Towards a Formalisation of Erlang Failure and Failure Detection

Audrianne Farrugia
CS Dept, ICT
University of Malta
afar0035@um.edu.mt

Adrian Francalanza
CS Dept, ICT
University of Malta
adrian.francalanza@um.edu.mt

ABSTRACT

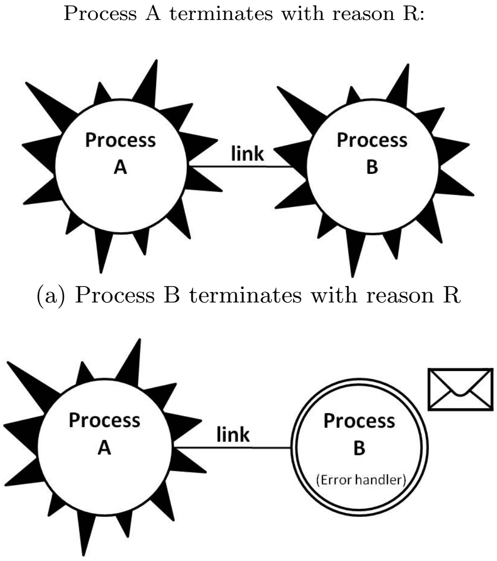
This paper discusses preliminary investigations on the behaviour of the error handling mechanisms in Erlang, a parallel language which is renowned for its fault tolerant capabilities. A formal model is defined in order to provide a precise and unambiguous description of the behaviour of these mechanisms. The correctness of the model is evaluated by considering a simple Erlang program and comparing the behaviour as described by the formal semantics with that of actual Erlang.

1. INTRODUCTION

Concurrent programming, popularised by recent technological advances such as distributed computing over the Internet and the proliferation of multi-core computing chips, lends itself naturally to the construction of *fault-tolerant* code. In this setting, failure - stemming from hardware faults as well as from software programming errors - typically affects only parts of the code, *i.e., partial failure*; the unaffected code then attempts to recover from this failure through the use of redundant resources or else degrade the computation gracefully. One clear manifestation of this is code written in the concurrent language Erlang[2, 3], an industry-strength, dynamically-typed, actor-based, functional language equipped with mature fault tolerance mechanisms.

1.1 Actors, Links and Failures

Concurrency in Erlang is based on the *actor* model [1], whereby concurrent threads of execution interact through message-passing rather than through the mutual access of a common shared state. More precisely, when a concurrent process, *i.e.,* actor, wants to share data with another process, it creates a *copy* of this data and sends it as a *message* to the other process, identified by a *unique process Id.* Processes are equipped with their own queue structure called a *mailbox*, where incoming messages are received; this mailbox acts as the process *exclusive, local memory* from which messages can be selected to be read using pattern-matching.



(a) Process B terminates with reason R

(b) Process B traps the exit signal

Figure 1: Error Propagation

Erlang builds its fault-handling mechanism on top of this messaging structure. For a start, when an Erlang process fails, no memory is affected apart from its own local copy; this simplifies the delineation of the fault effect. Moreover, a failing process generates an *exit* message that is automatically sent to other processes. The destinataries of the exit messages are determined through the process *linking* mechanism, where processes are explicitly linked to one another so as to listen to each others' termination messages. Finally, Erlang processes can determine how to handle these exit messages once received: by setting an appropriate flag, they can decide whether to automatically fail upon the receipt of an exit message, or else store this message in their mailbox to be handled later on; the latter option is often referred to as *trapping exits*, and the respective processes that do this are termed *system processes.*

Figure 1 illustrates how the error-propagation mechanism just described works. In both cases, processes A and B are linked to one another. In Figure 1(a), process B is not trapping exits and when process A terminates abnormally, *i.e.,* it fails, the exit message will cause process B to automatically terminate abnormally upon message receipt. In Figure 1(b) however, process B is a system process (denoted using double lines); when the exit signal is received, process *B* traps it to be able to carry out the necessary error handling operations.

1.2 Local and Remote Error Handling

Erlang's fault-handling mechanism suggests a *fail-fast* approach [7] to code design, whereby code passes the responsibility for handling errors to other processes at a higher design level. In fact, accepted Erlang code practices [3] discourage *defensive programming*, *i.e.*, code that tries to anticipate and handle locally abnormal computation, because it clutters the code and obfuscates the point from where the error originates. Instead [10, 3] claim that a fail-fast approach leads to (*i*) a reduction in code that is (*ii*) easier to debug and maintain, (*iii*) is less likely to erroneously perform irreversible or costly operations, and (*iv*) well-suited to fault-tolerant organisations through the use of redundancy.

Program 1.1 Local Error Handling in Erlang

```
fun1(X,Y,Pid) ->
   try
         Pid ! X + Y
   catch
         error:badarith -> Pid ! invalid
   end.
```

Program 1.1 is a a typical example of a defensive Erlang program that attempts to handle errors locally. Function `fun1` sends the sum of variables `X` and `Y` to the process identified by the variable *Pid*. The try-catch statement is used to handle any errors that may occur when computing the sum of the two variables. Thus, if `X` or `Y` are bound to non-numeric data, a `badarith` exception is raised; this exception is then caught by the try-catch statement and the atom `invalid` is sent to the process identifier bound to the variable `Pid`.

