Electronic Notes in Theoretical Computer Science 44 No. 2 (2001)
URL: http://www.elsevier.nl/locate/entcs/volume44.html 19 pages

A Rewriting Logic Implementation of Erlang

Thomas Noll [1]

Lehrstuhl für Informatik II
Aachen University of Technology
52056 Aachen, Germany

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. It presents a formal description of this language in Rewriting Logic, a unified semantic framework for concurrency which is semantically founded on conditional term rewriting modulo equational theories. The formalization is tailored to the SLC Specification Language Compiler Generator, which is being developed at Aachen University of Technology as part of the TRUTH verification platform. SLC can be used to automatically translate the Erlang description into a verification tool frontend, i.e., a parser and a semantic evaluator which, given an Erlang program, compute the associated transition system.

1 Introduction

The high quality demands on software for telecommunication systems may partly be ensured by the use of formal methods in the design and development. By the high degree of concurrency in applications of this area, *testing* is often not sufficient to guarantee correctness to a satisfactory degree. *Verification*, namely formally proving that a system has the desired properties, is therefore becoming a more and more widespread practice (see [4] for an overview).

Although a complete *formal specification* of a telecom application would probably be one of the best ways to ensure its correctness, in practice the descriptions are rather informal, written in natural language in combination with some fragments of for example the Standard Description Language SDL [14]. Reasons for the absence of a complete formal specification can be found in the fact that the specification changes several times during development, triggered by experiments with a release or by changed requirements. It is felt too time consuming to modify both the formal specification and the code.

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

Even the informal specification tends to run out of phase with the actual implementation and is often only updated after a release of the product.

Towards the end of a project, it is the running *code* that represents the best "specification" of the actual application. Questions about the implementation are therefore best formulated in terms of this code: "is there a possibility that this finite–state machine implementation deadlocks?", "does this server implementation correctly respond to all possible requests?".

Here we address the verification issue in the context of the functional programming language Erlang [1], which was developed by the Ericsson corporation to address the complexities of developing large–scale programs within a concurrent and distributed setting. It is successfully used in the design and implementation of telecommunication systems. Due to the presence of unbounded data structures and of dynamic process spawning, Erlang programs usually induce infinite–state systems. It is therefore natural to employ interactive *theorem–proving assistants* such as the EVT Erlang Verification Tool [2] to establish the desired system properties. Alternatively it is possible to formally abstract infinite state spaces to finite–state form, followed by fully automated *model checking* [7].

In any case an implementation of the Erlang semantics [6] has to be developed, mapping Erlang programs to transition systems. (Additionally, the abstraction functions have to be implemented for the model–checking approach.) Doing this by hand is a very time–consuming task. We therefore propose to employ a *compiler–generating approach* in which the verification tool frontend (that is, a parser and a semantic evaluator) is automatically generated from a given description of the Erlang programming language.

More concretely, we formally describe Erlang using the *Rewriting Logic* framework, which was proposed in [10] as a unified semantic framework for concurrency. It has proven to be an adequate modeling formalism for many concrete specification and programming languages [11]. 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. This *executable specification* property is exploited by the SLC specification language compiler generator [9] which, given a Rewriting Logic description of the syntax and semantics of the specification language in question, derives a compiler which parses any given system specification and computes the associated semantic object. SLC is part of the Truth verification tool [8] which can subsequently be used to visualize and to analyze the semantics. In particular it can be employed to perform model checking, i.e. to verify automatically that the system under consideration meets certain conditions given as formulae of some mathematical logic.

The remainder of this paper is organized as follows. Section 2 introduces

the Erlang programming language by sketching its syntactic constructs and their intuitive meaning. Section 3 briefly describes the Rewriting Logic framework and its concrete implementation, the SLC Specification Language Compiler Generator. The actual Rewriting Logic description of Erlang is presented in Section 4. We conclude with Section 5 by summarizing our first experiences, and by describing the topics of future work.

2 The Erlang Programming Language

Erlang/OTP is a programming platform providing the necessary functionality for programming open distributed (telecommunication) systems: the language Erlang with support for 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.

In the following we consider a core fragment of the Erlang programming language which allows to implement 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. Real Erlang has several additional features such as modules, distribution of processes (onto nodes), and support for interoperation with non–Erlang code written in, e.g., C or Java.

