Available online at www.sciencedirect.com
ScienceDirect
Electronic Notes in Theoretical Computer Science
ELSEVIER
Electronic Notes in Theoretical Computer Science 176 (2007) 147–163
www.elsevier.com/locate/entcs

Abstraction and Model Checking of Core Erlang Programs in Maude

Martin Neuhäußer[1] Thomas Noll[2]

Software Modeling and Verification Group
RWTH Aachen University
D–52056 Aachen, Germany
Fax: +49 241 80 22217

Abstract

This paper provides a contribution to the formal verification of programs written in the concurrent functional programming language ERLANG, which is designed for telecommunication applications. We present a formalization of this language in the Rewriting Logic framework, employing equations for defining abstraction mappings on the state space of the system. Moreover we give a sketch of an implementation in the MAUDE system, and demonstrate the use of its model checker to verify simple system properties.

Keywords: Formal verification, functional programming, ERLANG, MAUDE, rewriting logic

1 Introduction

In this paper we address the software verification issue in the context of the functional programming language ERLANG [1], which was developed by Ericsson corporation to address the complexities of developing large–scale programs within a concurrent and distributed setting. Our interest in this language is twofold. On the one hand, it is often and successfully used in the design and implementation of telecommunication systems. On the other hand, its relatively compact syntax and its clean semantics support the application of formal reasoning methods.

Here we try to employ fully–automatic *model–checking techniques* [5] to establish correctness properties of communication systems implemented in ERLANG. While simulation and testing explore some of the possible executions of a system, model checking conducts an exhaustive exploration of all its behaviors. In this paper we concentrate on the first part of the verification procedure, the construction of the (transition–system) model to be checked.

[1] Email: neuhaeusser@cs.rwth-aachen.de
[2] Email: noll@cs.rwth-aachen.de

doi:10.1016/j.entcs.2007.06.013

More concretely, our approach is based on Core Erlang [3,4], an intermediate language being used in the Erlang compiler which, however, is very close to the original language. We formally describe its semantics employing the *Rewriting Logic* framework, which was proposed in [9] as a unified semantic framework for concurrency. It has proven to be an adequate modeling formalism for many concrete specification and programming languages [8]. In this approach the state of a system is represented by an equivalence class of terms modulo a given set of equations, and transitions correspond to rewriting operations on the representatives. Hence Rewriting Logic supports both the definition of programming formalisms and, by employing (equational) term rewriting methods, the execution or simulation of concrete systems. We will see that the equations can be used to define abstraction mappings which reduce the state space of the system.

Furthermore we will show that by employing an executable implementation of the Rewriting Logic framework, Maude [6], it is possible to automatically derive the transition system of a given Erlang program, and to verify its properties using the Maude model checker.

The remainder of this paper is organized as follows. Section 2 presents the Core Erlang programming language by sketching its syntactic constructs and their intuitive meaning. Section 3 briefly introduces the Rewriting Logic Framework. Finally, Sections 4 and 5 constitute the main part of this paper in which the Rewriting Logic specification of the operational semantics of Core Erlang and its implementation in Maude are studied.

2 Core Erlang

Erlang/OTP is a programming platform providing the necessary functionality for programming open distributed (telecommunication) systems: the functional language Erlang with support for communication and concurrency, and the OTP (Open Telecom Platform) middleware providing ready–to–use components (libraries) and services such as e.g. a distributed data base manager, support for "hot code replacement", and design guidelines for using the components.

Today many commercially available products offered by Ericsson are at least partly implemented in Erlang. The software of such products is typically organized into many, relatively small source modules, which at runtime are executed as a dynamically varying number of processes operating in parallel and communicating through asynchronous message passing. The highly concurrent and dynamic nature of such software makes it particularly hard to debug and test by manual methods.

In the following we consider the core version of the Erlang programming language which has been introduced in [3], and which is used as an intermediate language in the Erlang compiler. It supports the implementation of dynamic networks of processes operating on data types such as atomic constants (atoms), integers, lists, tuples, and process identifiers (pids), using asynchronous, call–by–value communication via unbounded ordered message queues called mailboxes. Full Erlang has several additional features such as distribution of processes (onto nodes),

