Available online at www.sciencedirect.com
SCIENCE DIRECT
Electronic Notes in Theoretical Computer Science
ELSEVIER Electronic Notes in Theoretical Computer Science 118 (2005) 145–162
www.elsevier.com/locate/entcs

Equational Abstractions for Model Checking Erlang Programs

Thomas Noll[1]

Lehrstuhl für Informatik II
Aachen University
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. In particular it demonstrates the use of equations for defining abstraction mappings which reduce the state space of the system.

Keywords: Software Verification, Concurrent Functional Programming, Rewriting Logic

1 Introduction

In this paper we address the software verification issue in the context of the functional programming language Erlang [2], which was developed by the 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 supports the application of formal reasoning methods.

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

doi:10.1016/j.entcs.2004.01.037

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 [5,6] to establish the desired system properties.

Here we follow an alternative approach in which we try to employ fully–automatic *model–checking techniques* to establish correctness properties of communication systems implemented in Erlang. Here we concentrate on the first part of the verification procedure, the construction of the (transition–system) model to be checked.

More concretely, we formally describe Erlang using the *Rewriting Logic* framework, which was proposed in [11] as a unified semantic framework for concurrency. It has proven to be an adequate modeling formalism for many concrete specification and programming languages [10]. 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. In particular we will discuss examples where it is possible to shrink a system with infinitely many states to a finite one.

By employing an executable implementation of the Rewriting Logic framework such as the ELAN tool [4] it is possible to automatically derive the transition system of a given Erlang program. Thereafter model–checking tools such as TRUTH [8] can be used to automatically verify that the system meets certain conditions given as formulae of some mathematical logic. The latter, however, is outside the scope of this article.

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 introduces the Rewriting Logic framework, and employs it to define the transition–system semantics of Erlang. Then Section 4 demonstrates the use of equations to reduce the size of the transition system, and finally Section 5 concludes with some remarks.

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 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. Real Erlang has several additional features such as modules, distribution of processes (onto nodes), and support for robust programming and 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 Erlang expressions is summarized as follows:

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

Here X 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; e_1, e_2 denotes sequential composition; and $e(e_1, \ldots, e_n)$ represents a function call. An expression of the form $\texttt{case}\ e\ \texttt{of}\ p_1\ \texttt{->}\ e_1; \ldots; p_n\ \texttt{->}\ e_n\ \texttt{end}$ 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 behavior (i.e., side effects) are $e_1\texttt{!}e_2$ which denotes an output step, sending the value of e_2 asynchronously to the process identified by e_1, whereas $\texttt{receive}\ cs\ \texttt{end}$ 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 $\texttt{case}\ v\ \texttt{of}\ cs\ \texttt{end}$. $\texttt{spawn}(e_1, e_2)$ 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 $\texttt{self}()$ returns the pid of the local process.

As an introductory example we consider a short Erlang 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. An extended version of the algorithm is presented in [3], which in addition is capable of handling several resources from a given, finite set.

An Erlang program consists of a set of modules. Each module basically contains a list of function declarations. In our example the system is defined in one module. It is initialized using the `start` function, which, according to the `export` declaration, is the only function accessible from outside the `locker` module. By calling the corresponding startup functions, it generates three new processes: one locker and two clients. The actual process creation is performed by the `spawn` builtin function which receives the module identifier and the name of the function to be invoked in the new process, together with its arguments.

The locker process runs the `locker` function in a non–terminating loop. It employs the `receive` construct to check whether a request message has arrived. The latter is expected to be a pair composed of a `request` tag and a client process identifier (which is matched by the variable `Client`). The client is then granted access to the resource by sending an `ok` flag. Finally, after receiving the 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, the `self` builtin function returns the process identifier (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.

```
-module(locker).
-export([start/0]).

start() ->
  Locker = start_locker(),
  start_client(Locker),
  start_client(Locker).

start_locker() ->
  spawn(locker, []).

locker() ->
  receive
    {request, Client} ->
      Client!ok,
      receive
        {release, Client} ->
          locker()
      end
  end.
```

```
start_client(Locker) ->
  spawn(client, [Locker]).

client(Locker) ->
  Locker!{request, self()},
  receive
    ok ->
      % critical section
      Locker!{release,self()},
      client(Locker)
  end.
```

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 see how to check these properties by constructing the transition system of the above program. Using the latter, the absence of deadlocks can be verified by showing that the system can always proceed, i.e., that every state has a direct successor. Mutual exclusion can be established by proving that between the receptions of two successive `request` messages by the locker there must always occur a `release` operation.

