ	silent action
$\mid$	$pid!_{\mathit{from}} sig$	output action
$\mid$	$pid?_{\mathit{from}} sig$	input action
$\mid$	die$(node)$	node failure
$\mid$	disconnect$(node1, node2)$	node disconnection

That is, the actions visible in erlangMultiNodeSysAction are only the node-to-node communication and node failure. Messages

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

Figure 7. New com-rule

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

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

Figure 8. New spawn-rules (Symmetric spawn_link-rules omitted)

sent between processes executing on the same node are not visible at this level. Note also that at this level the input actions (?) are tagged with a *from*. This is not strictly necessary from a functionality point of view, but (as we see in Def. 22) *fairness* can be expressed in a simple and elegant way with tagged input actions.

Definition 17 The transistion relation for Erlang Multi-node systems, $\rightarrow$:erlangMultiNodeSystem $\times$ erlangMultiNodeSysAction $\times$ erlangMultiNodeSystem, written $n_1 \xrightarrow{\gamma} n_2$, is the least relation satisfying the rules in Fig. 9 – Fig. 14.

4.6 Operational output rules

In Fig. 9 the first rule, *output*$_{node}$ is the normal output rule. In this rule a message is sent to a process executing in a live ERTS, the message is appended to the node message queue *nq*. The message is later delivered to the receiving process by an input rule. The *output2*$_{node}$-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 not already a reply in the node queue of the sender (i.e. a message waiting for delivery in nq) for that particular process identifier. The reason to look inside nq for an exit-message is that as long as the error-message has not reached the linking process, it can not act upon the error. Therefore it can not know that it should establish a new link, and thus no new error-message should be constructed. The last output rule *output3*$_{node}$ take care of all other messages to a processes on a dead node, and simply discards them.

4.7 Operational input rules

In Fig. 10 there are two input-rules. The input rule *input*$_{node}$ uses the function nqMatch to retreive a message from the node message queue in a non-deterministic fashion. The selected message (sig) is then delivered to the actual receiving process. The nqMatch function is defined below. In Fig. 10 there is also the *silent*$_{node}$-rule, applied for everything except communication happening at system/process level. As with the output rules, note that messages sent between processes executing on the same node does not end up in the *node message queue*. They are handled by the modified *com*-rule (Fig. 7).

Definition 18 nqMatch(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.

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

In Fig. 10 we should note that the rule *input*$_{node}$ can be applied in an arbitrary order for pairs of a sender and a receiver. This means that messages can (possibly) be reordered. However, at the same time this rule introduces another problem, namely that a certain (sender,receiver)-pair is never considered. That means that the delivery of some messages could potentially be delayed forever. The problem is that 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. 4.9).

4.8 Operational node rules

There are four operational node rules, *node-failure*, *node-(re)start*, *node-disconnect* and *node-interleave*. The node-failure rule is presented in Fig. 11. It looks quite complicated, but most of the complexity is due to bookkeeping. When a node fail, and because we do not have any external handling of links, we have to collect all links that are currently defined and produce proper exit-messages. To simplify the collection of links, we use two functions links and getProcLinks (defined below) to collect the links and a third function deliverMsgsToNq (also defined below) to deliver the exit-messages. The node-failure rule also create a *dead ERTS* with the same node name as the failing node. The dead node also contains a list of all processes previously running on the node. One thing to note here is that there can be at most one link between a pair of processes, and therefore we can safely add all link messages directly to the node queues without worrying about message order. Another thing that we should observe is that links are collected both from the individual processes and the node message queue *nq*. The intuition behind this is that as soon as a process on another node has sent a link request, the sending process believes that it has a working link to the linked process. The linking mechanism is further discussed in Sect. 6.