$$
\begin{array}{rcl}
\mathit{Module} & ::= & \underline{\texttt{module}}\ \mathit{atom}\ \texttt{[}\mathit{Fname}_{i_1}\texttt{,}\ldots\texttt{,}\mathit{Fname}_{i_k}\texttt{]} \\
 & & \underline{\texttt{attributes}}\ \texttt{[}\mathit{atom}_1\texttt{=}\mathit{Const}_1\texttt{,}\ldots\texttt{,}\mathit{atom}_m\texttt{=}\mathit{Const}_m\texttt{]} \\
 & & \mathit{Fdef}_1 \ldots \mathit{Fdef}_n \\
\mathit{Fdef} & ::= & \mathit{FunName}\ \texttt{=}\ \mathit{Fun} \\
\mathit{FunName} & ::= & \mathit{atom}\ \texttt{/}\ \mathit{integer} \\
\mathit{Const}\,(c) & ::= & \mathit{Lit} \mid \texttt{[}\ \mathit{Const}_1\ \texttt{|}\ \mathit{Const}_2\ \texttt{]} \mid \texttt{\{}\ \mathit{Const}_1\ \texttt{,}\ \ldots\ \texttt{,}\ \mathit{Const}_n\ \texttt{\}} \mid \mathit{Fun} \\
\mathit{Val}\,(v) & ::= & \mathit{Const} \mid \texttt{<}\ \mathit{Const}_1\ \texttt{,}\ \ldots\ \texttt{,}\ \mathit{Const}_n\ \texttt{>} \\
\mathit{Lit} & ::= & \mathit{integer} \mid \mathit{float} \mid \mathit{atom} \mid \mathit{char} \mid \mathit{string} \mid \texttt{[]} \\
\mathit{Fun} & ::= & \underline{\texttt{fun}}\ \texttt{(}\ \mathit{var}_1\ \texttt{,}\ \ldots\ \texttt{,}\ \mathit{var}_n\ \texttt{)}\ \texttt{->}\ \mathit{Exps} \\
\mathit{Exps} & ::= & \mathit{Exp} \mid \texttt{<}\ \mathit{Exp}_1\ \texttt{,}\ \ldots\ \texttt{,}\ \mathit{Exp}_n\ \texttt{>} \\
\mathit{Exp}\,(e) & ::= & \mathit{var} \mid \mathit{Fname} \mid \mathit{Lit} \mid \mathit{Fun} \\
 & \mid & \texttt{[}\ \mathit{Exps}_1\ \texttt{|}\ \mathit{Exps}_2\ \texttt{]} \mid \texttt{\{}\ \mathit{Exps}_1\ \texttt{,}\ \ldots\ \texttt{,}\ \mathit{Exps}_n\ \texttt{\}} \\
 & \mid & \underline{\texttt{let}}\ \mathit{Vars}\ \texttt{=}\ \mathit{Exps}_1\ \ \underline{\texttt{in}}\ \mathit{Exps}_2 \mid \underline{\texttt{do}}\ \mathit{Exps}_1\ \mathit{Exps}_2 \\
 & \mid & \underline{\texttt{letrec}}\ \mathit{Fdef}_1 \ldots \mathit{Fdef}_n\ \underline{\texttt{in}}\ \mathit{Exps} \\
 & \mid & \underline{\texttt{apply}}\ \mathit{Exps}_0\ \texttt{(}\ \mathit{Exps}_1\ \texttt{,}\ \ldots\texttt{,}\ \mathit{Exps}_n\ \texttt{)} \\
 & \mid & \underline{\texttt{call}}\ \mathit{Exps}_{n+1}\ \texttt{:}\ \mathit{Exps}_{n+2}\ \texttt{(}\ \mathit{Exps}_1\ \texttt{,}\ \ldots\texttt{,}\ \mathit{Exps}_n\ \texttt{)} \\
 & \mid & \underline{\texttt{primop}}\ \mathit{atom}\ \texttt{(}\ \mathit{Exps}_1\ \texttt{,}\ \ldots\texttt{,}\ \mathit{Exps}_n\ \texttt{)} \\
 & \mid & \underline{\texttt{try}}\ \mathit{Exps}_1\ \underline{\texttt{catch}}\ \texttt{(}\ \mathit{var}_1\ \texttt{,}\ \mathit{var}_2\ \texttt{)}\ \texttt{->}\ \mathit{Exps}_2 \\
 & \mid & \underline{\texttt{case}}\ \mathit{Exps}\ \underline{\texttt{of}}\ \mathit{Clause}_1 \ldots \mathit{Clause}_n\ \underline{\texttt{end}} \\
 & \mid & \underline{\texttt{receive}}\ \mathit{Clause}_1 \ldots \mathit{Clause}_n\ \ \underline{\texttt{after}}\ \mathit{Exps}_1\ \texttt{->}\ \mathit{Exps}_2 \\
\mathit{Vars} & ::= & \mathit{var} \mid \texttt{<}\ \mathit{var}_1\ \texttt{,}\ \ldots\ \texttt{,}\ \mathit{var}_n\ \texttt{>} \\
\mathit{Clause}(cl) & ::= & \mathit{Pats}\ \underline{\texttt{when}}\ \mathit{Exps}_1\ \texttt{->}\ \mathit{Exps}_2 \\
\mathit{Pats} & ::= & \mathit{Pat} \mid \texttt{<}\ \mathit{Pat}_1\ \texttt{,}\ \ldots\ \texttt{,}\ \mathit{Pat}_n\ \texttt{>} \\
\mathit{Pat}\,(p) & ::= & \mathit{var} \mid \mathit{Lit} \mid \texttt{[}\ \mathit{Pat}_1\ \texttt{|}\ \mathit{Pat}_2\ \texttt{]} \mid \texttt{\{}\ \mathit{Pat}_1\ \texttt{,}\ \ldots\ \texttt{,}\ \mathit{Pat}_n\ \texttt{\}} \mid \mathit{var}\ \texttt{=}\ \mathit{Pat}
\end{array}
$$

Fig. 1. Syntax of CORE ERLANG

```
module 'locker' ['start'/0,'locker'/0,'client'/1] attributes []

'start'/0 = fun () ->
    let LockerPid = call 'erlang':'self'() in do
        call 'erlang':'spawn'('locker', 'client', [LockerPid]) do
          call 'erlang':'spawn'('locker', 'client', [LockerPid])
          apply 'locker'/0()

'locker'/0 = fun() ->
    receive
      {'req',Client} when 'true' -> do
        call 'erlang':'!' (Client, 'ok')
        receive
          {'rel',From} when call 'erlang':'=:=' (From, Client) ->
            apply 'locker'/0()
        after 'infinity' -> 'true'
    after 'infinity' -> 'true'

'client'/1 = fun (Locker) ->
    let Me = call 'erlang':'self'() in do
        call 'erlang':'!'(Locker, {'req',Me}) do
          receive
            'ok' when 'true' -> call 'erlang':'!'(Locker, {'rel',Me})
          after 'infinity' -> 'true'
        apply 'client'/1 (Locker)
end
```

Fig. 2. Resource locker in CORE ERLANG

and support for robust programming and for interoperation with non–ERLANG code written in, e.g., C or Java.

The syntax of CORE ERLANG is defined by the context–free grammar [3] in Figure 1. For further explanations, please refer to [3,4].

As an introductory example we consider a short program which implements a simple resource locker, i.e., an arbiter which, upon receiving corresponding requests from client processes (two in this case), grants access to a single resource. Its code is given in Figure 2.

Any CORE ERLANG program consists of a set of modules. Each module is

[3] To simplify notation, let the placeholder symbols c, v, e, etc. denote elements of the language generated by the respective nonterminals. Moreover, let a and x denote atoms and variables, respectively.

identified by a name, followed by a list of exported functions, and a list of function definitions. In our example the system is defined in one module named `locker`. It is initialized using the `start` function. By calling the `spawn` builtin function, the latter generates two additional processes both running the `client` function from the `locker` module. Here the `self` builtin function returns the process identifier (pid) of the locker process, which is then passed as an argument to the clients such that these are enabled to communicate with the locker.

The locker process runs the `locker` function in a non–terminating loop. It employs the `receive` construct to check whether a request has arrived. The latter is expected to be a pair composed of a `req` tag and a client process identifier (which is matched by the variable `Client`). The `after` clause can be used to specify the behavior of the process when no matching request arrives within a certain time limit; here it is deactivated by giving the `infinity` atom as the timeout value.

The client is then granted access to the resource by sending an `ok` flag. Finally, after receiving the `rel` (release) message from the respective client, the locker returns to its initial state.

A client process exhibits the complementary behavior. By issuing a request, it demands access to the resource. Here again the `self` builtin function is used to determine the pid of the client process, which is then used by the locker process as a handle to the client. After receiving the `ok` message it accesses the resource, and releases it afterwards.

The desirable correctness properties of such a system are straightforward:

no deadlock: there exists no cyclic chain of processes waiting for each other to continue, i.e., the locker should always be enabled to receive a new request or a release,

mutual exclusion: no two clients should gain access to the resource at the same time, and

no starvation: all clients enabled to enter the critical section should eventually be granted their demanded access.

Later we will exemplarily see how to check the second property by constructing the transition system of the above program.

3 The Rewriting Logic Framework

The Rewriting Logic framework has been presented by J. Meseguer in [9]. An introduction to this approach together with an extensive bibliography can be found in [8].

Rewriting Logic is intended to serve as a unifying mathematical model and uses notions from rewrite systems over equational theories. It separately describes the static and the dynamic aspects of a concurrent system. More exactly, it distinguishes the laws describing the structure of the states of the system from the rules which specify its possible transitions. The two aspects are respectively formalized as a set of equations and as a (conditional) term rewriting system. Both structures operate

on states, represented as (equivalence classes of) Σ–terms where Σ is the signature of the specification language under consideration.

