McErlang: A Model Checker for a Distributed Functional Programming Language

Lars-Åke Fredlund *
Facultad de Informática, Universidad Politécnica de Madrid, Spain
fred@babel.ls.fi.upm.es

Hans Svensson
Computer Science and Engineering, Chalmers University of Technology, Sweden
hanssv@cs.chalmers.se

Abstract

We present a model checker for verifying distributed programs written in the Erlang programming language. Providing a model checker for Erlang is especially rewarding since the language is by now being seen as a very capable platform for developing industrial strength distributed applications with excellent failure tolerance characteristics. In contrast to most other Erlang verification attempts, we provide support for a very substantial part of the language. The model checker has full Erlang data type support, support for general process communication, node semantics (inter-process behave subtly different from intra-process communication), fault detection and fault tolerance through process linking, and can verify programs written using the OTP Erlang component library (used by most modern Erlang programs).

As the model checking tool is itself implemented in Erlang we benefit from the advantages that a (dynamically typed) functional programming language offers: easy prototyping and experimentation with new verification algorithms, rich executable models that use complex data structures directly programmed in Erlang, the ability to treat executable models interchangeably as programs (to be executed directly by the Erlang interpreter) and data, and not least the possibility to cleanly structure and to cleanly combine various verification sub-tasks. In the paper we discuss the design of the tool and provide early indications on its performance.

Categories and Subject Descriptors D.2.4 [*Software Engineering*]: Software/Program Verification—Model checking

General Terms Verification

1. Introduction

To model check a modern distributed functional programming language is by no means a small task and there are many design decisions that have to be taken. One of the largest decisions is to choose between: (1) translating the program into some existing formalism and use (or possibly extend) existing model checking tools for this formalism, or (2) implement the verification algorithms directly for the language to be model checked. (In fact there is also a third alternative, namely to translate into a formalism which has been constructed for this particular task and implement tools for this formalism. This approach is briefly discussed in section 7.)

There are advantages (as well as weaknesses) with both approaches; Existing tools are probably optimized, and thus efficient to use. Translating into an existing formalism means that everything in the language has to be modeled, including data. Finding a suitable formalism might not be easy. Implementing model checking algorithms efficiently is hard and time consuming. Having the model checker in the language itself means that the pure functional part can be handled in a simple and efficient way.

One existing model checking tool for Erlang is the `etomcrl` tool set (Arts et al. 2004b), which consists of a translator from Erlang to μCRL, a state space generator for μCRL specifications, and the CADP state space analysis tools. Thus it is an example of the first alternative above, however early in the development of `etomcrl` its principal authors (Thomas Arts and Clara Benac Earle) were thinking of an implementation in Erlang itself. In the end the Erlang route was rejected because it was thought that it would be more efficient to reuse existing tools.

In this paper we describe the development and implementation of the McErlang model checker which follows the second implementation alternative above. The development of McErlang was started for several reasons. One reason was the curiosity to find out just how well an implementation in Erlang would work in practice. The main reason however was the wish to model check distributed as well as fault tolerant Erlang programs. (Both distribution and fault tolerance are missing in the `etomcrl` tool set). It was deemed too hard to extend the `etomcrl` tool set with the concepts of distribution and fault tolerance. The importance of supporting the distributed parts of Erlang is illustrated by Claessen and Svensson (2005). In their paper they show that it easy to overlook errors due to the loose synchronisation between processes in the distributed setting. They also demonstrate the presence of such errors in an open source Erlang implementation.

One significant advantage that the implementation in Erlang itself brings is that we can model check a larger fragment of the language than is normally achievable. There is for instance no separate step to compile the data sub-language of the source specification language to the often restrictive data language available in a model checker. Instead supporting the (purely functional) data part of Erlang is completely trivial; we can simply reuse the existing Erlang run-time system unchanged. It is in our opinion crucial to support a large fragment of Erlang in order to achieve some measure of acceptance of our tool by Erlang programmers.

The Erlang language contains many features not found in most normal programming languages (unless add-on libraries are used):

* The author was supported by a Ramón y Cajal grant from the Spanish Ministerio de Educación y Ciencia, and the DESAFIOS (TIN2006-15660-C02-02) and PROMESAS (S-0505/TIC/0407) projects

ICFP'07, October 1–3, 2007, Freiburg, Germany.

dynamic types (i.e., no static type system), concurrency via a process concept, inter-process communication using only asynchronous message passing, distribution by mapping processes onto (remote) processing nodes, fault tolerance via a failure detector mechanism, and a standardized set of high-level components built on top of this foundation. As Erlang programmers frequently make use of all these features we think it is vital that the verification tool supports them too.