Definition 19 The function links(erlangNodeQueue) traverses an Erlang node message queue and collects all pending link-request from the node queue. The function getProcLinks(erlangPid) return the *pl*-list (i.e. the list of linked nodes) for a process, given the process identifier of that process. Finally, the function deliverMsgsToNq(EMNS,Messages) deliver all messages to the correct node queue. E.g. let $n = [s_{11}, n_1, nq_1] \parallel [s_{21}, n_2, nq_2]$ and $p_1, p_3 \in \mathsf{pids}(s_{11})$ and $p_7 \in \mathsf{pids}(s_{21})$, then:
$\mathsf{deliverMsgsToNq}(n, \{(p_7, p_1, sig_1), (p_3, p_7, sig_2)\}) =$
$[s_{11}, n_1, nq_1 \cdot (p_7, p_1, sig_1)] \parallel [s_{21}, n_2, nq_2 \cdot (p_3, p_7, sig_2)]$

In Fig. 12 the node-(re)start rule is presented. The interesting thing to notice here is that we create an erlangDeadProcess for each

$$\textit{output}_{\textit{node}}\ \frac{s_1 \xrightarrow{\textit{pid}\ !_{\textit{from}}\textit{sig}} s_1' \qquad \mathsf{node}(\textit{from}) \neq \mathsf{node}(\textit{pid})}{[s_1, \mathsf{node}(\textit{from}), nq_1] \parallel [s_2, \mathsf{node}(\textit{pid}), nq_2] \xrightarrow{\textit{pid}\ !_{\textit{from}}\textit{sig}} [s_1', \mathsf{node}(\textit{from}), nq_1] \parallel [s_2, \mathsf{node}(\textit{pid}), nq_2 \cdot (\textit{from,pid,sig})]}$$

$$\textit{output2}_{\textit{node}}\ \frac{s_1 \xrightarrow{\textit{pid}\ !_{\textit{from}}\mathsf{link}(\textit{from})} s_1' \qquad (\textit{pid,from,}\mathsf{exited}(\textit{pid,}\texttt{noconnection})) \notin nq}{[s_1, \mathsf{node}(\textit{from}), nq] \parallel [\![\mathsf{node}(\textit{pid}), npl]\!] \xrightarrow{\textit{from}\ !_{pid}\mathsf{exited}(\textit{pid,}\texttt{noconnection})} [s_1', \mathsf{node}(\textit{from}), nq \cdot (\textit{pid,from,}\mathsf{exited}(\textit{pid,}\texttt{noconnection}))] \parallel [\![\mathsf{node}(\textit{pid}), npl]\!]}$$

$$\textit{output3}_{\textit{node}}\ \frac{s_1 \xrightarrow{\textit{pid}\ !_{\textit{from}}\textit{sig}} s_1' \qquad \begin{array}{c} sig \neq \mathsf{link}(\textit{from}) \ \vee \\ (\textit{pid,from,}\mathsf{exited}(\textit{pid,}\texttt{noconnection})) \in nq_1 \end{array}}{[s_1, \mathsf{node}(\textit{from}), nq_1] \parallel [\![\mathsf{node}(\textit{pid}), npl]\!] \xrightarrow{\tau} [s_1', \mathsf{node}(\textit{from}), nq_1] \parallel [\![\mathsf{node}(\textit{pid}), npl]\!]}$$

Figure 9. Inter-node communication – Output rules

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

$$\textit{silent}_{\textit{node}}\ \frac{s \xrightarrow{\tau} s'}{[s, node, nq] \xrightarrow{\tau} [s', node, nq]}$$

Figure 10. Inter-node communication – Input-rules

$$\textit{links} = \{(pid, pid') \mid pid' \in \mathsf{getProcLinks}(pid),\ pid \in \mathsf{pids}(s)\} \cup \mathsf{links}(nq)$$
$$\textit{linkMsgs} = \{(\textit{from}, \textit{to}, \mathsf{exited}(\textit{from,}\texttt{noconnection}) \mid (\textit{from,to}) \in \textit{links}\}$$
$$\textit{node-failure}\ \frac{n' = \mathsf{deliverMsgsToNq}(\textit{n,linkMsgs})}{n \parallel [s, node, nq] \xrightarrow{\mathsf{die}(\textit{node})} n' \parallel [\![node, \mathsf{pids}(s)]\!]}$$