More concretely, in Meseguer's approach the syntax of Rewriting Logic is given by a *rewrite theory* [4] $\mathfrak{T} = (\Sigma, E, R)$ where

- Σ is a signature, i.e., a ranked alphabet of function symbols,
- $E \subseteq T_\Sigma(X) \times T_\Sigma(X)$ is a finite set of equations over the set $T_\Sigma(X)$ of Σ–terms with variables from a given set X, and
- R is a finite set of *(conditional) transition rules* of the form

$$\frac{c_1 \longrightarrow d_1 \ \ldots \ c_k \longrightarrow d_k}{l \longrightarrow r}$$

With regard to the semantics of Rewriting Logic, Meseguer defines that a rewrite theory $\mathfrak{T}$ *entails* a *sequent* $[s]_E \longrightarrow [t]_E$ and writes

$$\mathfrak{T} \vdash [s]_E \longrightarrow [t]_E$$

if this sequent can be obtained by a finite number of applications of certain *rules of deduction* which specify how to apply the above transition rules. In this way it is possible to reason about concurrent systems whose states are represented by terms and which are evolving by means of transitions. Here, the states are structured according to the signature and equations are used to identify terms which differ only in their syntactic representation. Later we will see that they can also be employed to define abstraction mappings on the state space.

It is a fact, however, that (conditional) term rewriting modulo equational theories is generally too complex or even undecidable. Hence it is not possible to admit arbitrary equations in E. Following the ideas of P. Viry in [11], we therefore propose to decompose E into a set of directed equations (that is, a term rewriting system), ER, and into a set A expressing associativity and commutativity of certain binary operators in Σ. Given that ER is terminating modulo A, rewriting by R modulo E can be implemented by a combination of normalizing by ER and rewriting by R, both modulo A. Here the steps induced by R represent the actual state transitions of the system while the reductions defined by ER have to be considered as internal, non–observable computations.

4 Operational Semantics of Core Erlang

Given a Core Erlang program and an initial expression, we define its transition–system semantics by first considering only local evaluation steps. In Section 4.2 we then extend our transitions in order to capture the concurrent semantics, i.e., the semantics of evaluations which are afflicted with side effects. For an in–depth description of the small–step operational semantics including a formal definition of error handling in Core Erlang, see [10].

[4] In Maude, the Rewriting Logic specification is parameterized by a membership equational logic theory with its many–kinded signatures, cf. [2].

$$
\begin{array}{lll}
r[\cdot] ::= & \cdot \mid \texttt{[}r[\cdot]\texttt{|}e\texttt{]} \mid \texttt{[}v\texttt{|}r[\cdot]\texttt{]} & \\
\mid & \texttt{\{}v_1,\ldots,v_{k-1},r[\cdot],e_{k+1},\ldots,e_n\texttt{\}} & (1 \leq k \leq n) \\
\mid & \texttt{<}v_1,\ldots,v_{k-1},r[\cdot],e_{k+1},\ldots,e_n\texttt{>} & (1 \leq k \leq n) \\
\mid & \underline{\texttt{let}}\ \texttt{<}x_1,\ldots,x_n\texttt{>}\ \texttt{=}\ r[\cdot]\ \underline{\texttt{in}}\ e & (n \in \mathbb{N}) \\
\mid & \underline{\texttt{case}}\ r[\cdot]\ \underline{\texttt{of}}\ cl_1,\ldots,cl_n\ \underline{\texttt{end}} & (n \in \mathbb{N}) \\
\mid & \underline{\texttt{receive}}\ cl_1,\ldots,cl_n\ \underline{\texttt{after}}\ r[\cdot]\ \texttt{->}\ e & (n \in \mathbb{N}) \\
\mid & \underline{\texttt{do}}\ r[\cdot]\ e & \\
\mid & \underline{\texttt{apply}}\ r[\cdot](e_1,\ldots,e_n) & (n \in \mathbb{N}) \\
\mid & \underline{\texttt{apply}}\ f\,(c_1,\ldots,c_{k-1},r[\cdot],e_{k+1},\ldots,e_n) & (1 \leq k \leq n) \\
\mid & \underline{\texttt{call}}\ r[\cdot]:e_{n+1}\,(e_1,\ldots,e_n) & (n \in \mathbb{N}) \\
\mid & \underline{\texttt{call}}\ a_1:r[\cdot]\,(e_1,\ldots,e_n) & (n \in \mathbb{N}) \\
\mid & \underline{\texttt{call}}\ a_1:a_2\,(c_1,\ldots,c_{k-1},r[\cdot],e_{k+1},\ldots,e_n) & (1 \leq k \leq n) \\
\mid & \underline{\texttt{primop}}\ a(c_1,\ldots,c_{k-1},r[\cdot],e_{k+1},\ldots,e_n) & (1 \leq k \leq n)
\end{array}
$$

Fig. 3. Reduction contexts of Core Erlang expressions

4.1 Sequential Semantics

Let Exp denote the set of valid Core Erlang expressions according to Figure 1 and $\mathrm{fv}(e)$ denote the sequence of variables having at least one free occurrence in term e. We then formalize the semantics of closed Core Erlang expressions, i.e., expressions without free variables, by an associated transition system T_e:

Definition 4.1 Let $e_0 \in Exp$ and $\mathrm{fv}(e_0) = \emptyset$. The *associated transition system* is $T_e = (E, e_0, Act_e, \rightarrow_e)$ where $E := \{e \in Exp \mid \mathrm{fv}(e) = \emptyset\} \uplus \{\bot\}$ denotes the set of states with a distinguished initial expression e_0 and $\rightarrow_e \subseteq E \times Act_e \times E$ denotes the transition relation. Transitions are defined according to the following inference rules, and are labeled by actions from the set Act_e where $\tau \in Act_e$ denotes an unobservable action and the other labels represent the observable evaluation steps. $\bot$ denotes an undefined value that arises from errors occurring during expression evaluation.