Guaranteeing the no–starvation property, however, needs additional assumptions about the behavior of the process scheduler. In principle it could happen that one of the client processes indefinitely remains in its initial state,

i.e., is never scheduled for sending the request message to the locker. This situation, however, is excluded by the requirements which have to be met by all Erlang implementations (cf. [2, Sct. 5.6]). First, the scheduling algorithm must be *fair*, i.e., any process which is enabled for execution will eventually be run. Moreover no process will be allowed to block the machine for a longer period. This is postulated since Erlang should be suitable for *soft real–time* applications where response times must lie in the order of milliseconds.

As soon as the client has sent the request to the locker, eventual access to the resource is guaranteed since the implementation of the (locker) mailbox in the Erlang runtime system follows a FIFO policy.

3 Formal Semantics of Erlang

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.

3.1 The Rewriting Logic Framework

The Rewriting Logic framework has been presented by J. Meseguer in [11]. An introduction to this approach together with an extensive bibliography can be found in [10].

Rewriting Logic is intended to serve as a unifying mathematical model and uses notions from rewrite systems over equational theories. It aims at a separate description of the static and of 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. Since a single transition may comprise several independent rewriting steps, concurrent behavior can explicitly be modelled in this way.

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

- Σ is a signature, i.e., a ranked alphabet of function symbols,
- $E \subseteq T_\Sigma(Var) \times T_\Sigma(Var)$ is a finite set of equations over the set $T_\Sigma(Var)$ of Σ–terms with variables from a given set Var,
- L is a finite set of symbols called *(rule) labels*, and

- $R \subseteq L \times (T_\Sigma(Var) \times T_\Sigma(Var))^+$ is a finite set of *(conditional) transition rules* where each $(\rho, (l \longrightarrow r)(c_1 \longrightarrow d_1) \ldots (c_k \longrightarrow d_k)) \in R$ is represented as

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

  Here l and r are called the *left–hand side* and the *right–hand side*, respectively, of the rule. The upper part is called its *condition*, and may sometimes be abbreviated with the letter C. If $k = 0$, then the rule is called *unconditional*.

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 presented by terms and which are evolving by means of transitions. Here, the states are structured according to the signature, and the transition rules specify the local transitions in this structure whereas the deduction rules allow to reason about the overall behavior of the concurrent system given the local transformations.

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 [14], we therefore propose to decompose E into a set of directed equations (that is, a term rewriting system), ER, and into a set AC expressing associativity and commutativity of certain binary operators in Σ. Given that ER is terminating modulo AC, rewriting by R modulo E can be implemented by a combination of normalizing by ER and rewriting by R, both modulo AC. 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.

3.2 A Rewriting Logic Specification of Erlang

As mentioned earlier, the Erlang runtime system maintains a set of user *processes*. Any such process consists of three components: an Erlang *expression* which has to be evaluated, a *process identifier* (pid), which uniquely identifies

the respective process, and which is internally determined by the system, and a *mailbox* for incoming messages, which is essentially a list of Erlang values.

Moreover we will attribute a transition label and a current evaluation environment to a process. The former is used to indicate the type of the transition which lead to the current state. The latter stores the bindings between the Erlang variables and the values assigned to them. It is modified by an assignment or by a pattern matching operation. Syntactic restrictions imposed on the code guarantee that every occurrence of a variable name lies within the scope of a binding operation.