Program 1.2 Remote Error Handling in Erlang

```
fun1(X,Y,Pid) -> Pid ! X + Y.

ff_wrap(F,Args,Pid) ->
   P = spawn_link(?MODULE,F,Args),
   process_flag(trap_exit,true),
   receive
      {'EXIT',P,{badarith,Stack}}
                                   -> Pid ! invalid
      {'EXIT',P,normal} -> ok;
   end.
```

Program 1.2 performs the same task as Program 1.1 while avoiding any defensive programming. Here, `fun1` sends the sum of `X` and `Y` to `Pid` *without* attempting to anticipate possible misuses of the function; this keeps the function definition small and perspicuous. Abnormal computation is handled by another process executing `ff_wrap` (a wrapper for *fail-fast* functions), which listens for erroneous terminations of a function execution of (the redefined) `fun1` and reports to process `Pid` the atom `invalid` whenever a `badarith` error is detected.[1] The function `ff_wrap` is generalised as a pattern[2], and works with any function that may generate a `badarith` error; this improves code reuse which, in turn, leads to code reduction.

Intuitively, the two programs are meant to describe the same behaviour. Because of the advantages discussed earlier in Section 1.1, we would like to refactor Program 1.1 with Program 1.2 and, accordingly, substitute calls to spawn processes computing the addition of numbers `N` and `M`:

```
spawn(?MODULE,fun1,[N, M,self])          (1)
```

with the respective calls:

```
spawn(?MODULE,ff_wrap,[fun1,[N,M,self],self])  (2)
```

In order to carry out this refactoring in a safe manner, we would need to determine whether the two sub-programs exhibit the same behaviour under *any context* and for *every possible process interleaving.* It turns out that Program 1.1 and Program 1.2 may indeed exhibit *different* behaviour for certain process interleaving but, unfortunately, this fact tends to be notoriously hard to determine in concurrent system: (*i*) if such differing behaviour is discovered through testing, the specific process interleaving leading to the violation may be hard to replicate in other test runs and, moreover, (*ii*) the reason for the violation is often subtle and determining the exact cause requires intimate knowledge of how the constructs leading to the violation actually work.

1.3 Aims and Objectives

This paper discusses preliminary results towards developing a formal model for a subset of Erlang focussing on the error-handling mechanism of the language. The formal model is intended to help give a high-level understanding how certain Erlang constructs work, without needing to go *'under the bonnet'* to see how these constructs are implemented. This high-level understanding should, in turn, help us predict the behaviour of Erlang processes and facilitate debugging of subtle errors such as those discussed in Section 1.2. The model will also guide the implementation of debugging tools that analyse the behaviour of concurrent Erlang systems. Ultimately, however, we expect this model to lay the necessary foundations for the eventual development of a semantic theory for the language that would enable us to determine when two programs are semantically equivalent.

The rest of the paper is structured as follows. In Section 2 our formal model is presented. In section 3 this model is used to analyse the behaviour of the programs discussed in the Introduction; we also give a brief description of how the formal model was animated through an evaluator. Section 4 overviews of how the correctness of the formal model was assessed. Finally, Section 5 gives some suggestions for future work and Section 6 concludes.

2. A FORMAL SEMANTICS

We define a calculus for modelling the computation of Erlang programs. We assume denumerable sets of process identifiers

[1] The command `spawn_link(M,F,A)` spawns a new process executing function `F` defined in module `M` instantiated with arguments `A`, and links the spawned process to the process executing the command. The command `process_flag(trap_exit,true)` turns the process into a system process.

[2] In our case, it will be passed the function name `fun1` and the list `[X,Y,Pid]` as the arguments `F` and `Args`, respectively.

$i, j, h \in \textsc{Prc}$ and of variables $x, y, z \in \textsc{Var}$. We let $s, t \in \{\bullet, \circ\}$ range over actor trapping modes (status), where $\bullet$ and $\circ$ denote exit-trapping and non exit-trapping *resp.*

$$
\begin{array}{rcl}
A, B \in \textsc{Act} & ::= & i[c \lhd m]^{s}_{l} \mid i[v]_{l} \mid A \parallel B \\
c \in \textsc{Cmd} & ::= & x = e, c \mid e \\
e, f \in \textsc{Exp} & ::= & v \mid \mathsf{self} \mid e!e \mid \mathsf{rcv}\, x\, \mathsf{in}\, e \mid \mathsf{try}\, e\, \mathsf{catch}\, e \\
 & \mid & e\, e \mid \mathsf{trap} \mid \mathsf{spw}\, e \mid \mathsf{spw\_lnk}\, e \mid \ldots \\
v \in \textsc{Val} & ::= & x \mid i \mid \mathsf{fun}\,(x)c \mid \mathsf{exit} \mid \mathsf{end}(i, v) \mid \ldots
\end{array}
$$

A system is made up *actors*, $\textsc{Act}$, that are composed in *parallel*, $A \parallel B$, whereby each individual actor/process may be either *active* or *terminated*. A terminated actor, $i[v]_{l}$, is just a final value v (set to exit when termination is abnormal) that is *uniquely* identified by i, and is linked to a list of actors l. An active actor, $i[c \lhd m]^{s}_{l}$, can be either trapping exits ($s = \bullet$) or not ($s = \circ$). It consists of a command, c, executing *wrt.* a local mailbox, m, whereby, as in the case of a terminated actor, this pair is uniquely identified by an identifier, i, and linked to a list of other processes, l.

We let $l, k \in \textsc{Prc}^{*}$ range over lists of process identifiers, which are used to denote links to other actors; the notation $i : l$ denotes the link-list with process identifier i at the head and l being the tail; dually, $l : i$ denotes the link-list l appended with process identifier i. Mailboxes, are also denoted by lists: we let $m, n \in \textsc{Val}^{*}$ to range over message queues of values. Similar to link lists, $v : m$ denotes the mailbox with v at the head of the queue and $m : v$ denotes the mailbox m appended by v at the end.