The standard implementation of Core Erlang employs a leftmost–innermost evaluation strategy. To formalize argument evaluation, we use the concept of reduction contexts that was first introduced in [7]: intuitively, a reduction context is a Core Erlang term with a placeholder symbol "$\cdot$" in it, which identifies the subterm where the next evaluation step takes place in case the placeholder is substituted by a reducible expression. Formally, the set of reduction contexts is defined by the context–free grammar in Figure 3. Let Ctx denote the set of reduction contexts. The following inference rule then formalizes the leftmost innermost evaluation strategy:

Definition 4.2 Let $e, e' \in Exp, \alpha \in Act_e$ and $r \in Ctx$ such that there exist no $\tilde{e} \in Exp \setminus Val$ and $\tilde{r} \in Ctx \setminus \{\cdot\}$ such that $e = \tilde{r}\,[\tilde{e}]$. Then, the following inference rule is applicable:

$$\frac{e \xrightarrow{\alpha}_e e' \quad r \neq \cdot}{r[e] \xrightarrow{\alpha}_e r[e']}\,(\text{Context})$$

By the conditions imposed in Definition 4.2, the (Context) rule is only applicable wrt. a maximal reduction context, i.e., e can be evaluated directly without any further descent into its subterms. Using this concept, the semantics of the sequencing operator is captured by the following inference rules:

$$\frac{e \xrightarrow{\alpha}_e e'}{\underline{\texttt{do}}\ r[e]\ e_2 \xrightarrow{\alpha}_e \underline{\texttt{do}}\ r[e']\ e_2}\,(\text{Context}) \qquad \frac{}{\underline{\texttt{do}}\ v\ e_2 \xrightarrow{\tau}_e e_2}\,(\text{Seq})$$

According to Definition 4.2, the iterated application of the first rule[5] evaluates the first subexpression. As soon as it is completely evaluated, the second rule becomes applicable which formalizes the sequencing semantics in that the result is discarded and evaluation continued with the second subexpression.

4.1.1 Pattern Matching Semantics

To formalize the semantics of CORE ERLANG pattern matching, substitutions are used to syntactically replace free occurrences of variables (taken from the set *Var*) by their respective values.

Definition 4.3 A *substitution* is a partial mapping $\sigma : Var \uplus FunName \to Const$. $[x_1 \mapsto c_1, \ldots, x_n \mapsto c_n]$ denotes the finite substitution where x_i is replaced by the constant c_i for $1 \leq i \leq n$. Substitutions are extended to arbitrary CORE ERLANG expressions and clauses[6]:

- $x\sigma := \begin{cases} c_i & \text{if } \sigma(x) = c_i \\ x & \text{if } \sigma(x) = \bot \end{cases}$
- $\{e_1, \ldots, e_n\}\ \sigma := \{e_1\sigma, \ldots, e_n\sigma\}$
- $[e_1 | e_2]\ \sigma := [e_1\sigma | e_2\sigma]$
- $(\underline{\texttt{let}}\ \texttt{<}x_1, \ldots, x_n\texttt{>}\ \texttt{=}\ e\ \underline{\texttt{in}}\ e_2)\ \sigma \quad := \quad \underline{\texttt{let}}\ \texttt{<}x_1, \ldots, x_n\texttt{>}\ \texttt{=}\ e\sigma\ \underline{\texttt{in}}\ e_2\sigma'$
  where $\sigma' : Var \uplus FunName \to Const : x \mapsto \begin{cases} \sigma(x) & \text{if } x \notin \{x_1, \ldots, x_n\} \\ \bot & \text{otherwise} \end{cases}$

The pattern matching semantics is formalized using these syntactic substitutions: A clause p `when` g `->` e matches a value v if (i) a substitution σ exists such that $p\sigma = v$ holds and (ii) the guard expression $g\sigma$ evaluates to `'true'`. Formally, this is captured by the following definition.

Definition 4.4 Let $p \in Pat$, $e, g \in Exp$ and $v \in Val$. Then

$$\mathsf{match} : Val \times Clause \to Exp \uplus \{\bot\} :$$
$$(v, p\ \underline{\texttt{when}}\ g\ \texttt{->}\ e) \mapsto \begin{cases} e\sigma & \text{if } \exists\sigma.\ (v = p\sigma \wedge g\sigma \longrightarrow_e^* \texttt{'true'}) \\ \bot & \text{otherwise} \end{cases}$$

The CORE ERLANG `case` operator branches control according to a given value:

$$\frac{\exists i.\ (\mathsf{match}(v, cl_i) = e' \wedge \forall j < i.\ \mathsf{match}(v, cl_j) = \bot)}{\underline{\texttt{case}}\ v\ \underline{\texttt{of}}\ cl_1\ \cdots\ cl_k\ \underline{\texttt{end}} \xrightarrow{\tau}_e e'}\ (\text{Case}_1).$$

The clauses cl_i with $1 \leq i \leq k$ are tested sequentially against the value v. Evaluation continues according to the first matching clause; if no such clause is found, a runtime error is generated:

$$\frac{\forall i \leq k.\ \mathsf{match}(v, cl_i) = \bot}{\underline{\texttt{case}}\ v\ \underline{\texttt{of}}\ cl_1 \cdots cl_k\ \underline{\texttt{end}} \xrightarrow{\texttt{exc('error','case\_clause')}}_e \bot}\ (\text{Case}_2)$$

4.1.2 Message Passing

Given a mailbox $q \in Const^*$ and a nonempty sequence of clauses $cl_1, \ldots, cl_k$, $k \geq 1$, the predicate $\mathsf{qmatch}(q, cl_1, \ldots, cl_k)$ holds iff at least one message in q matches one

[5] Possibly with different reduction contexts

[6] In this paper, we only give an incomplete definition and refer the reader to [10, p. 59ff].

of the clauses cl_i. Within the sequential part of the semantics, we cannot infer the contents of the process' mailbox. Therefore we nondeterministically guess one possible prefix $q \cdot c$. In the inference rule Rcv_1, c denotes the first message that matches at least one of the clauses cl_i: the condition $\neg\mathsf{qmatch}(q, cl_1, \ldots, cl_k)$ is fulfilled iff q does not contain a matching message; in addition, we assume a successful [7] evaluation of the `case` term which captures the pattern matching semantics wrt. the message c and the clauses $cl_1, \ldots, cl_k$:

$$\frac{\neg\mathsf{qmatch}(q, cl_1, \ldots, cl_k) \quad \texttt{case}\ c\ \texttt{of}\ cl_1 \ldots cl_k\ \texttt{end} \xrightarrow{\tau}_e e' \quad c_t \in \mathbb{N} \cup \{\texttt{'infinity'}\}}{\texttt{receive}\ cl_1 \cdots cl_k\ \texttt{after}\ c_t\ \texttt{->}\ e_t \xrightarrow{\mathsf{recv}(q,c)}_e e'} (\mathrm{Rcv}_1)$$

If no matching message is received within the time bound c_t, e_t is evaluated. According to our time–abstract model, this is formalized by nondeterminism:

$$\frac{\neg\mathsf{qmatch}(q, cl_1, \ldots, cl_k) \qquad c_t \in \mathbb{N}}{\texttt{receive}\ cl_1 \cdots cl_k\ \texttt{after}\ c_t\ \texttt{->}\ e_t \xrightarrow{\mathsf{timeout}(q)}_e e_t} (\mathrm{Rcv}_2)$$

4.1.3 Higher–Order Concepts

Function abstractions are treated as values and are applied to a sequence of arguments using the `apply` operator. To capture its semantics, we replace every free occurrence of an argument variable by the corresponding value:

$$\frac{\sigma := [x_1 \mapsto c_1, \ldots, x_n \mapsto c_n]}{\texttt{apply}\ \texttt{fun}(x_1, \ldots, x_n)\ \texttt{->}\ e(c_1, \ldots, c_n) \xrightarrow{\tau}_e e\sigma} (\mathrm{App}_1)$$

Note that the argument evaluation is specified implicitly by the reduction contexts defined in Figure 3.