As mentioned earlier, there can be several processes running concurrently in a system. We therefore introduce the notion of a *process system*, which is just a set of concurrent processes, and which constitutes a state in the transition–system semantics:

$$\begin{aligned} S = \{\langle \alpha \mid e \mid i \mid q \mid \rho \rangle \mid{}& \alpha \text{ label}, e \text{ expression}, i \text{ pid}, \\ & q \text{ mailbox}, \rho \text{ environment}\} \\ \cup\, \{\langle i \rangle \mid{}& i \text{ pid}\} \\ \cup\, \{s_1 \parallel s_2 \mid{}& s_1, s_2 \in S\} \end{aligned}$$

The construct $\langle i \rangle$ denotes a *dead* process whose actual computation has been terminated. In contrast a process of the first form is called *live*. Both live and dead processes be combined using the associative and commutative *parallel composition operator* $\parallel$ to obtain a concurrent process system. This, however, makes only sense if every process is uniquely identified by its pid. We therefore call a process system *well formed* if all pids which occur in the process tuples are distinct, and assume every process system to be well formed from now on.

The states of both single processes and process systems evolve over time (e.g., the expression of a process changes due to evaluation, or the mailbox stores an incoming message). The transition labels attached to the processes reflect the kind of the transition that lead to the current state. These include the label τ which expresses that a step not involving a side effect was taken, such as the evaluation of a builtin function like `+`. Other transition labels such as $\mathtt{msg}(j, \mathtt{42})$ describe the sending of the value `42` to the process identified by j, while $\mathtt{spn}(\mathtt{foo}, \mathtt{[1,2]}, j)$ represents the call of a function involving side effects (the `spawn` function in this case). Here it is important to observe that transition labels are not just comment–like annotations to the processes. Rather they implement a means of communication between the two levels of the semantics, single processes and concurrent systems, thus determining the transitions a concurrent process system can take as a whole.

To simplify the presentation we refrain from formalizing those aspects of

Erlang which are related to its module system. Thus we always assume that the body of a function can somehow be determined from its module identifier and its name.

3.3 The Equational Theory

The next step involves the definition of the set of equations, E, of our rewrite theory. In the AC part we only need to declare the parallel operator $\|$ to be associative and commutative. The set of directed equations, ER, is used to model some auxiliary functions which are employed in the transition rules. Due to lack of space we refrain from showing the specification and refer to [1] instead. Let us just mention that we obtain a term rewriting system which is convergent modulo AC.

3.4 The Transition Rules

The most important part of our definition is the formalization of the operational behavior of Erlang process systems by conditional transition rules. To obtain a cleaner structure we decompose R into two disjoint subsets:

$$R = R_{Prc} \cup R_{Sys}.$$

Here, R_{Prc} contains the so–called *process–level rules* which operate on single processes while R_{Sys}, the set of *system–level rules*, deals with concurrent process systems. In the following we present some examples from both categories, again referring to [1] for the complete definition.

As before we will use certain standard denotations for the Rewriting Logic variables occurring in the rules, possibly in indexed or primed form. We let e denote an Erlang expression, p denote a pattern, X denote an Erlang variable, a denote an atom, v denote a value, f denote a function name, and c denote a clause. Moreover α refers to a transition label, cs to a list of clauses, i, j, k to pids, q to a mailbox, ρ to an environment, and s to a concurrent process system.

3.4.1 Expression–Level Rules

The first rule describes the recursive evaluation of lists. Due to the leftmost–innermost evaluation strategy of Erlang we have to start the evaluation with the first expression in the list constructor.

$$\frac{\langle \tau \mid e_1 \mid i \mid q \mid \rho \rangle \longrightarrow \langle \alpha \mid e_1' \mid i \mid q' \mid \rho' \rangle}{\langle \tau \mid [e_1 \mid e_2] \mid i \mid q \mid \rho \rangle \longrightarrow \langle \alpha \mid [e_1' \mid e_2] \mid i \mid q' \mid \rho' \rangle} \,(\mathrm{list}_1)$$

In general we assume in our formalization that a process–level rule is only applicable if the latest transition label of the respective process was τ. Oth-

erwise the label indicates a side effect which should be handled by another appropriate (system–level) rule.