Commands consist of a sequence of variable binding terminated by an expressions, $x_1 = e_1, \ldots, x_n = e_n, e_{n+1}$. Expressions are expected to evaluate to values, and may consist of self reference, self, output to another actor, $e!e$, input from the mailbox, $\mathsf{rcv}\, x\, \mathsf{in}\, e$, function application, $e\, e$, trap-exit setting, trap, process/actor spawn, $\mathsf{spw}\, e$, and spawn with linking, $\mathsf{spw\_lnk}\, e$, amongst others. Values may consist of process ids, i, functions, $\mathsf{fun}\,(x)e$, exit exceptions,[3] exit, and termination reports, $\mathsf{end}(i, v)$, amongst others. Commands and expressions also specify evaluation contexts for our expressions, denoted as $\mathcal{C}[-]$ and defined as:

$$
\begin{array}{rcl}
\mathcal{C} & ::= & \mathcal{E}[-] \mid x = \mathcal{E}[-], c \\
\mathcal{E} & ::= & [-] \mid \mathcal{E}[-]!e \mid v!\mathcal{E}[-] \mid \mathsf{try}\, \mathcal{E}[-]\, \mathsf{catch}\, e \\
\mid & & \mathcal{E}[-]\, e \mid v\, \mathcal{E}[-] \mid \mathsf{spw}\, \mathcal{E}[-] \mid \mathsf{spw\_lnk}\, \mathcal{E}[-] \mid \ldots
\end{array}
$$

At command level, an expression is only evaluated when at the top level variable binding or the final expression. We do not fully list the expression level evaluation contexts, but these are fairly standard.

$$\textsc{Cmd}\ \frac{v \neq \mathsf{exit}}{i[x = v, c \lhd m]^{s}_{l} \longrightarrow i[c\{{}^{v}\!/\!_{x}\} \lhd m]^{s}_{l}}$$

$$\textsc{CmdE}\ \frac{}{i[x = \mathsf{exit}, c \lhd m]^{s}_{l} \longrightarrow i[\mathsf{exit} \lhd m]^{s}_{l}}$$

$$\textsc{Slf}\ \frac{}{i[\mathcal{C}[\mathsf{self}] \lhd m]^{s}_{l} \longrightarrow i[\mathcal{C}[i] \lhd m]^{s}_{l}}$$

$$\textsc{Snd}\ \frac{}{i[\mathcal{C}[j!v] \lhd m]^{s}_{l} \parallel j[c \lhd n]^{t}_{k} \longrightarrow i[\mathcal{C}[v] \lhd m]^{s}_{l} \parallel j[c \lhd n : v]^{t}_{k}}$$

$$\textsc{Rcv}\ \frac{}{i[\mathcal{C}[\mathsf{rcv}\, x\, \mathsf{in}\, e] \lhd v : m]^{s}_{l} \longrightarrow i[\mathcal{C}[e\{{}^{v}\!/\!_{x}\}] \lhd m]^{s}_{l}}$$

$$\textsc{Try}\ \frac{v \neq \mathsf{exit}}{i[\mathcal{C}[\mathsf{try}\, v\, \mathsf{catch}\, e] \lhd m]^{s}_{l} \longrightarrow i[\mathcal{C}[v] \lhd m]^{s}_{l}}$$

$$\textsc{Ctc}\ \frac{}{i[\mathcal{C}[\mathsf{try\ exit\ catch}\, e] \lhd m]^{s}_{l} \longrightarrow i[\mathcal{C}[e] \lhd m]^{s}_{l}}$$

$$\textsc{App}\ \frac{}{i[\mathcal{C}[(\mathsf{fun}\,(x)c)\, v] \lhd m]^{s}_{l} \longrightarrow i[\mathcal{C}[c\{{}^{v}\!/\!_{x}\}] \lhd m]^{s}_{l}}$$

$$\textsc{Trp}\ \frac{}{i[\mathcal{C}[\mathsf{trap}] \lhd m]^{\circ}_{l} \longrightarrow i[\mathcal{C}[i] \lhd m]^{\bullet}_{l}}$$

$$\textsc{Spw}\ \frac{\mathit{fresh}(j)}{i[\mathcal{C}[\mathsf{spw}\, e] \lhd m]^{s}_{l} \longrightarrow i[\mathcal{C}[j] \lhd m]^{s}_{l} \parallel j[e \lhd \epsilon]^{\circ}_{\epsilon}}$$

$$\textsc{SpwL}\ \frac{\mathit{fresh}(j)}{i[\mathcal{C}[\mathsf{spw\_lnk}\, e] \lhd m]^{s}_{l} \longrightarrow i[\mathcal{C}[j] \lhd m]^{s}_{l:j} \parallel j[e \lhd \epsilon]^{\circ}_{i}}$$

$$\textsc{Fin}\ \frac{}{i[v \lhd m]^{s}_{l} \longrightarrow i[v]_{l}}$$

$$\textsc{TrmS}\ \frac{v \neq \mathsf{exit}}{i[v]_{j:l} \parallel j[c \lhd m]^{\circ}_{k} \longrightarrow i[v]_{l} \parallel j[c \lhd m]^{\circ}_{k-i}}$$

$$\textsc{TrmE}\ \frac{}{i[\mathsf{exit}]_{j:l} \parallel j[c \lhd m]^{\circ}_{k} \longrightarrow i[\mathsf{exit}]_{l} \parallel j[\mathsf{exit}]_{k-i}}$$

$$\textsc{TrmT}\ \frac{}{i[v]_{j:l} \parallel j[c \lhd m]^{\bullet}_{k} \longrightarrow i[v]_{l} \parallel j[c \lhd m : \mathsf{end}(i, v)]^{\bullet}_{k-i}}$$