Nevertheless, by choosing to implement a model checker in a functional programming language we risk paying a price with regards to loss of execution performance and increased storage requirements; there is clearly a trade-off between easy experimentation and expressive power on one hand, and implementation efficiency on the other. With the McErlang tool we want to explore this trade-off, and we hope that by not having to simulate the functional parts of Erlang this model checking approach is rather efficient.

Since we had access to a prototype implementation of the distributed Erlang semantics in Haskell, we also did some experiments with implementing a model checker for Erlang in Haskell. We did not implement a full model checker, but the experiments gave us some insight in the strengths and weaknesses with such an approach. In section 7 we briefly discuss these results to get a different perspective on the McErlang implementation.

One of the design goals with McErlang was that it should be easy to use. All that is needed to use McErlang is a program to test, a specification of the environmental constraints and a property to check. All three are written entirely in Erlang. The environmental constraints describe how the program is executed in the implementation of the run-time system provided by McErlang. The property is given in the form of a monitor/automaton that is executed in parallel with the program, checking for errors along the execution path. This work flow is discussed further in section 4 and section 5 and is illustrated in Fig. 7.

Contributions The main contribution of the paper is a presentation of the tool McErlang. The paper is not so much about the theory behind model checking or the semantics of Erlang, instead we focus on design choices, implementation decisions, adaptability and usability. McErlang is a model checker for Erlang implemented in Erlang, it supports a large subset of the Erlang programming language. In particular it supports all of the distributed and fault-tolerant parts of Erlang. This is especially important since distributed and fault-tolerant implementations are known to be error prone and hard to test and debug. McErlang is also easy to use and should be accessible to an ordinary Erlang programmer. Finally McErlang is designed in a very modular way and can easily be adapted to support other target languages.

Paper organization The next section contains an introduction to the most important features of the Erlang programming language and section 3 contains a description of the most prominent features of the Erlang semantics. In section 4 the parametric design of McErlang is described and section 5 presents the model checker itself, i.e., essentially an on-the-fly model checker which executes Büchi automatons (coded in Erlang) in parallel with the Erlang program under study. In section 6 we show some results and examples of using the model checker. Section 7 discuss a number of design choices in more detail, and section 8 summarises related work. Finally section 9 draws conclusions, and outline future research work.

Download McErlang McErlang can be downloaded at `http://babel.ls.fi.upm.es/~fred/McErlang/`.

2. The Erlang Programming Language

Erlang is a programming language developed at Ericsson for implementing telecommunication systems (Armstrong et al. 1996; Armstrong 2007). It provides a functional sub-language, enriched with constructs for dealing with side effects such as process creation and inter–process communication via message passing. Moreover Erlang has support for writing distributed programs; *processes* can be distributed over physically separated processing *nodes*.

Today several commercially available products developed by Ericsson and other companies are at least partly programmed in Erlang, an example is the AXD 301 ATM switch (Blau and Rooth 1998). The software of such products is typically organized into many, relatively small, source modules, which at run-time execute as a dynamically varying number of processes operating in parallel and communicating through asynchronous message passing. The highly concurrent and dynamic nature of such software makes it particularly hard to debug and test.

Erlang programmers, of course, mostly work with ready-made higher-level language components rather than the basic language. In practice programmers predominantly use the OTP component library (Torstendahl 1997), which offers a number of useful software components such as: a generic server component for client-server communication, a finite-state machine component, and a supervisor component that restarts failed processes. Our approach to model checking Erlang programs can verify software that is built using both the core message passing language and with these high level components.

A key feature of the systems for which Erlang was primarily created is fault-tolerance. Erlang implements fault-tolerance in a simple way. Links between two processes A and B can be set up so that process B is eventually notified of the termination of process A and vice versa (using the normal message-passing machinery). The default behavior of a process that is informed of the abnormal termination of a linked process is to terminate abnormally itself. Alternatively the linked process can specify that it wishes to receive a message with a notification that its linked process has terminated. This process linking feature can be used to build hierarchical process structures where some processes are supervising other processes, and can take corrective action (e.g., restarting them) if they terminate abnormally. In order to create such fault-handling structures, Erlang/OTP provides the supervisor behavior.