As soon as the first subexpression of the list constructor is irreducible (i.e., a value), the evaluation proceeds with the second subexpression:

$$\frac{\langle \tau \mid e \mid i \mid q \mid \rho\rangle \longrightarrow \langle \alpha \mid e' \mid i \mid q' \mid \rho'\rangle}{\langle \tau \mid \texttt{[}v\texttt{|}e\texttt{]} \mid i \mid q \mid \rho\rangle \longrightarrow \langle \alpha \mid \texttt{[}v\texttt{|}e'\texttt{]} \mid i \mid q' \mid \rho'\rangle}\ (\mathrm{list}_2)$$

The following rules deal with pattern–matching operations. Here we just formalize the `case` construct.

$$\frac{\langle \tau \mid e \mid i \mid q \mid \rho\rangle \longrightarrow \langle \alpha \mid e' \mid i \mid q' \mid \rho'\rangle}{\begin{array}{c}\langle \tau \mid \texttt{case}\ e\ \texttt{of}\ cs\ \texttt{end} \mid i \mid q \mid \rho\rangle \\ \longrightarrow \langle \alpha \mid \texttt{case}\ e'\ \texttt{of}\ cs\ \texttt{end} \mid i \mid b' \mid \rho\rangle\end{array}}\ (\mathrm{case}_1)$$

$$\frac{\mathsf{match}_{\mathsf{c}}(v, cs, \rho) = (e, \rho')}{\langle \tau \mid \texttt{case}\ v\ \texttt{of}\ cs\ \texttt{end} \mid i \mid q \mid \rho\rangle \longrightarrow \langle \tau \mid e \mid i \mid q \mid \rho'\rangle}\ (\mathrm{case}_2)$$

Finally we consider one of the Erlang builtin functions which evoke side effects on the system level of the semantics.

$$\frac{j = \mathsf{newPid}()}{\langle \tau \mid \texttt{spawn(}a\texttt{,}v\texttt{)} \mid i \mid q \mid \rho\rangle \longrightarrow \langle \mathsf{spn}(a, v, j) \mid j \mid i \mid q \mid \rho\rangle}\ (\mathrm{spawn})$$

Here $\mathsf{newPid}()$ is a function returning a fresh pid which uniquely identifies the new process, and which is returned as the result of the call of `spawn`.

The following rule handles one of the central concepts of Erlang: asynchronous sending of messages. As we shall see the message will be appended to the mailbox of the target process. Note that a process can also send a message to itself.

$$\frac{}{\langle \tau \mid j\,\texttt{!}\,v \mid i \mid q \mid \rho\rangle \longrightarrow \langle \mathsf{msg}(j, v) \mid v \mid i \mid q \mid \rho\rangle}\ (\mathrm{send})$$

3.4.2 System–Level Rules

The first rule just expresses that if a single process in a concurrent system performs a computation step then so does the complete system.

$$\frac{\langle \tau \mid e \mid i \mid q \mid \rho\rangle \longrightarrow \langle \tau \mid e' \mid i \mid q' \mid \rho'\rangle}{\langle \tau \mid e \mid i \mid q \mid \rho\rangle \parallel s \longrightarrow \langle \tau \mid e' \mid i \mid q' \mid \rho'\rangle \parallel s}\ (\mathrm{Silent})$$

Process generation is formalized as follows. The `spawn` builtin function comes with two arguments: a function atom, and a list of arguments. The new process will call this function with these arguments, starting with the empty mailbox and the empty environment.

$$\frac{\langle \tau \mid e \mid i \mid q \mid \rho\rangle \longrightarrow \langle \mathsf{spn}(a, v, j) \mid e' \mid i \mid q' \mid \rho'\rangle}{\begin{array}{c}\langle \tau \mid e \mid i \mid q \mid \rho\rangle \parallel s \\ \longrightarrow \langle \tau \mid e' \mid i \mid q' \mid \rho'\rangle \parallel s \parallel \langle \tau \mid a\texttt{(}v\texttt{)} \mid j \mid \mathsf{nil} \mid \mathsf{nil}\rangle\end{array}}\ (\mathrm{Spawn})$$

Next we specify how a message is stored in the mailbox of the receiving process.

$$\frac{\langle \tau \mid e_1 \mid i \mid q_1 \mid \rho_1 \rangle \longrightarrow \langle \mathsf{msg}(j, v) \mid e_1' \mid i \mid q_1' \mid \rho_1' \rangle}{\begin{array}{l} \langle \tau \mid e_1 \mid i \mid q_1 \mid \rho_1 \rangle \parallel \langle \tau \mid e_2 \mid j \mid q_2 \mid \rho_2 \rangle \parallel s \\ \longrightarrow \langle \tau \mid e_1' \mid i \mid q_1' \mid \rho_1' \rangle \parallel \langle \tau \mid e_2 \mid j \mid q_2 \cdot v \mid \rho_2 \rangle \parallel s \end{array}} \text{(Com)}$$