Figure 11. Node-failure rule

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

Figure 12. Start node rule

$$msg_{1\leftarrow 2} = \{(from, to, sig) \mid sig \in \mathsf{nqMatchAll}(nq, to, from),\ to \in \mathsf{pids}(s_1),\ from \in \mathsf{pids}(s_2)\}$$
$$msg_{2\leftarrow 1} = \{(from, to, sig) \mid sig \in \mathsf{nqMatchAll}(nq, to, from),\ to \in \mathsf{pids}(s_2),\ from \in \mathsf{pids}(s_1)\}$$
$$links_1 = \mathsf{filterLinks}(\{(pid_1, pid_2) \mid pid_2 \in \mathsf{getProcLinks}(pid_1),\ pid_1 \in \mathsf{pids}(s_1)\} \cup \mathsf{links}(nq_1), n_1, n_2)$$
$$links_2 = \mathsf{filterLinks}(\{(pid_1, pid_2) \mid pid_2 \in \mathsf{getProcLinks}(pid_1),\ pid_1 \in \mathsf{pids}(s_2)\} \cup \mathsf{links}(nq_2), n_2, n_1)$$
$$fail_1 = \{(\textit{from,to,}\mathsf{exited}(\textit{from,}\texttt{noconnection})) \mid (from, to) \in links_1\}$$
$$fail_2 = \{(\textit{from,to,}\mathsf{exited}(\textit{from,}\texttt{noconnection})) \mid (to, from) \in links_2\}$$
$$\textit{node-disc}\ \frac{}{[s_1, n_1, nq_1] \parallel [s_2, n_2, nq_2] \xrightarrow{\mathsf{disconnect}(n_1, n_2)} [s_1, n_1, (nq_1 \setminus msg_{1\leftarrow 2}) \cdot fail_2] \parallel [s_2, n_2, (nq_2 \setminus msg_{2\leftarrow 1}) \cdot fail_1]}$$

Figure 13. Node-disconnect rule

pid that has previously been running on the node (i.e. the process identifiers in *npl*). The reason for this is to simplify link handling. If all previous processes exists on the node, future link-requests sent to these processes get the correct respose without having to create any further semantic rules. The function init() is an initialization process which is started on a new node. What the init()-process does is not further specified in the semantics, but it could 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.

$$\textit{interleave}_{\textit{node}}\ \frac{n_1 \xrightarrow{\gamma} n_1'}{n_1 \parallel n_2 \xrightarrow{\gamma} n_1' \parallel n_2}$$

Figure 14. Node interleaving (symmetrical rule omitted)

The third node rule, node-disconnect, is presented in Fig. 13. The rule handles the situation when the communication channel between two nodes break down. The result of this is that all messages from/to the disconnected nodes that are currently in the node queue are lost, and that a set of link messages (exit-messages) are created and sent. The rule uses two functions nqMatchAll and filterLinks (defined below) to collect messages that are lost and links that should be converted to a exit-messages. The rule discards messages and adds link notifications to the node queues of the disconnected nodes. Some things should be noted, first, the order of messages between a pair of processes is not affected by this rule. Either the messages are delivered in order or not at all. Second, to drop all messages in transit between the disconnected nodes is a design choice, we could just as well drop only an arbitrary set of messages. This is further discussed in Sect. 6. Finally we note that there is no *node-(re)connect*-rule, since the reconnection of nodes is transparent at the semantic level.

Definition 20 The function nqMatchAll(erlangNodeQueue, To, From) is a function that given an Erlang node message queue, a sender process id (*from*) and a receiver process id (*to*) returns all messages in the queue sent by *from* to *to*. The function filterLinks($\mathcal{P}$(erlangPid $\times$ erlangPid), erlangNodeName, erlangNodeName) is a function that filters a set of process identifier pairs with respect to the node the processes are running at. E.g. let $\mathsf{node}(p_{1x}) = n_1$, $\mathsf{node}(p_{2x}) = n_2$ and $\mathsf{node}(p_{3x}) = n_3$ then:
$\mathsf{filterLinks}(\{(p_{11}, p_{21}), (p_{12}, p_{34}), (p_{27}, p_{12})\}, n_1, n_2) =$
$\{(p_{11}, p_{21})\}$

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 $\textit{input}_{\textit{node}}$ 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 [15]. That is, the underlying semantics properly destroy messages and reply to link-requests.