The `letrec` operator supports on–the–fly declaration of local functions. Its semantics is formalized by the following rule:

$$\frac{\forall i \le m.\quad e_i' := \texttt{letrec} \ldots a_j/n_j\texttt{=fun}(\boldsymbol{x}_j)\texttt{->}\ e_j \ldots \texttt{in}\ e_i}{\texttt{letrec} \ldots a_j/n_j\texttt{=fun}(\boldsymbol{x}_j)\texttt{->}\ e_j \ldots \texttt{in}\ e \xrightarrow{\tau}_e e\left[\ldots, a_j/n_j \mapsto \texttt{fun}(\boldsymbol{x}_j)\texttt{->}\ e_j', \ldots\right]} (\mathrm{LRec})$$

The function names a_i/n_i are treated as variables that range over the special domain of function abstractions. Evaluation of a `letrec` expression yields a new binding whose scope reaches over e and $e_1, \ldots, e_m$. This extended scope is reflected in the semantics by propagating the `letrec` statements into the bodies of the function abstractions (cf. the definition of e_i' in the premise).

4.2 Concurrent Semantics

To reason about concurrent systems implemented in Core Erlang, we now lift the semantics of sequential expressions to the system level where also side effects, i.e., process spawning and communication, are considered.

Definition 4.5 The set of *processes* is given by $P := Exp \uplus \{\bot\} \times \mathbb{N} \times Const^* \times 2^{\mathbb{N}} \times \mathbb{B}$. A process is denoted by $(e, i, q, L, t) \in P$ where e is the expression to be evaluated, i is the process identifier, q the process' mailbox, L the set of linked processes, and t is a flag controlling exit behavior.

[7] Matching failures would lead to a non–τ–transitions.

To describe the possible behaviors of an entire system, we extend this definition to sets of processes:

Definition 4.6 A finite subset $S \in 2^P$ is called a *process system*. S is *well formed* if, for every $p, p_1, p_2 \in S$,

$$p_1 \neq p_2 \Rightarrow \mathsf{Pid}(p_1) \neq \mathsf{Pid}(p_2) \qquad \text{and} \qquad \mathsf{Links}(p) \subseteq \bigcup_{p' \in S, p \neq p'} \mathsf{Pid}(p')$$

Here, $\mathsf{Pid} : P \to \mathbb{N}$ and $\mathsf{Links} : P \to 2^{\mathbb{N}}$ denote the projection on the process identifier and the link component respectively; furthermore, the mapping Pid is extended to sets of processes in the natural way.

By considering well formed process systems as states of the modeled reactive system, we obtain the following transition–system semantics:

Definition 4.7 Let $p_0 \in P$ be an initial process. The *corresponding transition system* is defined as $T_s = (\mathcal{S}, S_0, Act_s, \to_s)$ where $\mathcal{S} := 2^P$ is the set of states with initial state $S_0 := \{p_0\}$, and where $\to_s \subseteq \mathcal{S} \times Act_s \times \mathcal{S}$ denotes the transition relation labeled by actions from the set Act_s. Again, $\tau \in Act_s$ denotes a local (unobservable) evaluation step and the other labels reflect side effects.

The inference rules which formalize local evaluation steps are directly lifted from the sequential level to the system layer semantics:

$$\frac{e \xrightarrow{\tau}_e e'}{S \cup \{(e, i, q, L, t)\} \xrightarrow{\tau}_s S \cup \{(e', i, q, L, t)\}} \text{(Sequ)}$$

Here, the union operator in the conclusion reflects the interleaving semantics in a set theoretic way.

In most cases, the nondeterminism that was introduced in Section 4.1 can be resolved when considering process systems, where information about the system state is available [8].

4.2.1 Creation of New Processes

New processes are created by evaluating the `spawn` builtin function. Within the sequential layer, the pid j of the newly created process cannot be inferred; therefore it is chosen nondeterministically and reflected by the transition label which indicates the side effect:

$$\frac{}{\texttt{call 'erlang':'spawn'}(a_1, a_2, c) \xrightarrow{\mathsf{spawn}(a_1, a_2, c) \rightsquigarrow j}_e j} \text{(Spwn)}$$

The actual process creation is captured by the system layer rules where the new process term is introduced and the pid is fixed:

$$\frac{e \xrightarrow{\mathsf{spawn}(a_1, a_2, [c_1, \ldots, c_k]) \rightsquigarrow j}_e e' \qquad j \notin \mathsf{Pid}(S) \cup \{i\}}{S \cup \{(e, i, q, L, t)\} \xrightarrow{\mathsf{spawn}(j)}_s S \cup \{(e', i, q, L, t), (\texttt{call}\ a_1 : a_2(c_1, \ldots, c_k), j, \varepsilon, \emptyset, \mathsf{false})\}} \text{(Spawn}_1\text{)}$$

[8] An exception is the creation of new processes, where a new identifier is chosen nondeterministically.

4.2.2 Message Passing

Sending messages affects the state of the sender and the receiver; this is captured on the system layer by the inference rule (Send_1):

$$\frac{e_i \xrightarrow{j!c}_e e_i'}{\begin{array}{l} S \cup \{(e_i, i, q_i, L_i, t_i), (e_j, j, q_j, L_j, t_j)\} \\ \xrightarrow{\mathsf{send}(i,j,c)}_s S \cup \{(e_i', i, q_i, L_i, t_i), (e_j, j, q_j \cdot c, L_j, t_j)\} \end{array}} (\text{Send}_1)$$

The nondeterminism that was introduced to formalize the local evaluation of a `receive` term is fully resolved on the system layer:

$$\frac{e \xrightarrow{\mathsf{recv}(q_1,c)}_e e'}{S \cup \{(e, i, q_1 \cdot c \cdot q_2, L, t)\} \xrightarrow{\mathsf{recv}(i,c)}_s S \cup \{(e', i, q_1 \cdot q_2, L, t)\}} (\text{Recv})$$

Here the application of the qmatch predicate in the premise of inference rule (Rcv_1) assures that the first matching message c is chosen from the mailbox.

4.3 State–Space Reduction

The transition system T_s captures the semantics of a concurrent Core Erlang program by considering local evaluation steps as well as those afflicted with side effects. To reason about the behavior of the whole system, we are primarily interested in inter–process communication and the creation and termination of processes. Therefore we ignore local τ–evaluation steps:

Definition 4.8 The equivalence relation $\sim \subseteq \mathcal{S} \times \mathcal{S}$ is defined by $\sim := \overset{\tau}{\longleftrightarrow}_s^*$.