The next rule handles the situation when a process terminates normally, i.e., evaluates its expression to a value, becoming dead afterwards.

$$\frac{}{\langle \tau \mid v \mid i \mid q \mid \rho \rangle \parallel s \longrightarrow \langle i \rangle \parallel s} \text{(Termination)}$$

Many more rules are required to complete the definition of our rewrite theory $\mathfrak{T}$ for Erlang. Together with an initial expression e_0 (and a collection of modules with function definitions), it defines a labelled transition system as follows.

$$T = (S, s_0, \longrightarrow)$$

where S is the set of states defined by $S = \{s \mid s_0 \longrightarrow^* s\}$ and s_0 is the initial state given by $s_0 = \langle \tau \mid e_0 \mid i_0 \mid \mathsf{nil} \mid \mathsf{nil} \rangle$ for some initial pid i_0.

Note that the state space S is infinite in general, due to several reasons:

- Erlang supports unbounded data structures, such as integers or lists.
- The unbounded use of recursive function calls can give rise to arbitrarily large expressions.
- The mailbox of a process can store an unbounded number of messages.
- Moreover the combination of dynamic process creation with recursive functions gives rise to infinite state spaces.

However employing the implementation of our semantics, which will be sketched in the following, it can be shown that the state space of the locker example from Section 2 is finite. It comprises approximately 180 states. Experimenting with a varying number of client processes shows the (expected) effect that the state space exponentially grows with the number of clients—a typical example for the state–explosion problem in model–checking applications. Approaches to alleviate this problem will be discussed in Section 4.

At first glance it might look surprising that the locker system possesses a finite state space although both the `locker` and the `client` function are recursively defined. This can be explained by the fact that only *tail recursion* is employed, that is, a recursive function call can occur only as the last expression in the body of the respective function. Hence such a call corresponds to a jump to the beginning of the respective function body, and thus the size of the expression which represents the control state of the computation is bounded. In fact tail recursion gives rise to an implementation technique called *last*

call optimization which allows to evaluate such functions in constant memory space (see [13] for the general idea and [2, Sct. 9.1] for Erlang–specific details).

3.5 Implementation in ELAN

To develop an evaluator prototype for Erlang we have chosen an existing implementation of the Rewriting Logic framework, the ELAN tool (cf. [4]), which is being developed within the PROTHEO group at the LORIA research institute in Nancy, France.

The ELAN system provides an environment for specifying and prototyping deduction systems in a language based on rewrite rules whose application can be controlled by strategies. ELAN can be employed either as a logical framework or to describe and execute both deterministic and non–deterministic rule–based processes. Here we exploit the second feature by defining an executable specification of the semantics of Erlang.

System specifications are given in special ELAN modules called *element modules*. In an element module one can import other modules and define the sorts, operators, and rewrite rules of a rewrite theory. Equational reasoning is supported by an efficient AC–rewriting engine. Moreover it is possible to describe *strategies*, which define the way (i.e., the order and the position) in which the rules can be applied to terms. These features proved to be very useful for the implementation of our semantics for Erlang. The details can be found in [1].

As seen before, using this implementation it is possible, e.g., to compute the transition system of the locker example from Section 2 with a varying number of client processes (which are all spawned upon the initial call of the `start` function). This enables us to show that the system indeed exhibits the required properties (mutual exclusion etc.) which were mentioned in the beginning of this chapter. We will come back to this issue in the following section.

4 Equational Abstractions

Note that so far ER, the set of oriented equations, was only used to implement the auxiliary functions occurring in the Rewriting Logic specification of Erlang. This is somehow in contradiction to the initial motivation of introducing equations in order to reduce the number of rewriting rules and/or the state space of the transition system.

In the following we sketch two approaches which target the second goal. The first idea is to "hide" those computations in a given Erlang program which do not involve side effects, using the equational theory. In this way we obtain

an abstracted transition system which represents only those state changes one would like to observe.