Table 1: Reduction Rules

The behaviour of our modelled Erlang programs is given in terms of a reduction semantics over systems of actors. Table 1 gathers the main rules relevant to us. Rule $\textsc{Cmd}$ describes how command sequences act as let constructs where a value binding, v for x, is substituted in the continuation c; if however the value is an exception, $\textsc{CmdE}$, the remaining commands are ignored. Rule $\textsc{Slf}$ allows an actor to dynamically obtain its Id, since this cannot be known at compile time. Communication happens in two step: the sender first sends the message to the receiver's mailbox, $\textsc{Snd}$, and then the receiver reads it from its mailbox at some later stage, $\textsc{Rcv}$.[4] $\textsc{Try}$ and $\textsc{Ctc}$ describe local exception handling whereas $\textsc{App}$ describes standard function application.

The remaining rules are the most interesting. Rule $\textsc{Trp}$ sets the trap-exit flag, turning the actor into a system process.[5] The rule $\textsc{Spw}$ launches a new actor executing e *wrt.* the empty mailbox, ϵ while being assigned a unique process identifier, j, automatically set *not* to trap exits and is not linked to any other process, ϵ. The reduction of $\textsc{SpwL}$ is similar to that of $\textsc{Spw}$, but establishes a mutual link between

[3] We elide the *reason* parameter in exceptions in this exposition.

[4] Erlang uses pattern matching to select which message to read from the mailbox, something we do not model here.

[5] To keep our exposition self-contained, trap returns the pid of the actor, whereas in the actual Erlang JVM it returns a boolean.

the newly spawned actor and the actor executing the spawn. Fin describes the fact that once an actor executes down to a value, then it transitions to a terminated actor. The final three rules are related and describe how process termination and linking work. If a process terminates normally, *i.e.,* $v \neq \mathsf{exit}$, and the linked process is *not* trapping exits, then the link is silently removed, TrmS. If the linked process is *not* trapping exits but the actor terminates abnormally, this will cause the linked process to terminate abnormally as well, TrmE. Finally, if the linked process is a system process, then the actor termination will generate the appropriate termination message at the linked process's mailbox, TrmT, thereby trapping its exit message.

3. USING THE FORMAL SEMANTICS

Assuming standard currying conventions,[6] we can model the Erlang program (1) from Section 1.2 as the expression[7]

$$e_1 \overset{\text{def}}{=} \mathsf{spw}\,\big((\mathsf{fun}\,(x,y,z)\mathsf{try}\ z!(x+y)\ \mathsf{catch}\ z!\mathsf{invalid}\big)\ n\ m\ \mathsf{self}\big)$$

and program (2) as the expression[8]

$$e_2 \overset{\text{def}}{=} \mathsf{spw}\,\big(f\ (\mathsf{fun}\,(x,y,z)z!(x+y))\ n\ m\ \mathsf{self}\,\mathsf{self}\big)$$

where

$$f \overset{\text{def}}{=} \mathsf{fun}\,(x,y_1,y_2,y_3,z)\left(\begin{array}{l} w = \mathsf{spw\_lnk}\,\big(x\ y_1\ y_2\ y_3\big), \\ \mathsf{trap}, \\ \mathsf{rcv}\,x\,\mathsf{in}\,\mathsf{if}\ x = \mathsf{end}(w,\mathsf{exit}) \\ \qquad\qquad \mathsf{then}\ z!\mathsf{invalid}\ \mathsf{else}\ \mathsf{ok} \end{array}\right)$$

For cases whereby n is an integer but m is not, and a context

$$\mathcal{C}[-] \overset{\text{def}}{=} [-],\mathsf{rcv}\,x\,\mathsf{in}\,e$$

we can observe that the actors $i[\mathcal{C}[e_1] \lhd \epsilon]^s_\epsilon$ and $i[\mathcal{C}[e_2] \lhd \epsilon]^s_\epsilon$ exhibit different behaviour. In the first case, we can only have the following reduction sequence:

$$i[\mathcal{C}[e_1] \lhd \epsilon]^s_\epsilon = i[e_1,\mathsf{rcv}\,x\,\mathsf{in}\,e \lhd \epsilon]^s_\epsilon \longrightarrow \tag{3}$$

$$i[\mathsf{spw}\,\big((\mathsf{fun}\,(x,y,z)\mathsf{try}\ldots)\ n\ m\ i\big),\mathsf{rcv}\,x\,\mathsf{in}\,e \lhd \epsilon]^s_\epsilon \longrightarrow \tag{4}$$

$$\left(\begin{array}{l} i[j,\mathsf{rcv}\,x\,\mathsf{in}\,e \lhd \epsilon]^s_\epsilon \parallel \\ j[\big((\mathsf{fun}\,(x,y,z)\mathsf{try}\ldots)\ n\ m\ i\big) \lhd \epsilon]^\circ_\epsilon \end{array}\right) \longrightarrow \tag{5}$$

$$\left(\begin{array}{l} i[\mathsf{rcv}\,x\,\mathsf{in}\,e \lhd \epsilon]^s_\epsilon \parallel \\ j[\big((\mathsf{fun}\,(x,y,z)\mathsf{try}\ldots)\ n\ m\ i\big) \lhd \epsilon]^\circ_\epsilon \end{array}\right) \longrightarrow^3 \tag{6}$$