4.9 Fairness

As we noted above, the input-rule, i.e. the $\textit{input}_{\textit{node}}$ rule in Fig. 10, 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 a fairness-rule which will exclude certain unwanted behavior of the system. Fairness is defined in terms of permissable *execution sequences*.

Definition 21 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 22 [**Fairness for inter-node communication**]
It should hold for all execution sequences, $(\vec{n}, \vec{\gamma})$:

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

That is, Definition 22 state that every sent message is eventually delivered to the recieving process, or the node where the receiving process is executed dies, or a node disconnection involving the sending and the receiving node occurs.

4.10 Message reordering

The motivation for extending Fredlund's single-node semantics was partly to capture the distributed behavior where messages were reordered. Therefore, we repeat the 'hello world' example from Sect. 3, but now we execute the program from Fig. 4 in the extended semantics. The example is presented in Fig. 15.

4.11 Node disconnection

Another part of the motivation for the multi-node semantics was to capture the behavior where nodes disconnect. Therefore, we conclude the presentation of the extended semantics with an example with node disconnection. The example is presented in Fig. 17, where we execute the program in Fig. 16 in the multi-node semantics. The program consists of two processes where the first process sends a the number sequence [1, 2, 3] to the other. The effect of the node disconnect is that the second process can receive the sequence [1, 3].

5. Properties of the Multi-Node Semantics

In the previous section we have defined a distributed semantics for a subset of Erlang. The limitations are the same as Fredlund have in the single node semantics [15]; in short standard Erlang without: modules, floats, references, binaries, ports, the catch-expression and some internal process state information such as process dictionaries. In the design of the semantics we have had several informal properties which we have intended for the distributed semantics. In this section we argue for some of them and describe why they are desirable and why they actually hold for the distributed semantics.

5.1 Extension

The first property we want is that the distributed semantics is a true extension of the single-node semantics. That is if we take a single-node system and execute it in the multi-node semantics (this is possible with only minor modifications of the system) it should work exactly the same. In order to express 'exactly the same' and 'minor modifications' in a strict way we have to define the mkDist function as well as an *execution sequence*.

1. Initial system:

$N_0 = [\langle \texttt{PidC} = \texttt{spawn(nodeC,procC,[])} \ \ldots, p_0, \epsilon\rangle, n_0, \epsilon]$

2. The only scheduler option is to spawn `procC`. After that, the only option is to spawn `procB` since the `receive` in `procC` blocks. This results in three processes:

$N_0 = [\langle \texttt{PidC ! hello} \ldots, p_{01}, \epsilon\rangle, n_0, \epsilon]$
$N_1 = [\langle \texttt{receive } X \rightarrow \texttt{PidC ! } X \texttt{ end}, p_{11}, \epsilon\rangle, n_1, \epsilon]$
$N_2 = [\langle \texttt{receive } X \rightarrow \texttt{ok end}, p_{21}, \epsilon\rangle, n_2, \epsilon]$

3. Only N_0 can make progress, since N_1 and N_2 are blocked on a `recieve` statement:

$N_0 = [\langle \texttt{PidB ! world}, p_{01}, \epsilon\rangle, n_0, \epsilon]$
$N_1 = [\langle \texttt{receive } X \rightarrow \texttt{PidC ! } X \texttt{ end}, p_{11}, \epsilon\rangle, n_1, \epsilon]$
$N_2 = [\langle \texttt{receive } X \rightarrow \texttt{ok end}, p_{21}, \epsilon\rangle, n_2, \epsilon \cdot (p_{01}, p_{21}, \texttt{hello})]$

4. Now we have two options, either apply the $input_{node}$ on N_2 or let N_0 send its second message. Since the purpose is to show message re-ordering we let N_0 proceed:

$N_0 = [\langle\!\langle p_{01}, \epsilon \rangle\!\rangle, n_0, \epsilon]$
$N_1 = [\langle \texttt{receive } X \rightarrow \texttt{PidC ! } X \texttt{ end}, p_{11}, \epsilon\rangle,$
$\quad n_1, \epsilon \cdot (p_{01}, p_{11}, \texttt{world})]$
$N_2 = [\langle \texttt{receive } X \rightarrow \texttt{ok end}, p_{21}, \epsilon\rangle,$
$\quad n_2, \epsilon \cdot (p_{01}, p_{21}, \texttt{hello})]$