Besides Erlang *expressions* e we operate with the syntactical categories of *matching clauses* cs, *patterns* p, and *values* v. The abstract syntax of Core Erlang expressions is summarized as follows:

$$
\begin{aligned}
e ::= {} & \texttt{begin}\ e_1, \ldots, e_n\ \texttt{end} \mid e_1, e_2 \mid e(e_1, \ldots, e_n) \\
& \mid \texttt{case}\ e\ \texttt{of}\ cs\ \texttt{end} \mid p = e \mid e_1\,!\,e_2 \\
& \mid \texttt{receive}\ cs\ \texttt{end} \mid op(e_1, \ldots, e_n) \mid \texttt{spawn}(e_1, e_2) \\
& \mid \texttt{self}() \mid V \\
cs ::= {} & p_1\ \texttt{->}\ e_1; \ldots; p_n\ \texttt{->}\ e_n \\
p ::= {} & op(p_1, \ldots, p_n) \mid V \\
v ::= {} & op(v_1, \ldots, v_n)
\end{aligned}
$$

Here V ranges over Erlang variables, and op ranges over a set of primitive constants and operations including tupling $\{e_1, e_2\}$, list prefix $[e_1|e_2]$, the empty list $[\,]$, integers, pid constants, and atoms.

The functional sublanguage of Erlang is rather standard: atoms, integers, lists and tuples are value constructors; both $\texttt{begin}\ e_1, \ldots, e_n\ \texttt{end}$ and e_1, e_2 denote sequential composition; and $e(e_1, \ldots, e_n)$ represents a function call. The expression $\texttt{case}\ e\ \texttt{of}\ p_1\ \texttt{->}\ e_1; \ldots; p_n\ \texttt{->}\ e_n\ \texttt{end}$ involves matching: the

value that e evaluates to is matched sequentially against the patterns p_i. If this succeeds, evaluation continues with e_i where the variables bound by p_i are correspondingly instantiated. The same is true for the assignment $p = e$ where a runtime error is raised if the value of e does not match p, and where this value is returned as the result otherwise.

The constructs involving non–functional behaviour (i.e., side effects) are $e_1 ! e_2$ which denotes an output step, sending the value of e_2 asynchronously to the process identified by e_1, whereas `receive` cs `end` inspects the mailbox q of the local process and retrieves (and removes) the first element in q that matches any pattern in cs. Once such an element v has been found, evaluation proceeds analogously to `case` v `of` cs `end`. `spawn`(e_1, e_2) dynamically creates a new process in which the function given by e_1 is applied to the arguments given by the list e_2, and `self`() returns the pid of the local process.

Expressions are interpreted relative to an environment of "user–defined" function declarations, each of the form

$$\begin{aligned} f(p_{11}, \ldots, p_{1n}) \ \texttt{->}\ & e_1; \\ & \vdots \\ f(p_{k1}, \ldots, p_{kn}) \ \texttt{->}\ & e_k \end{aligned}$$

The following code fragment implements a simple concurrent server which repeatedly accepts an incoming query in form of a triple which is tagged by the `request` constant, and which contains the request itself (matched by the variable `Request`) and the pid of the client process (`Client`). It then spawns off a process which serves the request by invoking the `handle` function (which is not shown here), and by sending the result back to the client as a tuple tagged by the `response` constant.

```
central_server() ->
  receive
    {request, Request, Client} ->
      spawn(serve, [Request, Client])
  end,
  central_server().
serve(Request, Client) ->
  Client!{response, handle(Request)}.
```

The starting point of any kind of rigorous verification is a formal semantics. Here we use an operational semantics by associating a transition system with an Erlang program, giving a precise account of its possible computations. The states are parallel products of processes, each of the form $\langle e \mid p \mid q \rangle$, where e is an Erlang expression to be evaluated in the context of a unique pid p and a mailbox q for incoming messages. A set of rules is provided to derive labeled transitions between such states [6].

A *reduction context* is an Erlang expression $r[\cdot]$ with a "hole" $\cdot$ in it, which

identifies the position of r where the next evaluation step is going to take place (according to Erlang's leftmost–innermost reduction strategy). In this way, the rules for the actual expression evaluation have to be given only for exceptional cases, namely, when all parameters of an expression construct are *values*, i.e., have been fully evaluated.

For example, process creation is formally described by the following rule [2]:

$$\langle r[\texttt{spawn}(f,[v_1,\ldots,v_n])] \mid p \mid q\rangle \xrightarrow{\tau} \langle r[p'] \mid p \mid q\rangle \parallel \langle f(v_1,\ldots,v_n) \mid p' \mid \varepsilon\rangle$$

Here, f is a (function) atom, $v_1,\ldots,v_n$ are values, q is an arbitrary mailbox, and ε denotes the empty mailbox. Thus, a process evaluating a `spawn` function call has a transition to a system of two processes ($\parallel$ denotes parallel composition) which have to evaluate the expressions $r[p']$ (p' is the return value of `spawn`) and $f(v_1,\ldots,v_n)$, respectively. For the process identifiers p and p' we require $p \neq p'$.

It is obvious that the combination of dynamic process creation with recursive functions gives rise to infinite state spaces. The same is true for unbounded data values such as numbers and lists, and unbounded mailbox queues (see Section 4).

3 The SLC Specification Language Compiler Generator

SLC is a compiler generator providing generic support for different specification formalisms [9]. Given a formal description of a specification language, it automatically generates a corresponding HASKELL frontend for a verification tool. Although SLC is tailored to the TRUTH verification platform (cf. Figure 1), this frontend can in principle be used by other tools such as EVT as well provided that they support the interoperation with HASKELL functions.