$$\left(\begin{array}{l} i[\mathsf{rcv}\,x\,\mathsf{in}\,e \lhd \epsilon]^s_\epsilon \parallel \\ j[\big(\mathsf{try}\ i!(n+m)\ \mathsf{catch}\ i!\mathsf{invalid}\big) \lhd \epsilon]^\circ_\epsilon \end{array}\right) \longrightarrow \tag{7}$$

$$i[\mathsf{rcv}\,x\,\mathsf{in}\,e \lhd \epsilon]^s_\epsilon \parallel j[\big(\mathsf{try}\ \mathsf{exit}\ \mathsf{catch}\ i!\mathsf{invalid}\big) \lhd \epsilon]^\circ_\epsilon \longrightarrow \tag{8}$$

$$i[\mathsf{rcv}\,x\,\mathsf{in}\,e \lhd \epsilon]^s_\epsilon \parallel j[i!\mathsf{invalid} \lhd \epsilon]^\circ_\epsilon \longrightarrow \tag{9}$$

$$i[\mathsf{rcv}\,x\,\mathsf{in}\,e \lhd \mathsf{invalid}]^s_\epsilon \parallel j[\mathsf{invalid} \lhd \epsilon]^\circ_\epsilon \longrightarrow \tag{10}$$

$$i[\mathsf{rcv}\,x\,\mathsf{in}\,e \lhd \mathsf{invalid}]^s_\epsilon \parallel j[\mathsf{invalid}]_\epsilon \tag{11}$$

Reduction (3) is derived using Slf, instantiating self for i, whereas reduction (4) describes the spawning of a new actor using Spw. (5) is a trivial application of Cmd where no free variables need to be substituted for j. This is followed by three function application reductions using App, (6), substituting the function variables x, y, z for n, m, i *resp.* Since m is not an integer, the evaluation of the expression $i!(n+m)$ generates a fault exception exit, (7), which is caught, (8), using rule Ctc. Local fault handling then sends invalid back to the actor spawning the process, *i.e.,* actor i, using Snd, (9) followed by the spawned process j terminating using rule Fin, (10).

The second actor, $i[\mathcal{C}[e_2] \lhd \epsilon]^s_\epsilon$, corresponding to a call to Program 1.2, may however exhibit a different reduction sequence.

$$i[\mathcal{C}[e_2] \lhd \epsilon]^s_\epsilon \longrightarrow^2 \tag{12}$$

$$i[\mathcal{C}[\mathsf{spw}\,\big(f\ (\mathsf{fun}\,(x,y,z)z!(x+y))\ n\ m\ i\ i\big)] \lhd \epsilon]^s_\epsilon \longrightarrow \tag{13}$$

$$\left(\begin{array}{l} i[j,\mathsf{rcv}\,x\,\mathsf{in}\,e \lhd \epsilon]^s_\epsilon \parallel \\ j[f\ (\mathsf{fun}\,(x,y,z)z!(x+y))\ n\ m\ i\ i \lhd \epsilon]^\circ_\epsilon \end{array}\right) \longrightarrow^6 \tag{14}$$

$$\left(\begin{array}{l} i[\mathsf{rcv}\,x\,\mathsf{in}\,e \lhd \epsilon]^s_\epsilon \parallel \\ j\left[\begin{array}{l} w{=}\mathsf{spw\_lnk} \\ \quad \big((\mathsf{fun}\,(x,y,z)z!x+y)\ n\ m\ i\big),\mathsf{trap}, \\ \mathsf{rcv}\,x\,\mathsf{in}\,\mathsf{if}\ x = \mathsf{end}(w,\mathsf{exit}) \\ \qquad\qquad \mathsf{then}\ i!\mathsf{invalid}\ \mathsf{else}\ \mathsf{ok} \end{array} \lhd \epsilon\right]^\circ_\epsilon \end{array}\right) \longrightarrow^2 \tag{15}$$

$$\left(\begin{array}{l} i[\mathsf{rcv}\,x\,\mathsf{in}\,e \lhd \epsilon]^s_\epsilon \parallel \\ j\left[\begin{array}{l} \mathsf{trap}, \\ \mathsf{rcv}\,x\,\mathsf{in}\,\mathsf{if}\ x = \mathsf{end}(h,\mathsf{exit}) \\ \qquad\qquad \mathsf{then}\ i!\mathsf{invalid}\ \mathsf{else}\ \mathsf{ok} \end{array} \lhd \epsilon\right]^\circ_h \\ \parallel h[(\mathsf{fun}\,(x,y,z)z!x+y)\ n\ m\ i \lhd \epsilon]^\circ_j \end{array}\right) \longrightarrow^3 \tag{16}$$

$$\left(\begin{array}{l} i[\mathsf{rcv}\,x\,\mathsf{in}\,e \lhd \epsilon]^s_\epsilon \parallel \\ j\left[\begin{array}{l} \mathsf{trap}, \\ \mathsf{rcv}\,x\,\mathsf{in}\,\mathsf{if}\ x = \mathsf{end}(h,\mathsf{exit}) \\ \qquad\qquad \mathsf{then}\ i!\mathsf{invalid}\ \mathsf{else}\ \mathsf{ok} \end{array} \lhd \epsilon\right]^\circ_h \\ \parallel h[i!n+m \lhd \epsilon]^\circ_j \end{array}\right) \tag{17}$$