5. Again we have two options, we can apply $input_{node}$ to either N_1 or N_2, to illustrate our point, we choose N_1:

$N_1 = [\langle \texttt{receive } X \rightarrow \texttt{PidC ! } X \texttt{ end}, p_{11}, \epsilon \cdot \texttt{world}\rangle, n_1, \epsilon]$
$N_2 = [\langle \texttt{receive } X \rightarrow \texttt{ok end}, p_{21}, \epsilon\rangle, n_2, \epsilon \cdot (p_{01}, p_{21}, \texttt{hello})]$

6. Now we continue to ignore N_2 and let p_{11} read its message and send it to p_{21}:

$N_1 = [\langle\!\langle p_{11}, \epsilon \rangle\!\rangle, n_1, \epsilon]$
$N_2 = [\langle \texttt{receive } X \rightarrow \texttt{ok end}, p_{21}, \epsilon\rangle,$
$\quad n_2, \epsilon \cdot (p_{01}, p_{21}, \texttt{hello}) \cdot (p_{11}, p_{21}, \texttt{world})]$

7. Finally we have arrived in the wanted situation, we can apply $input_{node}$ to N_2, and by its construction it is perfectly ok to deliver the `world` message:

$N_2 = [\langle \texttt{receive } X \rightarrow \texttt{ok end}, p_{21}, \epsilon \cdot \texttt{world}\rangle,$
$\quad n_2, \epsilon \cdot (p_{01}, p_{21}, \texttt{hello})]$

8. Now `world` is received before `hello`.

Figure 15. Hello World – Multi-Node Execution

```
init() ->
    PidB = spawn(?NODE2,?MODULE,procB,[]),
    PidA = spawn(?NODE1,?MODULE,procA,[PidB]).

procA(PidB) ->
    PidB ! 1,
    PidB ! 2,
    PidB ! 3.

procB() ->
    receive X -> ok end,
    receive Y -> ok end,
    receive Z -> ok end.
```

Figure 16. Simple One-Two-Three counting program

Definition 23 The function mkDist(erlangNodeId,erlangSystem) takes a single-node Erlang system and prepare it for execution in the multi-node semantics (on the given node). The necessary change is to replace each $\mathsf{spawn}(f, [v_1, \ldots, v_m])$ with the distributed variant $\mathsf{spawn}(n, f, [v_1, \ldots, v_m])$, where n is the given node.

Definition 24 An *execution sequence* for an Erlang is defined to be the sequence of system actions (see Def. 9) performed by the executed system. Since the scheduler is non-deterministic an Erlang system will have a (large) set of possible execution sequences.

1. Initial system:

$N_0 = [\langle \texttt{PidB} = \texttt{spawn}(\texttt{n}_2\texttt{,procB,[]}) \ \ldots, p_0, \epsilon\rangle, n_0, \epsilon]$

2. The only scheduler option is to spawn `procB`. After that, the only option is to spawn `procA` since the `receive` in `procB` blocks. This results in two processes (the first one is terminated):

$N_1 = [\langle \texttt{PidA ! 1} \ldots, p_{11}, \epsilon\rangle, n_1, \epsilon]$
$N_2 = [\langle \texttt{receive } X \rightarrow \texttt{ok end} \ldots, p_{21}, \epsilon\rangle, n_2, \epsilon]$

3. Again there is only one option, to let `PidA` send `1`:

$N_1 = [\langle \texttt{PidA ! 2} \ldots, p_{11}, \epsilon\rangle, n_1, \epsilon]$
$N_2 = [\langle \texttt{receive } X \rightarrow \texttt{ok end} \ldots, p_{21}, \epsilon\rangle, n_2, \epsilon \cdot (p_{11}, p_{21}, \texttt{1})]$

4. To illustrate our point, we now let N_2 execute, the $input_{node}$- rule is applied:

$N_1 = [\langle \texttt{PidA ! 2} \ldots, p_{11}, \epsilon\rangle, n_1, \epsilon]$
$N_2 = [\langle \texttt{receive } X \rightarrow \texttt{ok end} \ldots, p_{21}, \epsilon \cdot \texttt{1}\rangle, n_2, \epsilon]$

