A Distributed Model Checking Tool Tailored to Erlang

Martin Leucker and Thomas Noll

Lehrstuhl für Informatik II
Aachen University of Technology (RWTH)
D–52056 Aachen, Germany
{leucker,noll}@informatik.rwth-aachen.de

Abstract

Erlang is a concurrent functional programming language, mainly being employed in telecommunication systems. To satisfy the high quality demands in this application area, formal methods are becoming more and more popular. We provide a contribution to this field by outlining the design of a tool supporting the verification of Erlang programs by means of model–checking methods. It includes a novel, distributed model checker which validates the given program with respect to a user–defined correctness property, specified in a temporal logic.

1 Introduction

Erlang is a concurrent functional programming language developed by Ericsson, which is mainly being used for implementing telecommunication systems [1]. The high quality demands on software for such applications may partly be ensured by the use of formal methods in the design and development. By the high degree of concurrency which is usually present, *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 [8] for an overview).

In view of the inherent complexity of formal methods it is desirable to provide the user with tool support. It is even indispensable for the design of safety–critical concurrent systems where an *ad hoc* or conventional software engineering approach is not justifiable. There is one particularly successful automated approach to verification, called *model checking*, in which one tries to prove that (a mathematical model of) a system has certain properties specified in a suitable logic.

Despite the undoubted improvements of model checking techniques in recent years, the so–called *state space explosion problem* still limits their application. Although techniques such as *partial–order reduction* or *symbolic model checking* reduce the state space by orders of magnitude, typical verification tasks still take days on a single workstation or are even unmanageable due to memory restrictions.

On the other hand, cheap yet powerful distributed computer systems can be constructed by building networks of workstations (*NOWs*). From the outside, a NOW appears as one single parallel computer with high computing power and, even more important, huge amount of memory. This enables concurrent programs to utilize the accumulated resources of a NOW to solve large problems.

In this paper we present a prototype of a distributed model–checking tool which is tailored to the Erlang programming language. It is based on the approach taken by Arts and Earle in [2] where the given Erlang program is first translated into μCRL, a process algebra with data [4]. Then the μCRL tool set is used to compute the associated transition system, which is model checked in parallel against the given property. First experiments with checking a mutual exclusion property of a resource locker exhibit promising results. We furthermore outline the design of a completely new tool which avoids the bottleneck of computing the transition system sequentially, which is done in the prototype version.

Other tool–assisted approaches to the formal verification of Erlang programs are the Erlang model checker by Huch [10], which uses predefined abstraction mappings, and the Erlang Verification Tool by SICS [15], which employs theorem–proving methods. However, neither of these provides a distributed implementation.

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 [1], 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 [16].

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.

The functional sublanguage of ERLANG is rather standard: there are value constructors for atomic constants (called *atoms*), integers, lists, and tuples. The expression e_1, e_2 denotes sequential composition; and $e(e_1, \ldots, e_n)$ represents a function call. **case** e **of** p_1 **->** $e_1; \ldots; p_n$ **->** e_n **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**!**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

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

The code fragment in Figure 1 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. It is a very simplified version of the algorithm presented in [2], which in addition is able to handle several resources from a given, finite set.

```
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 ->
      ok
  end,
  Locker!{release,self()},
  client(Locker).
```

Figure 1: Simple resource locker

The desirable correctness properties of such a system are straightforward: the locker should always be enabled to receive a new request or release (*no deadlock*), no two clients should gain access to the same resource at the same time (*mutual exclusion*), and all clients should eventually get their demanded access (*no starvation*). In the next section we will see how to establish the second property.

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 [9].

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[1]:

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

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 the 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 $p \neq p'$.

3 Model Checking

As explained in the previous section, a given ERLANG program is semantically evaluated by computing the associated labeled transition system. Its desired properties can then be specified using an appropriate temporal logic. Here we consider a variant of Kozen's *μ–calculus* [11], a powerful logic which allows to describe various safety, liveness, and fairness properties. It semantically subsumes the temporal logics CTL, CTL*, and LTL.

[1] Actually, [9] introduces a two–level semantics which distinguishes between expression–level and process–level steps. However this difference is of no importance here.

It employs the usual constructs from first–order logic such as the Boolean constants and connectives. To reason about the dynamic behavior of programs it supports the usual modalities `<`α`>`ϕ and `[`α`]`ϕ. They express that the current state of the system has an α–labeled transition to a state in which the property ϕ holds, and that every α–successor satisfies ϕ, respectively. Moreover it includes least and greatest fixed point formulae of the shape **`lfp`** $X.\phi$ and **`gfp`** $X.\phi$, which, roughly speaking, express eventuality (liveness) and invariant (safety) properties, respectively. Nesting of fixed points allows complicated reactivity and fairness properties to be expressed. Here, however, we consider only the *alternation–free* fragment of the μ–calculus which disallows the nesting of fixed–point formulae of different kinds, and which is sufficient for most practical applications.