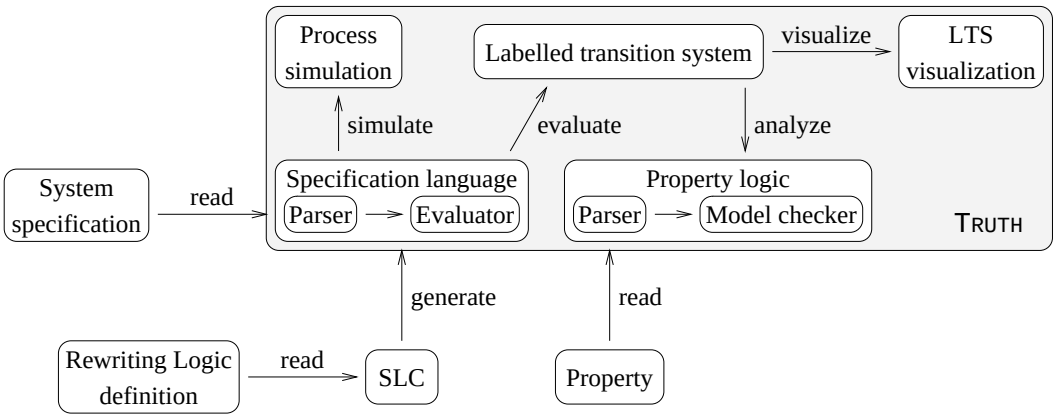

Fig. 1. Structure of TRUTH/SLC

As input, SLC expects a description of the syntax and of the semantics of the specification language in terms of Rewriting Logic. From this definition a

[2] Actually, [6] introduces a two–level semantics which distinguishes between expression–level and process–level steps. However this difference is of no importance here.

compiler is derived which is capable of parsing a concrete system specification and of computing the corresponding semantic object, such as a labeled transition system. This compiler is linked together with the Truth platform to obtain a model–checking tool which is tailored for the specification language in question.

The description of the specification language formalism consists of three parts. First, the syntax of the language has to be given by a context–free grammar (with typing information). This yields a sorted signature Σ of operation symbols. The second part is a set of conditional rewrite rules R defining the operational semantics. Finally, the description contains a set of equations E between process terms which identify certain states of the respective system, thus reducing the state space. From the point of view of the operational semantics this means that we are dealing with conditional term rewriting modulo equational theories.

Aiming at its practical usability, some fundamental restrictions are imposed on the Rewriting Logic framework as it is implemented in SLC. We will motivate them by some concrete examples taken from the Erlang programming language, picking them up again in Section 4 where we consider the SLC implementation of Erlang in greater detail.

Congruence: (Conditional) term rewriting systems usually induce congruence relations, that is, a rule can be applied in arbitrary contexts provided that the left–hand side matches the respective subterm and that the conditions are fulfilled. In the case of Erlang this assumption would contradict e.g. the fixed left–to–right evaluation order of function arguments, which is mandatory to respect since argument evaluations may involve side effects such as inter–process communication.

Equations: Since (conditional) term rewriting modulo equational theories is generally intractable or even undecidable, it is not possible to admit arbitrary equations in E.

Dataflow: In a conditional rewriting rule $\rho \in R$ of the general form

$$\frac{c_1 \longrightarrow d_1 \quad \ldots \quad c_k \longrightarrow d_k}{l \longrightarrow r} \ (\rho)$$

where $k \geq 0$, and where the single terms are built up using operators from Σ and variables from some set Var, the use of variables must be restricted in such a way that the "dataflow" between them can be implemented deterministically. For example, if c_1 would contain a variable that does not occur in l, it would be unclear how to evaluate it when applying ρ to some instance of l.

The solutions adopted by SLC are the following:

(i) A concrete language specification distinguishes *compatible* operators from *incompatible* ones, and the transition rules in R are allowed to be applied in "compatible contexts" only. More exactly, a rewriting step at some position of a term is possible only if every symbol on the path from

the root to this position is compatible. Formally this is reflected by the following rule, valid for every compatible symbol $f \in \Sigma$:

$$\frac{s_1 \longrightarrow t_1 \quad \ldots \quad s_k \longrightarrow t_k}{f(s_1, \ldots, s_k) \longrightarrow f(t_1, \ldots, t_k)} \text{ (CO)}$$

(ii) Following the ideas of Viry in [15,16], E is decomposed into a set of directed equations ER (that is, a term rewriting system) and into a set AC expressing the associativity and commutativity of certain binary operators in Σ. In the case of Erlang, one would e.g. choose the parallel operator $\|$ to be associative and commutative. Given that ER is terminating modulo AC, then rewriting by R modulo E can be implemented by a combination of normalizing by ER and rewriting by R, both modulo AC.