More concretely, we move those process–level rules which just produce τ transitions from R_{Prc} to ER, that is, we let

$$\begin{aligned} R_\tau &= \left\{ \frac{}{p \longrightarrow p'} \mid p' \text{ is of the form } \langle \tau \mid \ldots \mid \ldots \mid \ldots \mid \ldots \rangle \right\} \\ &= \{(\mathrm{list}_1), (\mathrm{list}_2), \ldots\} \\ {R_{Prc}}' &= R_{Prc} \setminus R_\tau \\ ER' &= ER \cup R_\tau \end{aligned}$$

In this way, "uninteresting" computations such as the invocation of a function by replacing the call with its body, or the evaluation of a `case` expression, do not unnecessarily increase the state space anymore but are instead hidden in the current state.

Of course the question whether a certain action of the program is interesting or not depends on the application. Therefore the choice of the set R_τ can vary for different verification problems. In any case a prime requirement is that the choice of the abstraction mapping ensures preservation of correctness properties: *false positives* should be excluded, that is, a property checked to be true for the abstract system should also hold for the concrete system being modelled. On the other hand, the term *false negative* refers to the (less critical) case that the abstract system exhibits an error which cannot be traced back to the concrete system. Here the abstraction is chosen too coarse, and the inspection of the error situation may then suggest a way to refine it.

Figure 1 shows the abstracted transition system of the locker example with two clients, as discussed in Section 2. Here start denotes the initial state, consisting of a single process which evaluates the function call `start`(). The locker process is denoted by L, possibly superscripted by client numbers (1 and/or 2) to indicate the contents of its message mailbox. For example, L^{12} represents a locker process which has stored `request` messages from the first and the second client (in that order) in its mailbox. The clients are denoted by C_1 and C_2 where a bar indicates that the respective client is in the critical section.

The transitions are labelled to indicate the type of action which caused the change of the state. Here spn refers to the creation of a process, req_i and rel_i indicate that C_i sends a `request` or, respectively, a `release` message to the locker (where $i \in \{1, 2\}$), and ok_i denotes the admission from the locker to enter the critical section.

Thus the abstracted system contains 14 states while the standard transition

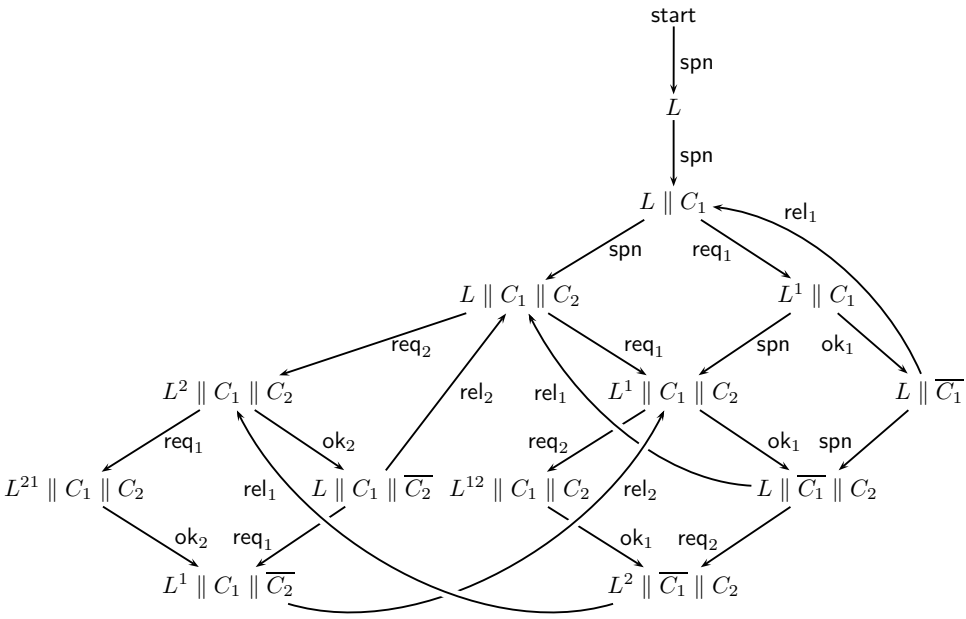

Fig. 1. Abstracted Locker Transition System

Locker with	1 client	2 clients	3 clients
Original LTS	65 states	182 states	536 states
	10 min	55 min	380 min
Abstracted LTS	5 states	14 states	42 states
	20 sec	90 sec	13 min

Table 1
Original vs. Abstracted Locker Systems

system derived in the original semantics comprises 182 states. Table 1 shows, for several numbers of clients, both the size of the state space and the time required to compute it. The results are very promising, inviting to further investigate the benefits of equational abstractions for Erlang programs.