5. Now we proceed by letting N_1 execute and send another number:

$N_1 = [\langle \texttt{PidA ! 3} \ldots, p_{11}, \epsilon\rangle, n_1, \epsilon]$
$N_2 = [\langle \texttt{receive } X \rightarrow \texttt{ok end} \ldots, p_{21}, \epsilon \cdot \texttt{1}\rangle, n_2, \epsilon \cdot (p_{11}, p_{21}, \texttt{2})]$

6. Now assume that the nodes disconnect, and thus we apply the *node-disconnect* rule, note that the message already delivered to p_{21} is not affected:

$N_1 = [\langle \texttt{PidA ! 3} \ldots, p_{11}, \epsilon\rangle, n_1, \epsilon]$
$N_2 = [\langle \texttt{receive } X \rightarrow \texttt{ok end} \ldots, p_{21}, \epsilon \cdot \texttt{1}\rangle, n_2, \epsilon]$

7. Now the nodes are re-connected, and we can proceed by letting N_1 execute and send yet another number:

$N_1 = [\langle\!\langle p_{11}, \epsilon \rangle\!\rangle, n_1, \epsilon]$
$N_2 = [\langle \texttt{receive } X \rightarrow \texttt{ok end} \ldots, p_{21}, \epsilon \cdot \texttt{1}\rangle, n_2, \epsilon \cdot (p_{11}, p_{21}, \texttt{3})]$

8. Now we see that p_{21} will receive the sequence [1,3], which is not possible in the single-node semantics.

Figure 17. One-Two-Three – Multi-Node Execution

Prop 1. A single-node Erlang system ($s \in$ erlangSystem) which is prepared for execution in the multi-node semantics by the mkDist function has exactly the same set of possible *execution sequences* as the system s executed in the single-node semantics, as long as the node is not allowed to fail.

This is true because the only semantic rules that are invoked are those in the single node semantics (together with the non-interfering $silent_{node}$ rule). This we can be sure of because there is only one way to 'escape to' the multi-node semantics, namely by communication not caught by the *com*-rule (Fig. 7). And because we have the side condition in the $spawn_{local}$-rule that the node is the same for the spawned processes, all communication is caught by the *com*-rule. Note that we have to disallow the node failure, or the distributed version of the system would have a larger set of possible execution sequences. Node disconnect is not a problem since we have only one node!

5.2 Message Reordering and Node Disconnect

The main motivation for the semantics for distributed Erlang was to be able to express message reordering and node disconnect in the semantics, it is therefore desirable that this is indeed possible.

Prop 2. In a distributed system, a message sent from a process (A) via a second process (B) to a third process (C), can arrive before a message sent directly from A to C at some earlier point in time. Messages between two processes at different nodes can also be lost due to node disconnection.

In the examples in Fig. 15 and Fig. 17 we demonstrate that messages can arrive in the desired way and that node disconnection is indeed possible. By the design of the function nqMatch the opposite, namely that messages between a pair of processes are always ordered, is also ensured.

5.3 Expressiveness

It is also important that the extended semantics is complete in terms of expressiveness. Every correct distributed system must be able to execute in the semantics, and it must not get stuck because there is something not expressible in the semantics.

Definition 25 Progress for a process is equivalent to applying one of the evaluation rules for processes. Further an Erlang system can make progress if and only if any of its processes can make progress.

Prop 3. An Erlang (Multi-node) system which is not in a deadlocked situation and that does have at least one live process should be able to make progress according to the semantic rules.

This is a property that should hold for the underlying semantics, even though it is not proved by Fredlund in [15]. We would like to prove something similar for ERTS/EMNS, but here we have the extra complications of nodes. Because of the rules *node-failure*,*node-start* and *node-disconnect* an EMNS can always do something namely that a node fail or restart or that two nodes disconnect.

5.4 Finite systems stays finite

Another concern is that the extension of the semantics may introduce unwanted overhead in terms of resources. Since the semantics should model the behavior of the run-time system it is important that a system that executes with finite bounds on the message queues does so also in the semantics.