From a semantical point of view this means that we are working with Abelian monoids or, in other words, multisets, which turn out to be appropriate for many applications in the area of concurrency and distributed systems. The paper [13] addresses the question under which premises this restriction of the fully–equational semantics is sound and complete.

(iii) In a rewrite rule $\rho \in R$ of the above form, we require that, for every $i \in \{1, \ldots, k\}$,

$$Var(c_i) \subseteq Var(l) \cup \bigcup_{j=1}^{i-1} Var(d_j)$$

and

$$Var(r) \subseteq Var(l) \cup \bigcup_{i=1}^{k} Var(d_i),$$

where $Var(t)$ denotes the set of variables occurring in a term t. Thus every rewriting step can be computed by evaluating the rules in a depth–first left–to–right fashion. [3]

4 Erlang in Rewriting Logic

This section presents some of the essential parts of the Erlang description in SLC. It starts with the declaration of the sorts:

```
ALGEBRA Erlang

sorts
  definition, clause, system, expression, value, pid, action
```

Their interpretation is as follows:

- `definition` represents a function definition. An Erlang program corresponds to a sequence of function definitions.

[3] This syntactic restriction corresponds to the 3–CTRS property of ordinary conditional term rewriting systems (cf. [12]).

- `clause` is the sort of the single clauses which make up a function definition or a `case` expression. Pattern matching is used to distinguish the single cases.
- `system` stands for the system states, i.e., parallel products of processes.
- `expression` represents Erlang expressions, which e.g. occur on the right–hand side of a function definition or as the first component in a process.
- `value` is the subsort of expressions in normal form.
- `pid` stands for process identifiers.
- `action` represents the action labels of the transitions in the operational semantics.

The next section declares the operators with their associated sorts, their textual output representation (enclosed in quotation marks), and their associativity and commutativity (`AC`) as well as their compatibility (`CO`) properties. Thus it represents the sorted signature Σ of a Rewriting Logic description. It should be mentioned that the operator output representation is only used for displaying system states; the input syntax is determined in the `SYNTAX` part of the SLC description (see below).

We just give some of the operators dealing with process systems, expressions, values, and actions, and leave out those for function definitions, builtin functions, and clauses.

```
cons
  -- Process systems
  Par: system * system -> system                  ("_ || _")  (AC,CO)
  Cnt: pid -> system                              ("[_]")
  Prc: expression * pid * value * value -> system ("<_|_|_|_>")

  -- Expressions
  Seq: expression * expression -> expression      ("_, _")
  Snd: expression * expression -> expression      ("_!_")
  App: expression * expression -> expression      ("_(_)")
  Cas: expression * (clause list) -> expression   ("case _ of _ end")
  Rcv: (clause list) -> expression                ("receive _ end")
  TpE: expression -> expression                   ("{_}")
  CnE: expression * expression -> expression      ("[_|_]")
  Val: value -> expression                        ("_")

  -- Values
  TpV: value -> value                             ("{_}")
  CnV: value * value -> value                     ("[_|_]")
  Nil: unit -> value                              ("[]")
  Atm: string -> value                            ("_")
  Num: string -> value                            ("_")
  Pid: pid -> value                               ("_")
  Var: string -> value                            ("_")
```

```
...

-- Pids
Fst: unit -> pid                                ("fst")
Nxt: pid -> pid                                 ("nxt(_)")

-- Actions
Tau: unit -> action                             ("tau")
Out: pid * value -> action                      ("_!_")
Spn: value * value -> action                    ("spawn(_,_)")

...
```

The representation of process systems deserves some comments. Just as in the original Erlang semantics, they correspond to parallel products of processes. However, every process has a fourth (value) component for reasons to become clear later. As before, the third component is the mailbox, which, for the sake of simplicity, is considered as a value. Moreover the creation of unique pids, which is essential in connection with process spawning, is handled explicitly by introducing a dedicated counter "process" which is used to store the next free pid. Thus, a process system is of the general form

$$[p] \parallel \langle e_1 \mid p_1 \mid q_1 \mid o_1 \rangle \parallel \ldots \parallel \langle e_n \mid p_n \mid q_n \mid o_n \rangle$$

where $n \geq 1$ or, in internal representation,

$$\texttt{Par}(\texttt{Cnt}(p), \texttt{Par}(\texttt{Prc}(e_1, p_1, q_1, o_1), \texttt{Par}(..., \texttt{Prc}(e_n, p_n, q_n, o_n))))$$

Pids are represented by constructor terms of the form `Nxt`(...`Nxt`(`Fst`)), starting with `Fst` and incrementing the process counter whenever a new process is being spawned (see the `rules` section below).

Regarding expressions and values it should be mentioned that SLC does not support an order–sorted typing framework. Hence the conversion from values to expressions has to be specified explicitly, using the `Val` operator, and different constructors have to be used for lists (`CnE`/`CnV`) and tuples (`TpE`/`TpV`) of expressions and values, respectively.