Another key feature of Erlang systems, which is particularly useful for 24/7 systems, is the mechanism for hot code replacement. In short it is possible to phase out old code and replace it with new code, having both old and new code running simultaneously. This feature enables bugs to be corrected and features to be added without stopping the system.

In summary, the Erlang/OTP programming environment is a comparatively rich programming environment for programming systems composed of (possibly) distributed processes that communicate by message passing. Fault tolerance is implemented by means of failure detectors (the linking mechanism), a standard mechanism in the distributed algorithms community. Moreover there is a process fairness notion, something which often makes it unnecessary to explicitly specify fairness in correctness properties. Moreover the language provides explicit control of distribution, and a clean model of distribution semantics. For distributed processes (processes executing on separate nodes) the communication guarantees are far weaker than for processes co-existing on the same processor node. This gives, in a clean way, considerable power with regards to checking a program under different environmental constraints (simply changing the mapping of processes to nodes), but on the other hand there is a requirement on implementing the runtime system with different guarantees for inter-node and intra-node communication.

Multi-core programming The concurrency oriented nature (Armstrong 2003) and the (almost) transparent distribution makes Erlang a really good candidate for writing efficient distributed software.

With the latest version of the Erlang Run-time System (Erlang 5.5/OTP R11B) this is taken even further, as it includes built-in support for SMP (Symmetric Multi Processing). SMP is today supported by most modern operating systems and becomes more and more important with the introduction of dual/quad/... processors, multi-core systems and hyper-threading technology. The SMP support in Erlang is transparent since most problems occurring in multi-threaded programs are solved by the Erlang VM. The SMP version of the VM can have many process schedulers running inside each OS thread, the default is to have as many schedulers as there are processors (or processor cores) in the system. Since the SMP support is completely transparent we get 'for free' an efficient multi-core implementation if we have a correct distributed implementation. This shows another benefit of having a working model checker for distributed Erlang.

3. Semantics

Erlang is at the same time both a simple language, having at its core a fairly uncomplicated dynamically typed functional language with eager evaluation, and a fairly complicated one. The complexity is due to the addition of language layers proving support for concurrency (processes and message passing), and distribution (processing nodes that encapsulate processes) and fairly elaborate interprocess fault detection and fault handling mechanisms (via process links and process monitors).

The intuitive picture of the distributed semantics is rather simple, the guarantees given are simply: "*communication between a pair of processes is assumed to be ordered*" as described by Armstrong (2003). The semantics of links and monitors are also fairly easy to get an intuitive understanding of. However, the full semantics for distributed Erlang is indeed complex. It consists of some rather long and technical transition rules. Especially the corner cases, such as using the link mechanism on a dead process, makes a presentation somewhat lengthy and less intuitive than one could wish. Nevertheless, our formal description of the semantics is layered in three layers in a very clear way.

- **Functional Semantics -** consists of the pure functional part of Erlang (function evaluation, pattern matching, etc). It is dynamically typed and fairly straight forward.
- **Process Semantics -** is above the functional semantics, and consists of process evaluation rules (sending and receiving messages and links, starting/terminating processes, and silent computation steps) as well as process communication rules (process interleaving and process communication). This is all for the single node case, that is all the involved processes are executing in the same run-time system.
- **Node Semantics -** is placed on top of the process semantics, and adds the concepts of nodes and full distribution to the semantics. Similarly to the process semantics it consists of node evaluation rules and node communication rules.

The functional semantics and the process semantics are described in detail in Fredlund (2001) and the node semantics is introduced in Claessen and Svensson (2005). The layering described here is, as we see later, clearly mirrored in the implementation of the model checker. Since it is not feasible to cover all aspects of the semantics in this paper, we just highlight a few important details. With the following example we show the importance of having the node semantics layer and that our intuitive understanding of the semantics is not sufficient in all cases.