For example, the (alternation–free) μ–calculus specification in Figure 2 can be used to check the mutual–exclusion property of the locker system given in Figure 1. First it defines the well–known CTL operator `AG(Phi)`, which asserts the safety condition that a given formula `Phi` holds in every (reachable) state of the system. Mutual exclusion can then be expressed as follows. After entering the critical section (which is indicated by the action `enter_cs`), no other process is allowed to enter it before it has been released (`exit_cs`). Extending this specification to systems with several resources is straightforward (cf. [2]).

```
macro AG(Phi) =
  gfp X.((Phi) and [tau]X)
end

macro ONCE(Act1, Act2) =
  [Act1](gfp X.(([Act1]false) and ([not Act2]X)))
end

macro MUTEX =
  AG(ONCE("enter_cs", "exit_cs"))
end
```

Figure 2: Specification of the mutual exclusion property

The design of efficient *model–checking algorithms*, i.e., decision procedures for testing whether a given system satisfies a given formula, is an important topic in the area of computer–aided verification methods. Particular interest is attracted by distributed approaches employing networks of workstations (NOWs), not only for reasons of reduced runtime requirements but mainly for fighting the *state space explosion problem* (see [5] for an overview).

In [5] a distributed model–checking algorithm for the alternation–free fragment of the μ-calculus is introduced (which is an improved version of the one presented in [6]). To test whether the initial state satisfies the desired property, it interleaves the computation of the transition system with a game–based model–checking procedure [17]. Both activities are carried out in parallel on a NOW, using a hashing function to map states to processors.

The version described in [6] was implemented in HASKELL as a component of the TRUTH verification platform, which supports the specification and verification of concurrent systems in the CCS process algebra [12, 13]. The distributed model checker exhibits promising features: the distribution of the transition system is homogeneous and the runtime scales very well with regard to the size of the NOW.

In [5] a standalone C++ version of the algorithm was developed in order to integrate the ERLANG–related activities described in [2]. Rather than computing the transition system of a given CCS process, it accepts the output format of the μCRL tools and, as before, tests whether the initial state has the desired property. This distributed model–checking procedure constitutes the kernel of the ERLANG verification system which is described in the following section.

4 A Prototype Implementation

In this section we briefly describe the initial design of a distributed ERLANG model checker. It is based on the work presented in [2] where, after some normalizing transformations on the source–code level, the given ERLANG program is automatically translated into μCRL, a process algebra with data [4]. Then the corresponding tools are used to generate an intermediate representation of the transition system which is accepted by our distributed C++ model checker.

First experiments with checking the mutual exclusion property of the resource locker exhibit promising results. More concretely, we considered an extended version of the system in Section 2, comprising three client processes which try to access four different resources. The corresponding transition system contains approximately one million states. In this setting we were able to establish the mutual–exclusion property (see Section 3) within nine minutes on a NOW consisting of 52 processors with a total main memory of 13 GB, connected by a 100 MBit FastEthernet network.

However, a closer look at the runtime figures (see Figure 3) reveals that the algorithm does not scale very well. At least five workstations are required to provide enough swapping space. At first, adding more processors yields considerable speedups since swapping is replaced little by little by in–memory operations. However as soon as the transition system fits into the accumulated main memory of the workstations adding more processors does not provide any significant improvement. The reason is that in contrast to the TRUTH version where the transition system is computed on–the–fly from a given CCS process, the system is already precomputed and stored in a file. Thus the single processors are only employed with model checking, which involves many costly communication operations.

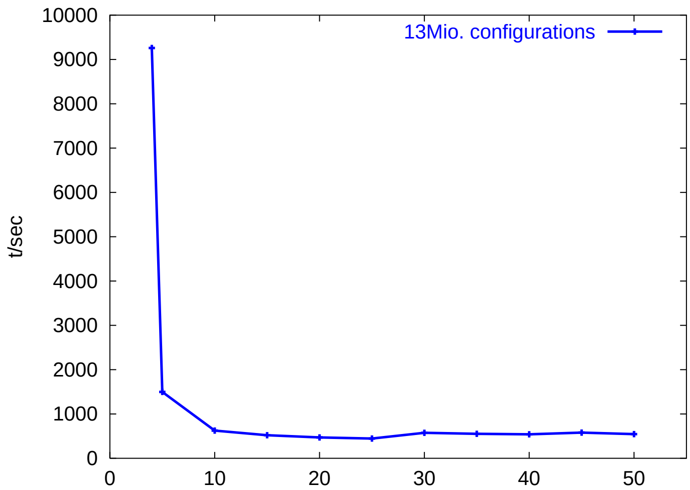

Figure 3: Runtimes

It seems to be sensible therefore to interleave the computation of the transition system, carried out in parallel, with the actual model checking, as it is done in the TRUTH/CCS version. (Note that using the sequential μCRL implementation the former task takes three hours, while the system can be checked within only nine minutes.) In the following section we outline the design of a corresponding, new tool.