The first two reductions, (12), are instantiations of self to i, whereas (13) is a spawn launching actor j. The six reductions that follow, (14), are function applications instantiating the variables x, y_1, y_2, y_3, z for the values $(\mathsf{fun}\,(x,y,z)z!(x+y))$, n, m, i, i *resp.* in the body of f at actor j, interleaved by an application of Cmd at actor i. Reductions (15) describe the spawning of a third actor, h, followed by the substitution of h for w using Cmd; note that the spawned actor is *linked* to actor j. The three reductions (16) is a series of function applications at actor h. Some of the reductions in (15) and (16) may interleave; they however still yield the same system (17).

$$\left(\begin{array}{l} i[\mathsf{rcv}\,x\,\mathsf{in}\,e \lhd \epsilon]^s_\epsilon \parallel \\ j\left[\begin{array}{l} \mathsf{trap}, \\ \mathsf{rcv}\,x\,\mathsf{in}\,\mathsf{if}\ x = \mathsf{end}(h,\mathsf{exit}) \\ \qquad\qquad \mathsf{then}\ i!\mathsf{invalid}\ \mathsf{else}\ \mathsf{ok} \end{array} \lhd \epsilon\right]^\circ_h \\ \parallel h[i!n+m \lhd \epsilon]^\circ_j \end{array}\right) \longrightarrow^2 \tag{18}$$

$$\left(\begin{array}{l} i[\mathsf{rcv}\,x\,\mathsf{in}\,e \lhd \epsilon]^s_\epsilon \parallel \\ j\left[\begin{array}{l} \mathsf{rcv}\,x\,\mathsf{in}\,\mathsf{if}\ x = \mathsf{end}(h,\mathsf{exit}) \\ \qquad\qquad \mathsf{then}\ i!\mathsf{invalid}\ \mathsf{else}\ \mathsf{ok} \end{array} \lhd \epsilon\right]^\bullet_h \\ \parallel h[i!n+m \lhd \epsilon]^\circ_j \end{array}\right) \longrightarrow^2 \tag{19}$$

$$\left(\begin{array}{l} i[\mathsf{rcv}\,x\,\mathsf{in}\,e \lhd \epsilon]^s_\epsilon \parallel \\ j\left[\begin{array}{l} \mathsf{rcv}\,x\,\mathsf{in}\,\mathsf{if}\ x = \mathsf{end}(h,\mathsf{exit}) \\ \qquad\qquad \mathsf{then}\ i!\mathsf{invalid}\ \mathsf{else}\ \mathsf{ok} \end{array} \lhd \epsilon\right]^\bullet_h \\ \parallel h[\mathsf{exit}]_j \end{array}\right) \longrightarrow \tag{20}$$

[6] The multi-argument function $\mathsf{fun}\,(x_1,\ldots x_n)c$ is shorthand for $\mathsf{fun}\,(x_1)\big(\ldots\big(\mathsf{fun}\,(x_n)c\big)\big)$ whereas the expression list $e_1\ e_2\ \ldots\ e_n$ denotes the applications $\big((\ldots(e_1\,e_2)\ldots)e_n\big)$.

[7] We avoid using lists in our modelling so as to keep the exposition self-contained.

[8] Expression sequencing e, c is shorthand for $x = e, c$ whenever $x \notin fv(c)$.

$$\left(\begin{array}{l} i[\mathsf{rcv}\,x\,\mathsf{in}\,e \lhd \epsilon]^s_\epsilon \parallel h[\mathsf{exit}]_\epsilon \parallel \\ j\left[\begin{array}{l}\mathsf{rcv}\,x\,\mathsf{in}\,\mathsf{if}\,x = \mathsf{end}(h,\mathsf{exit}) \\ \quad \mathsf{then}\ i!\mathsf{invalid}\ \mathsf{else}\ \mathsf{ok}\end{array} \lhd \mathsf{end}(h,\mathsf{exit})\right]^\bullet_\epsilon \end{array}\right) \longrightarrow \tag{21}$$

$$\left(\begin{array}{l} i[\mathsf{rcv}\,x\,\mathsf{in}\,e \lhd \epsilon]^s_\epsilon \parallel h[\mathsf{exit}]_\epsilon \parallel \\ j\left[\begin{array}{l}\mathsf{if}\,\mathsf{end}(h,\mathsf{exit}) = \mathsf{end}(h,\mathsf{exit}) \\ \quad \mathsf{then}\ i!\mathsf{invalid}\ \mathsf{else}\ \mathsf{ok}\end{array} \lhd \epsilon\right]^\bullet_\epsilon \end{array}\right) \longrightarrow \tag{22}$$

$$\begin{array}{l} i[\mathsf{rcv}\,x\,\mathsf{in}\,e \lhd \epsilon]^s_\epsilon \parallel h[\mathsf{exit} \lhd \epsilon]^\circ_\epsilon \parallel j[i!\mathsf{invalid} \lhd \epsilon]^\bullet_\epsilon \longrightarrow^2 \\ i[\mathsf{rcv}\,x\,\mathsf{in}\,e \lhd \mathsf{invalid}]^s_\epsilon \parallel h[\mathsf{exit}]_\epsilon \parallel j[\mathsf{invalid}]_\epsilon \end{array} \tag{23}$$

The next reduction step to be taken is however critical. The system shown in (17) may execute the trapping command at actor j, (18), making it a *system process* that traps exits. Thus, when the computation at actor h fails and terminates, (19), actor j can trap the exit message in its mailbox, (20), using TRMT. The termination message can then be read by j using RCV, (21), processed, (22), and the failure can be reported back to actor i, (23), as in (11) discussed earlier.