3.1 World Hello?

Consider the small Erlang program in Fig. 1. When we run the function `worldhello()` it will spawn A, which in turn results in two processes being spawned (B and C). Thereafter A will first send the message `hello` directly to C and then send the message `world` to B. Process B is very simple, once it receives a message, it will forward it to process C. Process C just receives two messages, and prints the result. (`?MODULE` is a built-in macro which is replaced by the name of the current module by the compiler, $\texttt{?NODE}_i$ are ordinary macros defined elsewhere.)

```
worldhello() ->
    spawn(?NODE1,?MODULE,procA,[]).

procA() ->
    PidC = spawn(?NODE3,?MODULE,procC,[]),
    PidB = spawn(?NODE2,?MODULE,procB,[PidC]),
    PidC ! hello,
    PidB ! world.

procB(PidC) ->
    receive world -> PidC ! world end.

procC() ->
    receive X -> ok end,
    receive Y -> ok end,
    io:format("~p ~p\n",[X,Y]).
```

Figure 1. 'World Hello'-program

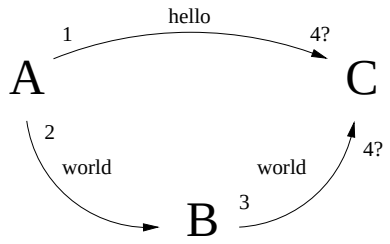

Figure 2. Possible message sequences

The interesting aspect of this program is that the result of running the program depends on the distributed environment! If the program is running on a single node (that is `?NODE1 = ?NODE2 = ?NODE3`), the result is always: `hello world`. However if the program is running in a distributed environment (that is $\texttt{?NODE1} \neq \texttt{?NODE2} \neq \texttt{?NODE3}$), the result could be either `hello world` or `world hello`. The reason for this is that there are different communication guarantees at the distributed level. In short; in a single run-time system message delivery is instantaneous (that is the message is immediately put in the receivers in-box), while in a distributed system the only guarantee is that messages between a pair of processes are ordered. The possible message sequences in the distributed case is shown in Fig. 2.

The `etomcrl` tool for example (and the same goes for many other Erlang verification efforts) does not have a notion of nodes at all, and therefore this aspect cannot be checked. It is clear that this is a problem, since the difference in communication guarantees is a definite source of errors in Erlang systems (see for example Arts et al. (2005)). It was therefore a strong requirement on McErlang that it should handle the node semantics. In fact, it is fair to say that the major part of the implementation effort of the model checker has been devoted to an accurate treatment of the often surprisingly complex semantics of the node semantics part of the run-time system.

3.2 Semantics implemented in McErlang

The McErlang tool has a full implementation of the distribution part of Erlang (i.e., explicit programmatic mapping of processes

to explicit nodes), and thus provides the possibility to verify code based on either the assumption that all process are local (on the same node), or remote (all processes reside on different nodes), or a mix of the disciplines. Thus it is possible to verify a program under quite weak communication guarantees and be sure that later processes can be freely mapped on distributed nodes. However, the drawback of the distributed semantics is that it greatly increases the state space of the verified programs; essentially the distributed semantics non-deterministically delays the delivery of messages to a receiving process.

4. Structure of the Implementation

The model checker implementation is parametric, using the Erlang/OTP style of behaviors to specify particular component behaviors that provide services to the model checking algorithm.

The basic task of the model checker is of course to check a program against a correctness property, a *monitor module*, that implements the correctness property to check.

Except specifying which program to check (a specific Erlang function), and which Erlang module that implements the correctness property, a user of the tool can also choose:

- the name of a *language* module providing an operational semantics,
- the particular *verification algorithm* to use, (e.g., a safety property checker, a liveness property checker or just testing – i.e., simulation of the program in conjunction with a correctness property),
- the name of a *state table* implementation, that records encountered program states (typically a hash table), and
- the name of an *abstraction module* that abstracts program states,

The modular composition of McErlang is illustrated in Fig. 3, and in the following sections we describe the functionality of these modules in turn.

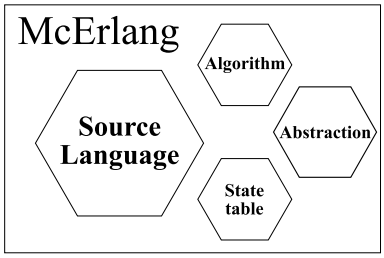

Figure 3. McErlang modular structure

4.1 Source Language

The language module should provide two functions implementing an operational semantics for the language: (i) `transitions` which given a state returns a list of all next actions executable by the program, and (ii) the function `commit` which given an action returns a concrete program state. The `transitions` function may not cause side effects outside the model checker environment (e.g., really writing out a file to the file system) whereas `commit` may (if used by the simulation algorithm). The language module most commonly used is clearly the one providing an operational semantics for Erlang, however, we have also implemented an operational semantics for the WS-CDL web choreography language (W3C 2005). Although the effort is less mature than the Erlang model checker, it is interesting that the basic framework of the model checker can be reused in a different language setting (Fredlund 2006). As XML and XPath constitutes integral parts of the WS-CDL definition, having good support libraries available for these languages is very useful when representing their operational semantics. As Erlang has seen considerable industrial usage, the language already had good library support for working with XML based documents; we expect the same kind of advantages from using Erlang when providing model checkers for other target languages.