The next part describes the syntactic structure of Erlang programs. It declares the tokens and their associativity (`left` or `right`) and priority (where higher numbers denote a higher priority), the nonterminal symbols, and the context–free production rules together with their abstract syntax tree representation.

In the `nonterminals` section, additional keywords are used to indicate that the objects derived from the respective symbol have a special meaning:

- `SYSTEM` labels the nonterminal symbol from which the definition of a program identifier (here, an Erlang function name) is derivable. During parsing, these are automatically registered in a symbol table which is made accessible to the `rules` part.
- `STATE` refers to the nonterminal which represents a state in the transition

system associated with the program (here, an Erlang process system).

- `ACTION` identifies the symbol from which the action labels of the transition system are derivable.

```
SYNTAX

tokens
  -- Keywords
  "begin"              => BEGIN
  "case"               => CASE
  "end"                => END
  ...

  -- Bracket symbols
  "\("                 => OPENB
  "\)"                 => CLOSEB
  "{"                  => OPENCB
  ...

  -- Other symbols
  "."                  => DOT
  ","                  => COMMA
  ";"                  => SEMICOLON
  ...

  -- Complex symbols
  "[A-Z_][A-Za-z0-9_]*" => VAR of String
  "[a-z][A-Za-z0-9_]*"  => ATOM of String
  "[0-9]+"              => NUMBER of String

priorities
  right 10 PARALLEL
  right 20 COMMA
  right 30 EQUAL
  ...

nonterminals
  def  of definition        (SYSTEM)
  sys  of system            (STATE)
  par  of system
  prc  of system
  mbx  of value
  exp  of expression
  nseq of expression
  nass of expression
  nout of expression
  napp of expression
```

```
  exps of expression
  val  of value
  vals of value
  pid  of pid
  mcls of (clause list)
  pat  of value
  act  of action            (ACTION)
  ...

grammar
  def : ...

  -- Process systems
  sys : OPENSB pid CLOSESB PARALLEL par   (Par(Cnt(pid), par))

  par : prc                               (prc)
      | prc PARALLEL par                  (Par(prc, par))

  prc : LT exp BAR pid BAR mbx BAR val GT (Prc(exp, pid, mbx, val))

  mbx : val                               (val)

  -- Expressions
  -- Hierarchy to avoid left recursion:
  -- exp: general expressions
  -- nseq: no sequencing
  -- nass: no assignments
  -- nout: no output
  -- napp: no function application
  -- val: values
  exp : nseq                              (nseq)
      | nseq COMMA exp                    (Seq(nseq, exp))

  nseq: nass                              (nass)
      | pat EQUAL exp                     (Cas(exp, [Cls(pat, Val(pat))]))

  nass: nout                              (nout)
      | nout BANG nseq                    (Snd(nout, nseq))

  nout: napp                              (napp)
      | napp OPENB exps CLOSEB            (App(napp, exps))

  napp: BEGIN exp END                     (exp)
      | CASE exp OF mcls END              (Cas(exp, mcls))
      | RECEIVE mcls END                  (Rcv(mcls))
      | OPENCB exps CLOSECB               (TpE(exps))
      | OPENSB nseq BAR nseq CLOSESB      (CnE(nseq1, nseq2))
```

```
    | OPENSB nseq CLOSESB              (CnE(nseq, Val(Nil())))
    | OPENSB CLOSESB                   (Val(Nil()))
    | VAR                              (Val(Var(VAR)))
    | ATOM                             (Val(Atm(ATOM)))
    | NUMBER                           (Val(Num(NUMBER)))
    | pid                              (Val(Pid(pid)))
    | ...

-- Encode function and tuple arguments as list
exps: EMPTY                            (Val(Nil()))
    | nseq                             (CnE(nseq, Val(Nil())))
    | nseq COMMA exps                  (CnE(nseq, exps))

-- Values
val : OPENCB vals CLOSECB              (TpV(vals))
    | OPENSB val BAR val CLOSESB       (CnV(val1, val2))
    | OPENSB val CLOSESB               (CnV(val, Nil()))
    | OPENSB CLOSESB                   (Nil())
    | ATOM                             (Atm(ATOM))
    | NUMBER                           (Num(NUMBER))
    | pid                              (Pid(pid))
    | ...

-- Encode tuple arguments as list
vals: EMPTY                            (Nil())
    | val                              (CnV(val, Nil()))
    | val COMMA vals                   (CnV(val, vals))

-- Pids
pid : FST                              (Fst())
    | NXT OPENB pid CLOSEB             (Nxt(pid))

mcls: ...

pat : ...

-- Actions
act : TAU                              (Tau())
    | pid BANG val                     (Out(pid, val))
    | SPAWN OPENB val COMMA val CLOSEB (Spn(val1, val2))
...
```

The abstract syntax tree representation (given by the expression in brackets following the respective production) uses the operators declared in the `cons` section. It essentially mimics the concrete syntax. The only exception is the assignment expression, which is translated into a `case` construct in order to simplify the semantic treatment. E.g., the expression $\{\mathtt{X},\mathtt{Y}\} = \mathtt{f}(1,2)$ is en-

coded by the operator term representing `case f(1, 2) of {X, Y} -> {X, Y} end`. (For a thorough explanation of the assignment expression and the pattern–matching mechanism in Erlang see [1].)