Prop 4. A finite system (that is with finite bounds of the length of the message queues) in the single-node semantics is still finite in the multi-node semantics.

The main argument for why this holds is that there is a one-to-one mapping of messages. That is there is no place in the multi-node semantics where a message is duplicated, therefore the number of messages stays the same. Since we also have the fairness rule all messages should eventually be delivered, and there is no inherent way that systems become infinite in the distributed semantics.

5.5 A word of caution

After we have argued above that the desired properties of the distributed semantics holds it is important to note a few artifacts of the functional differences between a single-node and a multi-node system. Because of the functional differences, a non-deadlocking single-node system might very well dead-lock if run in a distributed setting. For example imagine a system which depends on the message ordering implied by the single-node semantics[2], such a system could easily dead-lock if we distribute it over several run-time systems. In the same way, also a finite single-node system might very well become infinite if it is run in a distributed environment.

6. Discussion

The fundamental characteristics of Erlang are 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 [23]. 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.”

This is indeed true for the semantics presented by Fredlund [15], but due to the construction of the *com*-rule (in Fig. 3), even stronger properties hold in the single-node semantics. In Fredlund’s single-node semantics the delivery of a message to another process is instantaneous, meaning that **all** messages are delivered in exactly the order they are sent. Because of how the standard (OTP) Erlang run-time system is implemented, this happens to be true also for a real Erlang system where all processes are running on the same node. However, it is not true in general for a concurrency oriented programming language, and specifically not in a distributed setting with several different Erlang nodes.

There is an intricate choice here, on one side we have the de facto standard, the OTP implementation of the Erlang run-time system where local communication is instantaneous. On the other side we have the different Erlang specifications, where no support for such instantaneous communication can be found. Our original concern was to produce a semantics to be used for model checking, and therefore the presentation is biased by this. Since Fredlund’s single-node semantics faithfully describes what actually happens inside the standard run-time system, we argue that for efficient model checking of Erlang systems (to be run on in the standard run-time system), the underlying semantics should be kept as it is. Fredlund’s intra-node message passing is not consistent with the Erlang specifications, but using a special version of local message passing makes certain (local) systems easier to reason about.

Nevertheless, it is somewhat unconventional to produce a semantics for a particular implementation of a language, and thus one could argue that we should instead present the more general semantics. The alternative then is to only have the kind of message passing rules that we have in the node-to-node communication. Such a modification would be rather straightforward to do. The consequence of this is an overall simpler (and more general) semantics, which is less restrictive for a local system. However, this is problematic in the model checking context, since it results in a bigger state space. Another problem is the introduction of false negatives, because a local system might appear to fail due to an order of events not possible in reality (in the standard Erlang implementation). Such a general semantics would however be useful in case of a future Erlang implementation that adheres more closely to the Erlang specifications. (Such an implementation would of course also require changes to model checkers using the semantics presented in this paper.)

This picture may also be further complicated in the future by the introduction of multi-core systems and the SMP-version of the Erlang run-time system.

In the development of the multi-node semantics, we have also made several other distinctive choices. One particular choice is seen in the *node-disconnect* rule, where all messages currently in transit between two nodes are dropped. We could just as well have dropped an arbitrary sequence of messages in the node queue. However, dropping all messages is certainly easier and it makes the fairness rule less complicated. This is also an example of where a simpler rule offer the same expressivity as a more complicated one. Every sequence of messages in the node queue could indeed be dropped, it is just a matter of applying the rule at the right time. Another design 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.

[2] One could indeed say that such a system is simply containing a bug, since the enforcment of such a static message order is nowhere to be found in the Erlang specifications. Nevertheless, since the de facto standard, the OTP Erlang run-time system implementation, actually behave this way such programs are going to be written.

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. 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. 18. 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. 9.

Another part of the linking mechanism are the somewhat complicated *node-failure* and *node-disconnect* rules (Fig. 11 and Fig. 13), 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. This actually an important observation in a more general perspective. The goal with the semantics is to be able to express all the possible behavior of the Erlang implementation, and not to describe how the implementation actually works. In the Erlang implementation, the run-time system keeps track of links via a timeout construction. In the semantics we instead do the book keeping (so to say) at the other end. Therefore, we have to take extra care when node fails since messages in nq are otherwise lost.