4.2 Correctness Properties

Correctness properties are encoded as automata programmed in Erlang. A safety monitor is a function which is checked in every reachable program state, and which returns an error if an invalid state is seen. A *Büchi monitor* (automaton) is a monitor that additionally may mark certain states as accepting. A program violates a Büchi monitor if a cycle can be found in the combined state space of the program and the monitor, which contains an accepting state. As is well known (Vardi and Wolper 1986), linear temporal logic formulas can be automatically translated to Büchi automata.

The memory aspect of monitors is implemented by sending along the old monitor state as an argument to the Erlang function implementing the monitor. Concretely a monitor defines two callback functions: `init(`*parameters*`)` and `stateChange(`*programState*`,`*monitorState*`)`. The `init` function returns {`ok`,*monState*} where *monState* is the initial state of the monitor.

The `stateChange` function is called when the model checker encounters a new program state *programState* and the current monitor state is *monitorState*. If a safety monitor finds that the combination of program and current monitor state is acceptable, it should return a tuple {`ok`, *newMonState*} containing the new monitor state. If future states along this branch are uninteresting the monitor can return `skip` (e.g., to implement a search path depth limit), any other value signals a violation of the correctness property implemented by the monitor. A Büchi automatons should return a set of states, each state either accepting {`accepting`,*state*} or not {`nonaccepting`,*state*}. Normally we expect a "sound" `stateChange` function to be without side effects.

As an example, the code fragment in Fig. 4 implements a simple safety monitor that guards against program deadlocks: (a process is considered deadlocked if its execution state as recorded by the process data structure in the run-time system is `blocked`).

```
stateChange(State,MonState) ->
  case lists:any
       (fun (P) -> P#process.status =/= blocked end,
        State#state.processes) of
    true ->  {ok, MonState};
    false -> {deadlock, MonState}
  end.
```

Figure 4. Simple safety monitor

The syntax *variable*`#`*recordName*`.`*field* is used to access the field *field* of the record variable *variable*, of type *recordName*.

4.3 Algorithms

The McErlang tool currently offers two basic on–the–fly depth-first state traversal model checking algorithms, one to check safety properties and the other to check Büchi automatons (the liveness checking algorithm adapted from Holzmann et al. (1996)). To give an intuition to the coding of these algorithms in Erlang, a schematic representation of the algorithm for safety property checking is depicted in Fig. 5 (we have abstracted out the parameter passing of modules implementing language (`Lang`), monitors (`Mon`), abstraction (`Abs`) and table implementation (`Tab`)).

```
check([]) -> ok;
check([[]|Earlier]) -> check(Earlier);

check([[State|Alts]|Earlier]) ->
  {ProgState,MonState,StateTab,AState} = State,

  % Check monitor
  {ok,NewMonState} =
    apply(Mon,stateChange,[ProgState,MonState]),

  % Abstract state
  {ok, {AbsState,NewAState}} =
    apply(Abs,abstractState,
          [{ProgState,NewMonState},AState]),

  % Check whether state already seen
  case apply(Tab,addState,[AbsState,StateTab]) of
    no ->
      check([Alts|Earlier]);

    {ok, NewStateTab} ->
      NewStates =
        [{S,NewMonState,NewStateTab,NewAState} ||
         S <-
          lists:map
          (fun (Action) -> apply(Lang,commit,Action),
           apply(Lang,transitions,[ProgState]))],
      check([NewStates,Alts|Earlier]])
  end
end.
```

Figure 5. Safety property checking algorithm

To check an Erlang function call $m{:}f(p1,\ldots,pn)$, given an initial monitor state $monState$ and an empty state table t, and abstraction state a, the checking algorithm should be invoked with:

`check([[{mkProc(`$m,f,$`[`$p1,\ldots,pn$`]),`$monState,t,a$`}]])`

where `mkProc` constructs a model checking process executing the function call argument.