In the `SEMANTIC` part the rewriting rules in R, which define the transitional behavior of the terms, have to be given. Here, in contrast to the pure Rewriting Logic formalism, explicit transition labels can be used. They are automatically encoded by appropriate term structures.

We give the complete specification, starting with the variable declaration and with those rules that describe purely functional computations without any side effect on the process level. In the transition system, these are labeled by `tau` actions.

```
SEMANTIC

vars
  q, q1, q2, q': value
  v, v1, v2, v3, f: value
  o, o1, o2: value
  e, e1, e2, e': expression
  p, p1, p2, p': pid
  a: action
  cs: (clause list)

rules
  -- Sequence

              <e1 | p | q | o> -(a)-> <e' | p | q' | o>
  (Seq1)-----------------------------------------------------
         <e1, e2 | p | q | o> -(a)-> <e', e2 | p | q' | o>

  (Seq2)-------------------------------------------
         <v, e | p | q | o> -(tau)-> <e | p | q | o>

  -- Application

              <e1 | p | q | o> -(a)-> <e' | p | q' | o>
  (App1)-----------------------------------------------------
         <e1(e2) | p | q | o> -(a)-> <e'(e2) | p | q' | o>

            <e | p | q | o> -(a)-> <e' | p | q' | o>
  (App2)-------------------------------------------
         <f(e) | p | q | o> -(a)-> <f(e') | p | q' | o>

                          Def(f, cs) in Set
  (App3)-------------------------------------------------------------
         <f(v) | p | q | o> -(tau)-> <case v of cs end | p | q | o>
```

```
-- Case (covers assignment as well)

       <e | p | q | o> -(a)-> <e' | p | q' | o>
(Cas1)-----------------------------------------
       <case e of cs end | p | q | o> -(a)->
       <case e' of cs end | p | q' | o>

                  ("maybe cmatch VCS(v, cs) e'")
(Cas2)-------------------------------------------------------
       <case v of cs end | p | q | o> -(tau)-> <e' | p | q | o>

-- Tuple

        <e | p | q | o> -(a)-> <e' | p | q' | o>
(Tup)-------------------------------------------
      <{e} | p | q | o> -(a)-> <{e'} | p | q' | o>

-- List

            <e1 | p | q | o> -(a)-> <e' | p | q' | o>
(Lst1)-----------------------------------------------
       <[e1|e2] | p | q | o> -(a)-> <[e'|e2] | p | q' | o>

           <e | p | q | o> -(a)-> <e' | p | q' | o>
(Lst2)-----------------------------------------------
       <[v|e] | p | q | o> -(a)-> <[v|e'] | p | q' | o>
```

Since SLC does not provide any automatic support for reduction contexts, all possible situations regarding the evaluation state of the subexpressions have to be considered. In the case of the sequencing construct, for example, this means that the head expression first has to be reduced to normal form using (`Seq1`). Thereafter (`Seq2`) can be applied which removes the result. (Note that `v` represents a value.)

Applicative expressions are handled in a similar way where the function (`App1`) and its arguments (`App2`) are reduced to normal form. Then the condition "`Def(f, cs) in Set`" in (`App3`) looks for a definition of the atom `f` in the function environment, returning the corresponding list of clauses which are matched against the argument `v` using a `case` expression.

The `case` construct is described by first reducing the expression to be tested to normal form (`Cas1`), and by invoking the HASKELL function `cmatch` afterwards using a special form of conditions which provides access to the HASKELL environment (`Cas2`). The definition of `cmatch` is omitted here; it tests whether the value `v` matches one of the clauses in `cs`, and returns its correspondingly instantiated right–hand side `e'` in this case. Otherwise, the condition fails. Tuples and lists are handled as one would expect: their com-

ponents are just reduced to normal form (`Tup`, `Lst1`, `Lst2`).