By migrating to the quotient transition system $T_{/\sim}$, we abstract from local τ–evaluation steps and only observe the processes' interaction:

Definition 4.9 Let $T_\sim := \left(\mathcal{S}_{/\sim}, [S_0]_\sim, Act_\sim, \rightarrow_\sim\right)$ denote the *quotient transition system* where $\mathcal{S}_{/\sim} := \{[S]_\sim \mid S \in \mathcal{S}\}$ denotes the set of states, $[S_0]_\sim$ is the initial state, $Act_\sim := Act_s \setminus \{\tau\}$ is the set of actions, and where the transition relation $\rightarrow_\sim \subseteq \mathcal{S}_{/\sim} \times Act_\sim \times \mathcal{S}_{/\sim}$ is defined by

$$[S]_\sim \xrightarrow{\alpha}_\sim [T]_\sim \quad :\Longleftrightarrow \quad \exists S', T' \in \mathcal{S}. \quad S \overset{\tau}{\longleftrightarrow}_s^* S' \xrightarrow{\alpha}_s T' \overset{\tau}{\longleftrightarrow}_s^* T.$$

Regarding the possible state space reduction, the following lemma holds:

Lemma 4.10 *Let $S = \{p_1, \ldots, p_n\} \in \mathcal{S}$ a process system, $p_j = (e_j, i_j, q_j, L_j, t_j)$ for $1 \leq j \leq n$. Further, let k_j denote the number of consecutive τ–steps of process p_j before reaching a $\xrightarrow{\tau}_e$–normal form. The cardinality of $\mathsf{Post}^*(S, \tau) := \{S' \in \mathcal{S} \mid S \xrightarrow{\tau}_s^* S'\}$ is then bounded by: $|\mathsf{Post}^*(S, \tau)| \leq \prod_{1 \leq j \leq k} (k_j + 1)$.*

According to Lemma 4.10, the $\prod_{1 \leq j \leq k} (k_j + 1)$ successor states of a process system S are represented by one equivalence class within $T_\sim$. Most importantly, in $T_\sim$ we do no longer consider interleaving of τ–evaluation steps which is natural given that those transitions do not affect other processes at all.

```
sort Process .
op <_|_|_|_|_|_|_|_> : Label SysResult Expr Pid Mailbox
                       PidSequence Bool ModEnv -> Process [ctor] .
sort Processes .
subsort Process < Processes .
op #empty-processes : -> Processes [ctor] .
op _||_ : Processes Processes -> Processes [ctor assoc comm
                                            id: #empty-processes] .
sort ProcessEnvironment .
op ((_,_,_,_)) : SysLabel Processes
                 ModEnv PidSequence -> ProcessEnvironment [ctor] .
```

Fig. 4. Signature definition

5 Implementation in Maude

The small step operational semantics introduced in Section 4 relates a given Core Erlang program with an initial process p_0 to a quotient transition system $T_\sim$ thereby formalizing the possible system behaviors. In order to automatically reason about properties of such systems, in this section we use a Rewriting Logic specification of our semantics to operationalize the computation of $T_\sim$.

According to the Rewriting Logic framework, in Figure 4 we first define the signature of processes and process systems: to implement our semantics, we extend the representation of a process $P = Exp \uplus \{\bot\} \times \mathbb{N} \times \mathsf{Const}^* \times 2^{\mathbb{N}} \times \mathbb{B}$ (see Definition 4.5) by three additional components:

(i) a process `Label` that summarizes the process' state,
(ii) a `SysResult` term indicating the result of the last side effect and
(iii) the set `ModEnv` of known function declarations.

Accordingly, process systems are multisets of `Process` terms, represented by the associative and commutative list constructor "`||`".

5.1 Core Erlang *Signature*

Apart from these basic operator declarations, a slightly restricted[9] Core Erlang syntax (cf. Section 2) is specified as a many–kinded signature so that it can be parsed by the Maude interpreter. To allow arbitrarily many arguments (e.g., when considering `apply` expressions), we define a flattened argument list operator:

```
subsort Expr < NeExprList .
op _,_ : NeExprList NeExprList -> NeExprList [ctor assoc] .
```

As a consequence, unbounded argument lists are internally replaced by a single argument of sort `NeExprList`. Note however, that due to the flattened concatenation operator "`_,_`" this does still allow to parse arbitrary Core Erlang expressions.

In the Maude system, terms are built using many–kinded signatures. Here a sort denotes a semantic concept whereas kinds refer to the notion of a sort in the context of traditional many–sorted signatures.

On the syntactical level, each well–formed term is assigned a kind whereas its affiliation to a designated sort has to be inferred using membership axioms of an underlying membership equational theory (cf. [2] for details).

[9] Additional whitespaces are required.

5.2 Specification of the Process Layer

In principle, mapping the quotient transition–system semantics to a Rewriting Logic specification is straightforward: the equivalence relation $\sim := \overset{\tau}{\hookleftarrow}_s^*$ is transformed into an equational theory that models local τ–evaluation steps. Operationally, these equations are split into a set of equations A that is inferred according to associativity and commutativity attributes of the operators and a set of directed equations ER that constitute a terminating and confluent term rewrite system.

Deviating from the Core Erlang semantics, in our implementation we specify the sequential semantics already wrt. the term representation of a single process instead of considering closed Core Erlang expressions only [10]. In the underlying sequential semantics, we use nondeterminism to model side effect afflicted evaluation steps. However, the implementation requires the set ER of directed equations to converge modulo A. We therefore augment the process terms with a `Label` and a `SysResult` component to avoid nondeterministic choices.

As a first example, consider the semantics of the sequencing operator `do`:

```
var ESL : StopLabel .

ceq < tau | RES | do EX1 EX2 | PID | MBOX | LINKS | TRAP | ME > =
    < #filterExit(ESL) | RES1 | do EX1' EX2 | PID | MBOX | LINKS | TRAP | ME >
  if not(EX1 :: Const)
     /\ < ESL | RES1 | EX1' | PID | MBOX | LINKS | TRAP | ME > :=
        < tau | RES | EX1 | PID | MBOX | LINKS | TRAP | ME > .
```

A necessary condition for the applicability of this directed equation is that the first subexpression `EX1` is not a value yet. Only then, further evaluation takes place within the second condition yielding a new expression `EX1'`. The sort of the label variable `ESL` is crucial here: If the evaluation of `EX1` yields a side effect, the process' label changes to a term of sort `StopLabel` thereby reaching a normal form wrt. the equational rewriting. Therefore, the result of the side effect is not guessed non–deterministically within the "local" process layer, but is resolved later by the rewrite rules that operate on these normal forms.