The *node-disconnect* rule looks rather horrible from a programmers point of view, at any time, all messages between a pair of nodes may be lost. However this is not as disastrous as one might think, as long as one is aware that this might happen. This is because the link-mechanism works in the node disconnect case, and as long as communication is restricted to monitored receivers there is no immediate danger.

Finally we should discuss one limitation of the distributed semantics, namely that *monitors* are not a part of the semantics. This is a limitation because monitors are widely used, and the correctness of many distributed Erlang systems rely on monitors. One is tempted to belive that it is possible to implement monitors in terms of links. This is however only partly true, since monitors would have to be implemented using a named and dedicated process for each node. This means that in order to get a correct behavior we have to ensure that no one is for example killing the monitor process. Therefore, it is not obvious how to implement monitors in terms of links and it seems that we have to make some non-trivial assumptions.

7. Related Work

The semantics for Erlang is informally described in [2]. Thereafter, a first, not completed, attempt to formally specify the semantics of Erlang was made by Petterson [20]. Petterson, inspired by similar work with Standard ML and Relational ML, used Natural Semantics but did not finish the work. 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 [12] and [13]. These attempts, compared to the semantics presented by Fredlund in [15],

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

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

Figure 18. Erlang program - Linking

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 [19]. 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 [12] and [19] make use of subsets of Erlang referred to as core fragments of Erlang. These references should not be confused with the Core Erlang project [10], 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.

The distributed semantics for Erlang presented in [11] has been used in a model checker for Erlang, McErlang [16]. Implementing a model checker is the ultimate test for a semantics, and several limitations were indeed found, which motivated the further work on a distributed semantics for Erlang.

8. Conclusions and Future Work

In 2005, we proposed an extension of the Fredlund's single-node semantics into a distributed (multi-node) semantics for Erlang [11]. We augmented the single-node semantics with a distributed layer, and were able to successfully model multi-node programs in the extended semantics. Together with the distributed semantics there was also a warning to the community of potential pit-falls with testing and verification using a single-node semantics.

Later the multi-node semantics was used in the implementation of a model checker for Erlang (McErlang, [16]). The model checker implementation revealed some inconsistencies as well as a major shortcoming in the multi-node semantics. The main problem was the Erlang behavior in the case of *node disconnect*. Two Erlang nodes can become disconnected from each other if the link between them fails, when this happens both involved nodes regard the other node as dead. This does have some interesting consequences when the nodes later re-connect. The correct behavior was later implemented in the model checker, however this behavior could not be modeled in the multi-node semantics we had proposed.

In this new presentation of a distributed semantics for Erlang, we have restructured some parts of the distributed layer. Further, we have added the node disconnect functionality, we have corrected errors in the original presentation and we have simplified and clarified the presentation in many aspects. The result is a more accurate and more expressive multi-node semantics for Erlang, with a clearer presentation and less complicated semantic rules. We have also added a discussion of the desired properties of the multi-node semantics.

Future Work The introduction of multi-core computer systems and the development of a SMP-version of the Erlang run-time is already a fact. And it will be interesting to see if there are any new semantic implications because of this. There is also further work to be done with model checking Erlang in the distributed setting.

Acknowledgments

We thank Koen Claessen 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. Barklund and R. Virding. Erlang 4.7.3 reference manual. Draft (0.7), Ericsson Computer Science Laboratory, 1999.

[9] 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.

[10] 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.

[11] K. Claessen and H. Svensson. A semantics for distributed erlang. In *ERLANG '05: Proceedings of the 2005 ACM SIGPLAN workshop on Erlang*, pages 78–87, New York, NY, USA, 2005. ACM Press.

[12] 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.

[13] 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.

[14] 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.

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

[16] L.-Å. Fredlund and C. B. Earle. Model checking erlang programs: the functional approach. In *ERLANG '06: Proceedings of the 2006 ACM SIGPLAN workshop on Erlang*, pages 11–19, New York, NY, USA, 2006. ACM Press.

[17] 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.

[18] 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.

[19] 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.

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

[21] 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.

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

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