As mentioned earlier, using the formal semantics we can derive that our implementation of the locker behaves correctly. More concretely, the (abstracted) transition system from Figure 1 possesses the following properties:

no deadlock: there exists no cyclic chain of processes waiting for each other

to continue, which can be deduced from the fact that every state in the system has a direct successor,

mutual exclusion: no two clients can gain access to the resource at the same time since in every state there is at most one client process in the critical section (i.e., marked with a bar), and

no starvation: every client enabled to enter the critical section eventually gets its demanded access. At first glance this property seems to be violated since there exists a (reachable) cycle in the system in which the second client is never active:

$$\mathsf{start} \xrightarrow{\mathsf{spn}} L \xrightarrow{\mathsf{spn}} L \parallel C_1 \xrightarrow{\mathsf{req}_1} L^1 \parallel C_1 \xrightarrow{\mathsf{ok}_1} L \parallel \overline{C_1} \xrightarrow{\mathsf{rel}_1} L \parallel C_1 \xrightarrow{\mathsf{req}_1} \dots$$

(A similar scenario can be constructed to show that also the first client could starve.)

However this cycle indefinitely excludes the `start` function in the main process from spawning the second client process, and is therefore in contradiction to the requirements which are postulated for every implementation of the Erlang runtime system: any process which is enabled for execution will eventually be run, using a bounded time slice. Thus we can indeed guarantee that C_2 will be spawned and, moreover, that C_2 will obtain the chance to send a `request` message to the locker. In other words, the system will eventually reach a state in which the index 2 appears in the mailbox superscript of the locker process. The same applies to the first client.

To establish the no–starvation property it therefore suffices to show that, for each $i \in \{1, 2\}$, from every state in which the locker is superscripted by i, each possible continuation will eventually pass through an action of the form ok_i. This is easily verified by inspecting the transition system.

Of course it is possible to have these properties automatically be checked by suitable tools such as Truth.

Let us now turn towards the second approach which employs directed equations to define abstraction mappings. The following example shows that such equations can sometimes be used to reduce an infinite to a finite system. The code fragment given below implements a simple concurrent server which repeatedly accepts an incoming query in the form of a triple which is tagged by the atom `request`, and which contains the request itself (matched by the variable `Request`) and the pid of the client process (`Client`). It then spawns 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` atom.

```
concurrent_server() ->
  receive
    {request, Request, Client} ->
      spawn(serve, [Request, Client])
  end,
  concurrent_server().

serve(Request, Client) ->
  Client!{response, handle(Request)}.
```

Having server requests handled in this way, i.e., by separate concurrent subprocesses, represents a programming technique which is widely used in Erlang applications. Apart from the fact that the load of the host running the server process can easily be balanced by controlling the number of concurrent processes, it offers the great advantage that the computations of the server itself and of the single requests are kept in isolation, and cannot influence each other therefore (cf. [2, Chapter 8]). However it has the consequence that after every completion of a serving process a dead process remains in the system. Hence if the total number of calls to the concurrent server is not bounded, the number of system states is not bounded either.

Using the reduction rule $s \parallel \langle i \rangle \longrightarrow s$ (where s denotes a system and i a pid) it is possible to obtain a finite transition system, assuming that the number of simultaneous calls to the server is bounded, and that the processing of a single request involves only finite behavior.

5 Conclusions

In this paper we have proposed a variant of Meseguer's Rewriting Logic as a semantic framework in which the operational semantics of the Erlang programming language can be formalized. In particular we have seen that the equational reasoning allows to test and to compare different abstractions to reduce the size of the state space. Thus it avoids the error–prone and slow construction of abstract system models by hand.