Note that those of the above rules which do not describe the actual computation step but which implement the reduction context (like `Seq1`) use a variable action label `a`. The reason is that the rules which establish the connection between the purely functional expression–level evaluations and the process level employ special labels to indicate the corresponding side effect. These rules are given in the following.

```
-- Process creation

(Spn1)-------------------------------------------
       <spawn(f, v) | p | q | p'> -(spawn(f, v))->
       <p' | p | q | p'>

       <e | p | q | p'> -(spawn(f, v))-> <e' | p | q' | p'>
(Spn2)-----------------------------------------------------
       [p'] || <e | p | q | o> -(tau)->
       [nxt(p')] || <e' | p | q' | o> || <f(v) | p' | [] | o>

-- Send

          <e1 | p | q | o> -(a)-> <e' | p | q' | o>
(Snd1)-----------------------------------------------
       <e1!e2 | p | q | o> -(a)-> <e'!e2 | p | q' | o>

         <e | p | q | o> -(a)-> <e' | p | q' | o>
(Snd2)-------------------------------------------
       <v!e | p | q | o> -(a)-> <v!e' | p | q' | o>

(Snd3)-------------------------------------------------
       <p!v | p | q | o> -(tau)-> <v | p | q @ [v] | o>

(Snd4)-------------------------------------------------
       <p2!v | p1 | q | o> -(p2!v)-> <v | p1 | q | o>

       <e1 | p1 | q1 | o1> -(p2!v)-> <e' | p1 | q' | o1>
(Snd5)-----------------------------------------------------
       <e1 | p1 | q1 | o1> || <e2 | p2 | q2 | o2> -(tau)->
       <e' | p1 | q' | o1> || <e2 | p2 | q2 @ v | o2>

-- Receive

            ("maybe qmatch VCS(q, cs) EVV(e', q1, q2)")
(Rcv)-------------------------------------------------------------
      <receive cs end | p | q | o> -(tau)-> <e' | p | q1 @ q2 | o>
```

```
  -- self()

  (Slf)-----------------------------------------------
        <self() | p | q | o> -(tau)-> <p | p | q | o>

end
```

Process creation is modeled as follows (cf. also the abstract rule in Section 2). Using (App1, App2), both the function to be spawned, f, and its argument list, v, are computed. Now (App3) is not applicable since spawn is not a general atom but a recognized keyword. Instead (Spn1) is used to signal that a new process should be created. In (Spn2) this action is picked up on the process–system level where the next free pid, p', is passed to the spawning process where p' is returned as the result of the evaluation. On the outer level, a new process with pid p' and empty mailbox is created, and the pid counter is incremented.

Here (and also in (Snd5)) it is important that the parallel operator is declared to be associative and commutative (by using the AC annotation in the above cons section). Thus we can always assume that the counter and the spawning process can be arranged to meet the given order. Moreover it is compatible (CO) which means that the computation steps in a parallel subcomponent are applicable to the complete process system.

The introduction of the fourth process component is required by the dataflow restrictions which SLC imposes on transition rules. As explained in Section 3, any variable occurring on the right–hand side of a rule must be bound either by the left–hand side or by the conditions. This would not be the case in (Spn1) if p' did not occur on the left–hand side.

A similar mechanism is employed for process communication. As before, the normal form of the output expression is established using (Snd1) and (Snd2). Then we distinguish whether the message is being sent to the local or to some remote process. In the former case, it is simply appended to the mailbox (using (Snd3); "@" denotes the queue concatenation operator). In the latter case, the output step is signalled to the process level (Snd4) where the message is stored in the mailbox of the receiving process (Snd5).

The processing of this message by the receiver is described in (Rcv): qmatch is a HASKELL function which determines the first message in the mailbox q that matches any of the clauses in cs, and returns the instantiated right–hand side (e') and the two sublists preceding (q1) and following (q2) the matching value, whose concatenation forms the new mailbox. Again the condition fails if none of the values in q matches any of the clauses in cs.

Finally the self function gives access to the pid of the local process, which is required to establish networks of linked processes. It is straightforward to model it in our setting (Slf).

The description concludes with the oriented equations in *ER* which identify certain states of the transition system, using the variables declared above.

Here they are used to define the queue concatenation operator, and to promote the `Val` tag through tuples and lists.

```
equations
  -- Queue concatenation
  (Pls1)
    [] @ q -> q
  (Pls2)
    [v|q1] @ q2 -> [v|q1 @ q2]

  -- Promote value tag through tuples and lists
  (Val1)
    TpE(Val(v)) -> Val(TpV(v))
  (Val2)
    CnE(Val(v1), Val(v2)) -> Val(CnV(v1, v2))

end
```

5 Conclusions

From the previous section it should have become clear that Rewriting Logic provides an adequate framework to formally describe the syntax and semantics of the Erlang programming language. Its constructs can be specified in an elegant way, and the compiler–generating approach adopted by SLC supports an incremental working style: it is easy to modify an existing definition in order to correct errors, to change the semantics, or to integrate new syntactic constructs. Thus SLC can be used as a rapid prototyping tool for language implementations (see also [9]).

First preliminary experiences with the obtained Erlang frontend in TRUTH show that it works rather slow, mainly due to the complexity implied by conditional equational term rewriting. Here we see some demand, but also potential for optimizations.

Moreover, in order to remove the rather artificial separation between Erlang expressions and values (cf. Section 4), it would be nice to extend SLC to an order–sorted framework, as it is employed e.g. by the Maude system [5]. Another modification concerns the use of SLC as frontend generator for the Erlang Verification Tool EVT [2].