As seen in the listing, model checking states are composed of a program state, a monitor state, a state table, and an abstraction state. Program states are checked against the monitor, and if accepted, are abstracted using an abstraction function provided by the module `Abs`. The abstracted states are checked against membership in the state table. If the program state is new, the set of next states is computed using the function `transitions`. Note that the particular choice of abstraction and table storage is abstracted out from the algorithm itself.

In addition there is a simple *simulator* available, which by default chooses the next program state randomly, but in addition has some debugging functionality, e.g., next states can be explicitly chosen, transitions can be single or multiple stepped, breakpoints can be set, and backtracking to previous states is supported. The simulator is also used to explore safety model checking counterexamples (traces).

Fairness Constraints on Executions The Erlang language standard requires that process schedulers must be fair. The McErlang tool accordingly implements (weak) process fairness directly in its (liveness) model checking algorithm by omitting non-fair loops (i.e., ones that constantly bypass some enabled process) from the accepting runs.

4.4 Tables

A state table records pairs of program and monitor states encountered during model checking, to detect recurring states. The state table implementations used are normally imperative (e.g., updates to them are destructive) for performance reasons; however purely functional implementations of the tables are available.

```
-module(hashAbs).
-export([init/1, abstractState/2]).

init(Size) ->
  {ok,Size}.

abstractState(State,Size) ->
  {ok,{erlang:phash2(State,Size),Size}}.
```

Figure 6. Abstraction module for hashing

4.5 Abstractions

An abstraction abstracts a concrete program state into an abstract representation. It can be used to drastically reduce the checked state space of a program. The idea is inspired by the use of abstractions in Arts and Fredlund (2002). A typical abstraction used in model checking is to compute a hash value from the state, and to use the hash value as the abstract state when checking for membership in the state table. However, program specific abstraction functions can also be implemented. For example, an abstraction could transform an integer variable into a boolean value, signaling whether the integer is less than zero. Clearly, there is in general no guarantee that such an abstraction is safe, i.e., that it does not cause a program failure to escape undetected (false positive).

As a second example we have implemented the usual abstraction of collapsing a whole state to a single integer (through hashing), and using a bit array table module to implement the state table. Thus, in a modular fashion, we have obtained an implementation of Holzmann's bit-state hashing verification algorithm (Holzmann 1991). An implementation of a hashing abstraction thus becomes as simple as Fig. 6, where `erlang:phash2` is a built-in function which computes a hash value between 0..`Size` for its term argument. Note that is an unsafe abstraction, although as proven in practise in many verifications, also a highly useful one.

5. Executing Erlang Programs in McErlang

The model checking capability for Erlang programs is provided by executing Erlang programs directly in the existing Erlang run time system. This enables an easy and reasonably efficient handling of computations that act solely on data (the purely functional sub-part of Erlang). However, the existing Erlang run time system does not provide a method to capture the combined system state of a running program (check-pointing). This is unavoidable, since in general an Erlang computation could be distributed and so the combined state cannot be efficiently, or even reliably, collected.

For this reason we have implemented in Erlang a new run-time system for the concurrent and distributed part of the language, that implements easy access to the combined system state of an Erlang program. This run-time system simply simulates distribution and concurrency, all computations take place inside a single real Erlang process. Structurally the new run-time system is layered on top of the old one, replacing only the process handling and the concurrency part of the old system. This layered structure also in many ways resemble the layered structure of the Erlang semantics in section 3.

Essentially a complete verification model consists of three parts: (1) an Erlang program containing the original program to be checked, (2) a re-usable implementation of the run-time system (also written in Erlang) and (3) a specification of the environmental constraints (e.g., which process/node failures and link failures occur). See section 6.2 for a concrete example of such environ-

mental constraints. By separating the model cleanly into these three parts we can independently experiment with different assumptions/implementations. The workflow is illustrated in Fig. 7.