The concrete runtime figures have shown that a lot remains to be done with respect to the optimization of both the tools implementing the Rewriting Logic framework (ELAN in our case) and the description of the programming language under consideration, which usually involves rather complex semantic structures.

Of course one should not be over–optimistic in that respect. There will always be a certain complexity implied by conditional equational term rewriting modulo associativity and commutativity, which is the essential basis of our

framework. It is therefore illusory to expect that an evaluator which has been automatically derived from the Rewriting Logic definition of a specification language will achieve the efficiency of a hand–written, optimized implementation. Instead the "executable specification" approach proves its strength with respect to flexibility and user–friendliness. Thus ELAN and similar systems should be regarded as rapid–prototyping tools for frontends of verification tools. This aspect is being discussed in [9].

Further techniques for formally abstracting Erlang programs with infinite state spaces to finite–state form, followed by fully automated model checking, are investigated by F. Huch in [7]. He employs the technique of *abstract interpretation* to define finite–domain abstractions for unbounded data structures. This enables him to show that the abstract interpretation of an Erlang program in which only tail recursion is employed, which spawns only a finite number of processes, and which uses only finite parts of the mailboxes induces a finite transition system, and is thus amenable to classical model–checking methods.

Since an infinite state space can not always be reduced to a finite one without losing "essential" information, it is natural to employ interactive *theorem–proving assistants* such as the EVT Erlang Verification Tool (see [5,12,6]) to establish the desired system properties in such situations. In any case the semantics of Erlang has to be implemented, mapping Erlang programs to transition systems. (Additionally, the abstraction functions have to be implemented for the model–checking approach.) Here again our compiler–generating approach can be used for automatically deriving the verification tool frontend.

References

[1] V. Amiranashvili. A rewriting logic formalization of Core Erlang semantics. Master's thesis, Aachen University of Technology, Germany, 2002.

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

[3] T. Arts, C.B. Earle, and J. Derrick. Verifying Erlang code: a resource locker case study. In *Formal Methods – Getting IT Right*, volume 2391 of *Lecture Notes in Computer Science*, pages 184–203. Springer–Verlag, 2002.

[4] ELAN home page. http://www.loria.fr/ELAN/.

[5] L.–Å. Fredlund, D. Gurov, and T. Noll. Semi–automated verification of Erlang code. In *16th IEEE International Conference on Automated Software Engineering (ASE'01)*, pages 319–323. IEEE Computer Society Press, 2001.

[6] L.-Å. Fredlund, D. Gurov, T. Noll, M. Dam, T. Arts, and G. Chugunov. A verification tool for Erlang. *International Journal on Software Tools for Technology Transfer*, 4(4):405–420, 2002.

[7] F. Huch. Verification of Erlang programs using abstract interpretation and model checking. *ACM SIGPLAN Notices*, 34(9):261–272, 1999.

[8] M. Lange, 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, pages 150–159. Springer–Verlag Wien, 1999.

[9] M. Leucker and T. Noll. Rapid prototyping of specification language implementations. In *Proceedings of the 10th IEEE International Workshop on Rapid System Prototyping (RSP'99)*, pages 60–65. IEEE Computer Society Press, 1999.

[10] N. Martí–Oliet and J. Meseguer. Rewriting logic: Roadmap and bibliography. *Theoretical Computer Science*, 285(2):121–154, 2002.

[11] J. Meseguer. Conditional rewriting logic as a unified model of concurrency. *Theoretical Computer Science*, 96(1):73–155, 1992.

[12] T. Noll, L.-Å. Fredlund, and D. Gurov. The Erlang Verification Tool. In *Proceedings 7th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'01)*, volume 2031 of *Lecture Notes in Computer Science*, pages 582–585. Springer–Verlag, 2001.

[13] G.L. Steele. Debunking the "expensive procedure call" myth, or procedure call implementations can be considered harmful, or LAMBDA, the ultimate GOTO. In *ACM Conference Proceedings*, pages 153–162, 1977.

[14] P. Viry. Rewriting: An effective model of concurrency. In *Proceedings of PARLE'94 – Parallel Architectures and Languages Europe*, volume 817 of *Lecture Notes in Computer Science*, pages 648–660. Springer–Verlag, 1994.