Besides this work on the tool side, we are planning to exploit the expressive features of the Rewriting Logic framework to reduce the state space of the transition system. For example, purely functional evaluations without side effects (such as function calls) can be represented by (directed) equations. In this way they are completely hidden within the current state. Furthermore equations can be used to define abstraction mappings on the states, again reducing the state space (in some cases even from infinite to finite size).

Another issue is the integration of specific architectural components for

software systems, such as generic server components from OTP. Our idea is to specify a set of transition rules which characterize the abstract behaviour of the generic server functions. For example, the following transition rule describes the situation that a client process (with pid p) sends a request *req* to an idling server (with pid sp). The client then changes into a waiting state, while the server processes the request by executing the `handle_call` function.

$$\begin{aligned} &\langle r[\texttt{call}(sp, req)] \mid p \mid q\rangle \parallel \langle \texttt{ready}(state) \mid sp \mid sq\rangle \\ \xrightarrow{\tau}\ &\langle r[\texttt{wait}(sp)] \mid p \mid q\rangle \parallel \langle \texttt{busy}(\texttt{handle\_call}(req, p, state), p) \mid sp \mid sq\rangle \end{aligned}$$

A complete formal model has been implemented in EVT [3], and we do not expect any problems in developing a corresponding SLC specification. In this way we can reason in a compositional way about any client–server application without having to consider the concrete implementation details of the generic part, which simplifies formal reasoning dramatically. It might even allow to reduce an infinite–state system to finite size since generic servers employ synchronous communication, which can be modeled ignoring the mailbox queue (in the above rule, neither q nor sq is touched). Thus one potential source of infinite–state behaviour is eliminated.

References

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

[2] Arts, T., M. Dam, L. Fredlund and D. Gurov, *System description: Verification of distributed Erlang programs*, in: *Proceedings of CADE'98*, Lecture Notes in Artificial Intelligence **1421** (1998), pp. 38–41.

[3] Arts, T. and T. Noll, *Verifying generic Erlang client–server implementations*, Technical Report 00–08, Aachen University of Technology, Aachen, Germany (2000), `ftp://ftp.informatik.rwth-aachen.de/pub/reports/2000/00-08.ps.gz`.

[4] Clarke, E. and J. Wing, *Formal methods: State of the art and future directions*, Technical Report CMU–CS–96–178, Carnegie Mellon University, Pittsburg, USA (1996), `http://reports-archive.adm.cs.cmu.edu/anon/1996/CMU-CS-96-178.ps`.

[5] Clavel, M., F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer and J. Quesada, *The Maude system*, in: *Proceedings of the 10th International Conference on Rewriting Techniques and Applications (RTA'99)*, Lecture Notes in Computer Science **1631** (1999), pp. 240–243.

[6] Dam, M., L. Fredlund and D. Gurov, *Toward parametric verification of open distributed systems*, in: *Compositionality: the Significant Difference*, Lecture Notes in Computer Science **1536** (1998), pp. 150–185.

[7] Huch, F., *Verification of Erlang programs using abstract interpretation and model checking*, ACM SIGPLAN Notices **34** (1999), pp. 261–272, proceedings of the ACM SIGPLAN International Conference on Functional Programming (ICFP '99).

[8] Lange, M., M. Leucker, T. Noll and S. Tobies, *Truth – a verification platform for concurrent systems*in: *Tool Support for System Specification, Development, and Verification*, Advances in Computing Science, Springer-Verlag Wien New York, 1999 pp. 150–159.

[9] Leucker, M. and T. Noll, *Rapid prototyping of specification language implementations*, in: *Proceedings of the 10th IEEE International Workshop on Rapid System Prototyping* (1999), pp. 60–65.

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

[11] Meseguer, J., *Rewriting logic as a semantic framework for concurrency: a progress report*, in: *Seventh International Conference on Concurrency Theory (CONCUR'96)*, Lecture Notes in Computer Science **1119** (1996), pp. 331–372.

[12] Middeldorp, A. and E. Hamoen, *Completeness results for basic narrowing*, Journal of Applicable Algebra in Engineering, Communication and Computing **5** (1994), pp. 313–353.

[13] Noll, T., *On coherence properties in term rewriting models of concurrency*, in: *Proceedings of the 10th International Conference on Concurrency Theory (CONCUR'99)*, LNCS **1664** (1999), pp. 478–493.

[14] *CCITT Specification and Description Language (SDL)*, Technical Report 03/93, International Telecommunication Union (1993), `http://www.itu.int/`.

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

[16] Viry, P., *Rewriting modulo a rewrite system*, Technical Report TR-95-20, Dipartimento di Informatica, Universita di Pisa (1995), `ftp://ftp.di.unipi.it/pub/techreports/TR-95-20.ps.Z`.