5 The Tool

The promising results of the prototype implementation encouraged us to start working on a completely parallelized implementation of the ERLANG model checker in C++. However we do not intend to reinvent the wheel by building a parser and a semantic evaluator for ERLANG from scratch. We will rather employ the ELAN system, an environment for specifying and prototyping deduction systems in a language based on rules controlled by strategies [7]. This requires to specify the syntax and semantics of ERLANG in terms of abstract data types and conditional term rewriting rules modulo equational theories along the lines of [14].

This approach promises several advantages:

- The ELAN compiler translates the given specification into C code, thus supporting the interoperation with our C++ model checker.
- The ELAN compiler is highly optimizing, promising a good runtime efficiency of the generated code. In particular, rewriting modulo associativity and commutativity is well supported, which turns out to be useful when modeling communication between ERLANG processes.
- The expressive features of term rewriting modulo equational theories can be exploited to reduce the state space of the transition system. For example, purely functional evaluations (such as function calls) not involving side effects can be presented by (directed) equations. In this way they can be hidden within the current state. Furthermore equations can be used to implement abstraction mappings on the states, again reducing the state space (in some cases even from infinite to finite size).
- Moreover the rapid development of higher–level specifications for distinguished architecture components will be supported, such as generic servers as defined in the OTP. Here it is possible to specify a set of rewriting rules which characterize the abstract behavior 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 evaluates the request by executing the **`handle_call`**

function.

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

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. A complete formal model is presented in [3].

References

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

[2] T. Arts and C. B. Earle. Development of verified Erlang programs. In *Proc. of 6th International Workshop Formal Methods in Industrial Critical Systems (FMICS 2001)*, 2001. Available at `http://www.ericsson.com/cslab/~thomas/papers/fmics2001.ps`.

[3] T. Arts and T. Noll. Verifying generic Erlang client–server implementations. In *Selected Papers of the 12th International Workshop on the Implementation of Functional Languages (IFL'00)*, volume 2011 of *Lecture Notes in Computer Science*, pages 37–52. Springer, 2001.

[4] S. Blom, W. Fokkink, J. F. Groote, I. van Langevelde, B. Lisser, and J. van de Pol. μCRL: A toolset for analysing algebraic specifications. In G. Berry, H. Comon, and A. Finkel, editors, *Proc. 13th Conference on Computer Aided Verification (CAV'01)*, volume 2102 of *Lecture Notes in Computer Science*, pages 250–254. Springer–Verlag, 2001.

[5] B. Bollig, M. Leucker, and M. Weber. Local parallel model checking for the alternation free μ–calculus. Technical Report AIB–04–2001, Aachen University of Technology, 2001. Available at `http://aib.informatik.rwth-aachen.de/2001/2001-04.ps.gz`.

[6] B. Bollig, M. Leucker, and M. Weber. Parallel model checking for the alternation free μ-calculus. In T. Margaria and W. Yi, editors, *Proc. 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 543–558. Springer–Verlag, 2001.

[7] P. Borovanský, C. Kirchner, H. Kirchner, P.-E. Moreau, and C. Ringeissen. An overview of ELAN. In C. Kirchner and H. Kirchner, editors, *Proc. International Workshop on Rewriting Logic and its Applications*, volume 15 of *Electronic Notes in Theoretical Computer Science*. Elsevier Science, 1998.

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

[9] M. Dam, L. Fredlund, and D. Gurov. Toward parametric verification of open distributed systems. In *Compositionality: the Significant Difference*, volume 1536 of *Lecture Notes in Computer Science*, pages 150–185. Springer–Verlag, 1998.

[10] F. Huch. Verification of Erlang programs using abstract interpretation and model checking. *ACM SIGPLAN Notices*, 34(9):261–272, 1999. Proceedings of the ACM SIGPLAN International Conference on Functional Programming (ICFP '99).

[11] D. Kozen. Results on the propositional μ–calculus. *Theoretical Computer Science*, 27:333–354, 1983.

[12] 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 New York, 1999.

[13] M. Leucker and T. Noll. Truth/SLC – a parallel verification platform for concurrent systems. In G. Berry, H. Comon, and A. Finkel, editors, *Proc. 13th Conference on Computer Aided Verification (CAV'01)*, volume 2102 of *Lecture Notes in Computer Science*, pages 255–259. Springer–Verlag, July 2001.

[14] T. Noll. A Rewriting Logic implementation of Erlang. In M. van den Brand and D. Parigot, editors, *Proc. of First Workshop on Language Descriptions, Tools and Applications (ETAPS/LDTA'01)*, volume 44 of *Electronic Notes in Theoretical Computer Science*. Elsevier Science Publishers, 2001.

[15] T. Noll, L. å. Fredlund, and D. Gurov. The Erlang Verification Tool. In *Proc. 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.

[16] Open Telecom Platform (OTP). Ericsson Utvecklings AB, `http://www.erlang.org/documentation/doc/index.html`.

[17] P. Stevens and C. Stirling. Practical model checking using games. In B. Steffen, editor, *Proc. of 4th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'98)*, volume 1384 of *Lecture Notes in Computer Science*, pages 85–101. Springer–Verlag, 1998.