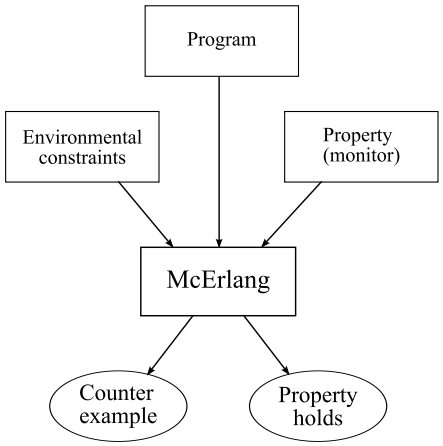

Figure 7. McErlang workflow

5.1 Run-time Organization

The state of the run-time system, e.g., recording process states, communication queues and so on (see section 5.3 for details), is stored in the imperative Erlang process dictionary; all (simulated) model checking processes run, interleaved, in a single Erlang process. All state updates and queries are thus implemented as accesses to the process dictionary. We have also experimented with a solution where the state of the run-time system is kept in a separate process, a solution more in the spirit of the Erlang design philosophy. Unfortunately, that solution severly impacts on the speed of model checking, slowing down a typical verification with a factor of three compared to the process dictionary solution. An obvious alternative would be to pass along the global state as a parameter everywhere in the verifier code. In e.g. Haskell cleaner as well as more efficient solutions are obviously possible.

5.2 Translation

A vital part of the model checker is a compiler that translates an Erlang program to be verified to a modified Erlang program that uses the new run-time system.

Actually we still use the old runtime system to execute even the translated functions (this is to avoid having to re-implement any part of the data handling in Erlang). However, calls to Erlang functions with side effects in the old runtime system have been replaced with calls to Erlang functions with side effects in the model checker instead.

The principal goal of the translation is to transform Erlang functions that use the `receive` construct[1] so that instead of executing that construct, which would immediately hang the execution of the model checker as there would be no value to be received, the modified function instead returns a special return value. The return value indicates the desire to receive a message, and a continuation function coding the normal execution of the function after the reception of the message.

The translation takes a set of modules as input and returns a set of translated ones. The resulting Erlang modules can be compiled by the normal Erlang compiler (which is a requirement for using the model checker).

In case the compiled application makes use of OTP components (generic server, supervisor, etc…) the McErlang tool will include in the compilation the source code of tailored versions of these libraries, written in Erlang of course.

Replacing API calls Apart from transforming code that uses the `receive` construct, the translation does a very simple transformation of other API calls such as e.g. sending a value to a process.

As the Erlang language lacks a good reflection capability, the new run-time system is provided as a new application library `evOS`. For example, an application that used to send a message $\{\texttt{request},22\}$[2] to a process with process identifier pid using the send construct $pid\,!\{\texttt{request},22\}$ should instead call the library function $\texttt{evOS:send}(pid,\{\texttt{request},22\})$. The functions that implement the new API calls are implemented in Erlang itself and operate directly on the global system state (nodes, ether, processes, links, and a register map as discussed in section 5.3 below).

Handling Reception of Messages The mapping of calls to Erlang API functions to the new run-time system works for all Erlang constructs except the `receive` statement which is used by a process to retrieve a value from its mailbox (or process queue), as the `receive` call suspends until a matching value is available.

Instances of `receive` statements in the Erlang code to be model checked are instead replaced with code that returns a tuple like: $\{\texttt{recv},\{module,fun,context\}$, where the contained inner tuple $\{module,fun,context\}$ identifies a function that implements the logic of the particular receive statement.

When an invoked Erlang function, in an Erlang process, returns such a `recv` tuple the new run-time system recognizes the special return value and marks the process as `blocked`, and then checks whether there is any receivable value in the process mailbox (in which case the process status is upgraded to `receivable`). In any case, the run-time system can schedule another enabled process.

The transformation of an Erlang program containing a `receive` statement into one returning a `recv` expression is explained by the small example in Fig. 8 and Fig.9.

```
server(State) ->
  receive
    {new_state, NewState, Pid} ->
       Pid!{reply,State},
       server(NewState)
  end.
```

Figure 8. Receive statement – before translation

The code fragment in Fig. 8 defines a function `server` which guards some private state. The state can be changed by sending a call message to the server process, containing a process identifier and a new state. The server replies with the old state. The translation of the server function is shown in Fig. 9. In the transformed code, a call to $\texttt{server}(state)$ will immediately return a tuple $\{\texttt{recv},\{\texttt{?MODULE},\texttt{f\_0},[state]\}\}$ which is a special form recognizes by the model checker.

In general the function referenced in `recv` should accept two parameters, a message in the queue to be tested whether it is receivable, and a list of variables needed in the evaluation of `receive`. If the message is receivable, the function should return a tuple with a new anonymous function; if not `false` should be returned. The anonymous function receives the same parameters as the original function, and contains the body of the `receive` clause. The separation of receive into two functions serves to separate the testing whether a message is receivable from the actual retrieval of the message from the queue (as the process could continue by performing some side effect).