$$\left(\begin{array}{l} i[\mathsf{rcv}\,x\,\mathsf{in}\,e \lhd \epsilon]^s_\epsilon \parallel \\ j\left[\begin{array}{l}\mathsf{trap}, \\ \mathsf{rcv}\,x\,\mathsf{in}\,\mathsf{if}\,x = \mathsf{end}(h,\mathsf{exit}) \\ \qquad\qquad \mathsf{then}\ i!\mathsf{invalid}\ \mathsf{else}\ \mathsf{ok}\end{array} \lhd \epsilon\right]^\circ_h \\ \parallel h[i!n+m \lhd \epsilon]^\circ_j \end{array}\right) \longrightarrow \tag{24}$$

$$\left(\begin{array}{l} i[\mathsf{rcv}\,x\,\mathsf{in}\,e \lhd \epsilon]^s_\epsilon \parallel \\ j\left[\begin{array}{l}\mathsf{trap}, \\ \mathsf{rcv}\,x\,\mathsf{in}\,\mathsf{if}\,x = \mathsf{end}(h,\mathsf{exit}) \\ \qquad\qquad \mathsf{then}\ i!\mathsf{invalid}\ \mathsf{else}\ \mathsf{ok}\end{array} \lhd \epsilon\right]^\circ_h \\ \parallel h[\mathsf{exit} \lhd \epsilon]^\circ_j \end{array}\right) \longrightarrow \tag{25}$$

$$\begin{array}{l}\left(\begin{array}{l} i[\mathsf{rcv}\,x\,\mathsf{in}\,e \lhd \epsilon]^s_\epsilon \parallel \\ j\left[\begin{array}{l}\mathsf{trap}, \\ \mathsf{rcv}\,x\,\mathsf{in}\,\mathsf{if}\,x = \mathsf{end}(h,\mathsf{exit}) \\ \qquad\qquad \mathsf{then}\ i!\mathsf{invalid}\ \mathsf{else}\ \mathsf{ok}\end{array} \lhd \epsilon\right]^\circ_h \\ \parallel h[\mathsf{exit}]_\epsilon \end{array}\right) \longrightarrow \\ \quad i[\mathsf{rcv}\,x\,\mathsf{in}\,e \lhd \epsilon]^s_\epsilon \parallel j[\mathsf{exit}]_\epsilon \parallel h[\mathsf{exit}]_\epsilon \end{array} \tag{26}$$

Alternatively, however, system (17) may have a different interleaved execution of its parallel actors, exhibiting behaviour that differs from that reached in (11). In particular, before actor j can execute the trap command enabling it to trap exits, the spawned actor h may terminate abnormally, (24) and (25). Since actor h is linked to actor j, this may cause j to terminate abnormally with it, (26), instead of handling the exit as in the previous reduction sequence, thereby blocking actor i.

Program 3.1 Remote Error Handling in Erlang

```
fun1(X,Y,Pid) -> Pid ! X + Y.

ff_wrap(F,Args,Pid) ->
   process_flag(trap_exit,true),
   P = spawn_link(?MODULE,F,Args),
   receive
      {'EXIT',P,{badarith,Stack}}
                                   -> Pid ! invalid
      {'EXIT',P,normal} -> ok;
   end.
```

Identifying this alternative reduction sequence allows us not only to identify potential problems should we decide to refactor Program 1.1 with 1.2, but also helps us determine the root of the problem. In this case, the problem is caused by a race condition at the execution point corresponding to the system (17). We can avoid this by modifying Program 1.2 to the one show in Program 3.1, which swaps the order of commands `P = spawn_link(?MODULE,F,Args)` and `process_flag(trap_exit,true)`; this ensures that the process is spawned *after* the spawning process is trapping exits.

3.1 A Tool Implementation

The Erlang semantics designed in Section 2 was animated through a prototype evaluator for our calculus, implemented in Haskell; the implementation was based on parser combinators which allows us to easily extend it to handle the modelling of more Erlang constructs. One of the aims of the evaluator is that of assisting the automation of the discussion held in Section 3, by going through all the possible interleavings that the program may take, thus, identifying all the different ways in which the program may behave. Once the different behaviours are identified, it returns the sequences of reduction steps that cause the input program to behave in a particular behaviour.

4. EVALUATION

Since the semantics of Section 2 was defined post-hoc, it was important to ensure that it does indeed correspond to the actual behaviour of the Erlang constructs it is trying to model. The tool described in Section 3.1 was used as a vehicle for this correctness analysis, since it was built to reflect the semantics directly. The correctness of the model was therefore assessed using the strategy described in Figure 2.

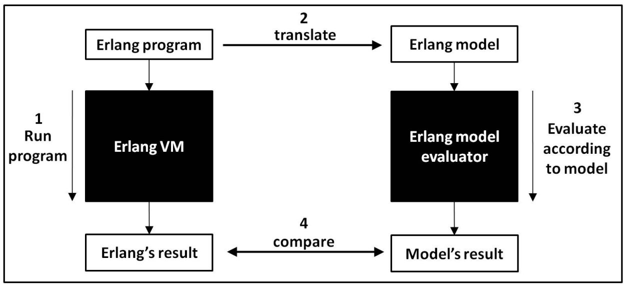


Figure 2: Evaluation Strategy

Considering a number of Erlang programs dealing with error handling, the correctness of the model was analysed by:

1. Running the program on the Erlang VM, using the inbuilt tracer function to record the intermediary steps of the execution.