5.2.1 Substitutions

The pattern matching semantics is based on syntactic substitutions on Core Erlang expressions; a single binding is represented as a term of sort `Binding`. An environment is then constructed as a comma–separated associative and commutative list of such variable bindings:

```
sort Binding Env .
op _-->_ : Var Const -> Binding [ctor] .

subsort Binding < Env .
op #empty-env : -> Env [ctor] .
op _,_ : Env Env -> Env [ctor assoc comm id: #empty-env] .
```

Given a Core Erlang expression, the `#subst` function specifies the substitution of free variables according to a given environment by recursively descending into the subterms. Therefore, the base cases include

[10] Therefore argument evaluation cannot be specified by Maude's evaluation strategies.

```
eq #subst(V, (V --> C), ENV) = C .
eq #subst(E, ENV) = E [owise] .
```

Here, the first rule specifies the substitution of a variable that gets bound according to the environment. Note that since AC matching is involved, we can assume without loss of generality that the corresponding `Binding` term is the first binding in the environment. When considering Core Erlang expressions that introduce new variable bindings, the environment has to be shrunken accordingly:

```
ceq #subst(let VS = EX1 in EX2, ENV) = let VS = #subst(EX1, ENV) in #subst(EX2, ENV1)
  if VSET := #projectVarSet(VS) /\ ENV1 := #restrictEnv(VSET, ENV) .
```

The scope of a new binding introduced by evaluating a `let` expression ranges over the `EX2` expression only. Therefore, the substitution is applied to the `EX1` subterm without any modification. According to the semantics of `let`, in `EX2` free occurrences of variables of the sequence `VS` are bound and may not be substituted. In `ENV1`, such "critical" bindings are removed thereby excluding the variables bound by the `let` context. Based on the variable sequence `VS`, the functions `#projectVarSet` and `#restrictEnv` compute the set of newly bound variables and shrink the environment accordingly.

5.2.2 Pattern Matching

Pattern matching is formalized by the partial function `#match`. Matching failures are represented by introducing a constant `#nomatch` with a new sort `Env?` which is declared as a supersort of variable environments.

During pattern matching, we recursively descend into the subterms of the given pattern and try to construct a unifying variable environment. In case of a clash failure[11], the `#nomatch` constant is included in the environment indicating the matching failure. Therefore we have:

```
subsort Env < Env? .
op #match : Pat Const ~> Env .

eq #match(VAR, CONST) = (VAR --> CONST) .
eq #match(CONST, CONST) = #empty-env .
eq #match([PAT1|PAT2], [C1|C2]) = #match(PAT1, C1), #match(PAT2, C2) .
eq #match(PAT, CONST) = #nomatch [owise] .
```

According to Core Erlang's semantics, all variables in a pattern are free; therefore we can directly construct a variable binding when matching against a constant value. In the same way, two identical values match each other without entailing a new binding.

In more complex patterns like lists, the `#match` function recursively descends into the corresponding subterms. Finally the fourth equation covers matching failures and is – according to the `owise` attribute – applicable only if none of the other `#match`–equations allows to continue the matching process.

The `#subst` and `#match` functions allow to specify the semantics of Core Erlang's pattern matching operations. A Core Erlang `case` expression has the form `case` v `of` $cl_1 \cdots cl_k$ `end` where evaluation continues with the expression e_i

[11] Due to Core Erlang's pattern matching semantics occur failures cannot happen.

of the first matching clause $cl_i = p_i$ `when` g_i `->` e_i in the sequence $cl_1, \ldots, cl_k$. This is specified by the following directed equation:

```
ceq <tau|#no-res|case C of PAT when GUARD -> EX CLAUSES end|PID|MBOX|LINKS|TRAP|ME>
  = <tau|#no-res|EX2|PID|MBOX|LINKS|TRAP|ME>
if ENV := #match(PAT, C)
   /\  <exception(exit, atom("normal"))|#no-res|EX1|PID|#empty-mbox|LINKS|TRAP|ME> :=
       <tau|#no-res|#subst(GUARD,ENV)|PID|#empty-mbox|LINKS|TRAP|ME>
   /\ EX2 := if EX1 == atom("true") then #subst(EX, ENV)
                                    else case C of CLAUSES end fi .
```

According to the first condition, the equation is applicable only if the pattern `PAT` matches the value `C`. The second condition formalizes the guard evaluation. Dependent on its result, evaluation continues either with the clause's right hand expression or – if the guard does not evaluate to `'true'` – a modified `case` term is reevaluated with the failed clause removed.

Furthermore, the pattern matching itself may fail:

```
ceq <tau|#no-res|case C of PAT when GUARD -> EX CLAUSES end|PID|MBOX|LINKS|TRAP|ME>
  = <tau|#no-res|case C of CLAUSES end|PID|MBOX|LINKS|TRAP|ME>
if not(#match(PAT, C) :: Env) .
```

If the first clause in the sequence does not match the constant `C`, the environment computed by the `#match` function contains the `#nomatch` constant; therefore `#match(PAT, C)` is of sort `Env?` and the membership formula is not fulfilled. In this case, the failed clause is removed and matching is continued with the tail of the clause sequence.

5.3 System layer semantics

The directed equations introduced so far describe unobservable local evaluation steps. The observable transitions that we consider now model interactions between ERLANG processes and are formalized by rewriting rules. From an operational point of view, these rules operate on normal forms wrt. equational rewriting.

The rules that cover process creation provide a first example. When symbolically evaluating an expression `call 'erlang':'spawn'`(a_1, a_2, c), the process' label is changed to indicate the side effect. This yields a normal form wrt. equational rewriting; the system–layer rules then compute a new identifier and extend the process environment:

```
crl (SL, <spawn(A1, A2, LIST)|#no-res|EX|PID|MBOX|LINKS|TRAP|ME> || PRCS, ME', PIDS)
 => (sys-newproc(PID, pid(INT), A1, A2, LIST),
     <tau|#res-spawn(INT)|EX|PID|MBOX|LINKS|TRAP|ME> ||
     <tau|#no-res|EX1|pid(INT)|#empty-mbox|#empty-pid-seq|false|ME'> ||
     PRCS, ME', pid(INT), PIDS)
if INT := #getNewPid(PIDS)
   /\ EX1 := if LIST == [] then call A1:A2 ()
                           else call A1:A2 (#getListElements(LIST)) fi .
```

When evaluating the expression `call 'erlang':'!'(Pid, Msg)`, the message is appended to the mailbox of the process identified by `Pid`. This information is passed to the corresponding rewrite rule from within the equational theory:

```
eq <tau|#no-res|call atom("erlang"):atom("!")(int(INT),C)|PID|MBOX|LINKS|TRAP|ME>
 = <pid(INT)!C|#no-res|call atom("erlang"):atom("!")(int(INT),C)|PID|MBOX|LINKS|TRAP|ME> .
```

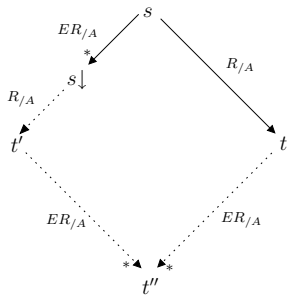
The transition rules of the system level operate on this normal form by extracting the receiver's PID and the message from the process' label and appending the message to the receiver's mailbox:

```
crl (SL, <pid(INT)!C|#no-res|EX|PID|MBOX|LINKS|TRAP|ME> ||
         <EL1|#no-res|EX1|pid(INT)|MBOX1|LINKS1|TRAP1|ME1> || PRCS, ME', PIDS)
 => (sys-sendmsg(PID, pid(INT),C),
       <tau|#res-send(true)|EX|PID|MBOX|LINKS|TRAP|ME> ||
       <EL1'|#no-res|EX1|pid(INT)|MBOX1:C|LINKS1|TRAP1|ME1> || PRCS,ME',PIDS)
if EL1' := if (EL1 == waiting) or (EL1 == blocked) then tau
                                                   else EL1 fi .
```

5.3.1 Completeness

Operationally, a given process environment is first normalized by equational rewriting. Then, the system layer's transition rules are applied to these normal forms. The following result from [10] shows that this implementation is complete, i.e., by first normalizing the process environment, no transitions get lost that would otherwise be possible.