2. Translate the same Erlang program in terms of the model.

3. Execute the the translated Erlang program on the Evaluator, which returns a documented sequences of reduction steps describing the different ways the input program may be evaluated according to the defined model.

4. Ensure that the recorded execution on the Erlang VM corresponds to one of the possible reduction sequences documented by the Evaluator.

Whenever the Evaluator tool returned a number of differing possible outcomes for a particular translated program, we also evaluated whether these reduction sequences do indeed correspond to possible executions over the Erlang VM. This required us to analyse the different sequences of reductions, identify what actor interleaving let to the different behaviour and then inject `sleep(time)` calls at strategic points in the original code so as to induce a certain scheduling order that matches the reduction sequence reported.

Our empirical tests so far supported our conjecture that the model corresponded to the behaviour on the Erlang VM, with the exception of certain simplifying shortcuts we consciously chose to adopt in our model, so as to keep the calculus and the tool analysis more tractable.

5. RELATED AND FUTURE WORK

The ultimate aim of our work is that of developing a behavioural theory for program equivalence for (a subset of) Erlang, which would help us understand better the semantics of the language and assist the construction of refactoring tools such as [11, 9]. At present, our analysis is performed under a "closed-world" setting, whereby a system of actors is analysed in isolation. A proper theory of equivalence will however require us to consider congruences of some sort, which will probably entail a shift towards an "open-world" setting, where reductions consider also interactions with the context. We also plan to use the semantics to define formally correct runtime-verification tools for Erlang[5]. Much work still needs to be done in this regard.

For a start, we need to compare our work with existing work on formalising the language [6, 4, 12, 13]. It appears that our formalisation is more lightweight than [4, 12, 13], which consists of a three-layered semantics, *i.e.,* expressions, processes/actors and nodes. Our semantics ignores distribution, *i.e.,* nodes and is more fine tuned to study error handling; despite using a single-layer semantics it still seems to retain the essential features of Erlang concurrent computation. Technically, the developments are slightly different as well, since ours is given in terms of a reduction semantics, whereas existing work relies on labelled-transition systems.

Crucially however, the main focus of [12, 4] seems to be that of understanding existing Erlang VM implementations, unravelling subtle discrepancies between processes communications in a distributed setting; [13] is a follow-up, proposing ways how to simplify this distributed semantics. The precursor to [13, 12, 4] is [6] which is perhaps the closest in spirit to our work. In his thesis, Fredlund focussed on developing a logical framework for reasoning about Erlang programs in terms of the modal μ-calculus[8]; program equivalence is not central to this work. The calculus developed there does not address the study of the different modes of error handling in Erlang either; in fact, local error handling through the *try-catch* construct is omitted.

6. CONCLUSION

We have reported on preliminary investigations of the Erlang language and the behaviour of its constructs, focussing on the various way how error handling can be carried out in the language. We developed a calculus that can model Erlang programs and showed how this can be used to determine differing behaviour between seemingly equivalent programs. We also outline the development of a tool based on this calculus used to automate the process of program analysis and to validate the correctness of the calculus developed.

7. REFERENCES

[1] Gul Agha. *Actors: a model of concurrent computation in distributed systems.* PhD thesis, MIT, 1986.

[2] Joe Armstrong. *Programming Erlang: Software for a Concurrent World.* Pragmatic Bookshelf, 2007.

[3] Francesco Cesarini and Simon Thompson. *Erlang Programming.* O'Reilly, 2009.

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

[5] Christian Colombo, Adrian Francalanza, and Rudolph Gatt. ELarva:A Monitoring tool for Erlang. In *Runtime Verification (RV) 2011*, volume 7186 of *LNCS*, pages 370–374. Springer, 2012.

[6] Lars-Ake Fredlund. *A Framework for Reasoning about Erlang Code.* PhD thesis, Royal Institute of Technology, Stockholm, Sweden, 2001.

[7] Jim Gray. Why do computers stop and what can be done about it? Technical report, Tandem Computers, 1985.

[8] Dexter Kozen. Results on the propositional μ-calculus. *TCS*, 27(3):333 – 354, 1983. Special Issue (ICALP) 1982.

[9] Huiqing Li and Simon Thompson. Refactoring support for modularity maintenance in erlang. In *Proceedings of the 2010 10th IEEE Working Conference on Source Code Analysis and Manipulation*, SCAM '10, pages 157–166, Washington, DC, USA, 2010. IEEE Computer Society.

[10] J. H. Nyström, P. W. Trinder, and D. J. King. High-level distribution for the rapid production of robust telecoms software: comparing c++ and erlang. *Concurr. Comput. : Pract. Exper.*, 20:941–968, June 2008.

[11] Konstantinos Sagonas and Thanassis Avgerinos. Automatic refactoring of Erlang programs. In *Proceedings of the Eleventh International ACM SIGPLAN Symposium on Principles and Practice of Declarative Programming*, pages 13–24, New York, NY, USA, September 2009. ACM.

[12] Hans Svensson and Lars-Ake Fredlund. A more accurate semantics for distributed erlang. In *Proceedings of the 2007 SIGPLAN workshop on ERLANG Workshop*, ERLANG '07, pages 43–54, New York, NY, USA, 2007. ACM.

[13] Hans Svensson, Lars-Ake Fredlund, and Clara Benac Earle. A unified semantics for future erlang. In *Proceedings of the 9th ACM SIGPLAN workshop on Erlang*, Erlang '10, pages 23–32, New York, NY, USA, 2010. ACM.