Theorem 5.1 (Completeness) *Let $\mathfrak{T} = (\Sigma, E, R)$ be the Rewriting Logic specification of the* Core Erlang *semantics where $E = ER \uplus A$ and s, t, t', t'' denote process environments. Then it holds:*



According to Theorem 5.1, given a process environment s, any applicable transition rule is also applicable to the ER normal form $s{\downarrow}$ of s and the resulting terms are again in the same equivalence class (modulo ER).

6 Verifying system properties

Based on the transition–system model of a given Core Erlang program that is computed by the interpreter, Maude's integrated LTL model checker allows to verify system properties: terms of sort `ProcessEnvironment` constitute the states of the computed transition system. Because we focus on inter–process communication in a distributed environment, the relevant properties refer to the system's transitions instead of its states. Therefore, each state is augmented by the transition label of the incoming transition. This label is reflected as a term of sort `SysLabel` in the first component of each `ProcessEnvironment`.

With this approach, it is possible to define state predicates (for an in–depth discussion of the Maude model checker, refer to [6]). For example, the `send` predicate

is defined as follows:

```
op send : Int Int Const -> Prop .
eq (sys-sendmsg(pid(P1),pid(P2),C),PRCS,ME,PIDS) |= send(P1,P2,C) = true .
```

It takes the identifiers of the sender and receiver and the message as arguments. In the equation, the validity of the predicate is defined wrt. the `SysLabel` component of the state which reflects the action that led into this state. Therefore it determines that the parameterized predicate `send(P1, P2, C)` holds iff the incoming transition label reflects the corresponding send operation.

An essential issue when reasoning about concurrent systems is the possibility to specify fair scheduling strategies. In our approach, we impose fairness constraints as LTL premises. For an a priori given set of processes, the `scheduler` function evaluates to an LTL–premise specifying a fair scheduling strategy. It has the form

$$\varphi_{\mathsf{scheduler}(i_1,\ldots,i_n)} := \bigwedge_{k=1}^{n} \Box \left(p_{\mathsf{running}(i_k)} \rightarrow p_{\mathsf{running}(i_k)} \,\mathsf{U}\, \left(p_{\mathsf{scheduled}(i_k)} \vee p_{\mathsf{blocked}(i_k)} \right) \right)$$

where $i_1, \ldots, i_n$ denote the included processes and $p_{\mathsf{running}(i_k)}$ states the existence of the kth process. The predicate $p_{\mathsf{scheduled}(i_k)}$ holds iff the process with identifier i_k caused the last system level transition, and $p_{\mathsf{blocked}(i_k)}$ is valid iff the corresponding process is blocked during message reception. Intuitively, it states that whenever a process exists, it is scheduled sometime later or it becomes blocked waiting for message reception.

Considering again the mutual exclusion program for two competing processes from Figure 2, we can now successfully verify the mutual exclusion property specified by the following LTL formula:

$$\begin{aligned} \varphi_2 = \varphi_{\mathsf{scheduler}(0,1,2)} \rightarrow \Box \left(p_{\mathsf{send}(0,1,\text{'ok'})} \rightarrow \left(\neg p_{\mathsf{recv}(2,\text{'ok'})} \,\mathsf{U}\, p_{\mathsf{recv}(0,\{\text{'rel'},1\})} \right) \right) \\ \wedge \Box \left(p_{\mathsf{send}(0,2,\text{'ok'})} \rightarrow \left(\neg p_{\mathsf{recv}(1,\text{'ok'})} \,\mathsf{U}\, p_{\mathsf{recv}(0,\{\text{'rel'},2\})} \right) \right) \end{aligned}$$

Recapitulatory, in our approach we use Meseguer's Rewriting Logic framework and the Maude system to automatically compute the transition–system model of a given Core Erlang program. On this basis, we define state predicates and use Maude's integrated LTL model checker to verify certain linear–time properties.

References

[1] Armstrong, J., S. Virding, M. Williams and C. Wikström, "Concurrent Programming in Erlang," Prentice Hall International, 1996, 2nd edition.

[2] Bouhoula, A., J.-P. Jouannaud and J. Meseguer, *Specification and proof in membership equational logic*, Theoretical Computer Science **236** (2000), pp. 35–132.

[3] Carlsson, R., *An introduction to Core Erlang*, in: *Proceedings of the PLI'01 Erlang Workshop*, 2001. URL http://www.erlang.se/workshop/carlsson.ps

[4] Carlsson, R., B. Gustavsson, E. Johansson, T. Lindgren, S.-O. Nyström, M. Pettersson and R. Virding, *Core Erlang 1.0.3 language specification* (2004), http://www.it.uu.se/research/group/hipe/corerl/doc/core_erlang-1.0.3.pdf.

[5] Clarke, E., O. Grumberg and D. Peled, "Model Checking," The MIT Press, 1999.

[6] Clavel, M., F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer and C. Talcott, "Maude manual (Version 2.1.1)," Menlo Park, CA 94025 USA (2005).
URL http://maude.cs.uiuc.edu/

[7] Felleisen, M. and R. Hieb, *The revised report on the syntactic theories of sequential control and state*, Technical Report 100-89, Department of Computer Science, Rice University (1991).

[8] Martí–Oliet, N. and J. Meseguer, *Rewriting logic: Roadmap and bibliography*, Theoretical Computer Science **285** (2002), pp. 121–154.

[9] Meseguer, J., *Conditional rewriting logic as a unified model of concurrency*, Theoretical Computer Science **96** (1992), pp. 73–155.

[10] Neuhäußer, M., "Abstraktion und Model Checking von Core Erlang Programmen in Maude," Master's thesis, Faculty of Mathematics, Computer Science and Natural Sciences, RWTH Aachen (2005), http://www-i2.cs.rwth-aachen.de/Staff/Current/neuhaeusser/publications/da.pdf.

[11] Viry, P., *Rewriting: An effective model of concurrency*, in: *Proceedings of PARLE'94 – Parallel Architectures and Languages Europe*, LNCS **817** (1994), pp. 648–660.