[1] `receive` is a process construct to retrieve a value sent to the invoking process.

[2] A tuple containing a literal symbol `request` and the number 22. In Erlang variables begin with a capital letter and atoms (literals) with a lowercase letter.

```
server(State) -> {recv, {?MODULE, f_0, [State]}}.

f_0({new_state, NewState, Pid}, [State]) ->
  {true,
   fun ({new_state, NewState, Pid}, [State]) ->
      evOS:send(Pid,{reply,State}), server(NewState)
   end};
f_0(_, _) -> false.
```

Figure 9. Receive statement – after translation

Handling a non tail-recursive `receive` The translation of the `receive` construct sketched above is correct only when it occurs in a tail-recursive position. For the general case, what is essentially a run-time stack is used instead.

The run-time stack is implemented using another special return value: {`letexp`,{*expr*,{*module*,*f*,*parameters* }}}, which is used in the situation when a receive statement occurs in an expression context (i.e. not in a tail-recursive position). Consider for example the recursive function `server` in Fig. 10 which repeatedly calls a function `doRequest` which in turn contains a receive statement.

```
server(State) ->
  {ok, NewState} = doRequest(State),
  server(NewState).
```

Figure 10. Non tail-recursive receive – before translation

The example in Fig. 10 is translated into a `letexp` return value as seen in Fig. 11. The function referenced in the `letexp` special expression is called when the inner function has returned a value, and receives as arguments the returned value as first argument and as second argument a list of variables necessary in the continued computation. In general all non tail recursive calls to functions that contain a `receive` in their body will have to be similarly guarded using a `letexp`. We use a global analysis over the set of input modules to the translator for computing the transitive closure of which functions may execute a `receive` statement.

```
server(State) ->
  {letexp, {doRequest(State), {?MODULE, f_1, []}}}.

f_1({ok,NewState}, []) ->
  server(NewState).
```

Figure 11. Non tail-recursive receive – after translation

The translation is somewhat complicated by the need to support the Erlang "feature" of permitting variable bindings to migrate out of their scope. The Erlang example in Fig. 12, which compiles without warning and does not cause run-time errors, illustrates the translation difficulty (`Logger` is assumed to be bound to a process identifier). Note that the variables `Msg` and `NewV` are bound in different branches of the `receive` construct, but may still be used outside of it.

Non-determinism in Erlang Another special return value is {`choice`,[{*module*, *fun*,*context* },...]} which introduces explicit non-determinism in Erlang; the model checker will non-deterministically select the continuation function from the list of function alternatives. This construct is needed to use Erlang as a specification language rather as a programming one. As an example, suppose that we have implementing a drink machine in Erlang, offering either coffee or tea. Using the choice construct it is easy to model a machine user that non-deterministically selects either coffee or tea, and to verify that the program works correctly regardless

```
pingOrpong(Logger) ->
  receive
    {ping,V,Sender} ->
      Sender!{Msg=pong,NewV=V+1,self()};
    {pong,V,Sender} ->
      Sender!{Msg=ping,NewV=V+1,self()}
  end,
  Logger!{Msg,NewV},
  pingOrPong(Logger).
```

Figure 12. Migrating variable bindings

what drink the user chooses (the model checker automatically explores both possibilities).

Finally {`pause`,{*module*,*fun*,*context* }} is short hand for a choice with a single continuation function; it is used to facilitate detection of interesting states in correctness properties.

5.3 Data Structures in the Run-time System

An Erlang state in our run-time system is a hierarchical structure and mimics to a large extent the organization of the real run-time system (and the structure of the layered Erlang semantics!) for Erlang, except, of course the state is physically centralized.

The top level of the hierarchical structure is composed of a tuple

$$\langle nodes, ether \rangle,$$

combining a data structure containing the nodes of the running system and an *ether* data structure containing messages in transit between nodes. Each message is identified by the following tuple:

$$\langle receivingNode, sendingNode, messageContent \rangle.$$

The ether data structure essentially has a separate queue of messages, sorted by sending time, for each pair of sending and receiving nodes. This is needed since the language guarantees that communication between any two nodes is FIFO-like, i.e., messages are delivered in order, if they are delivered at all. The $messageContent$ contains the message itself (e.g., a normal message sent between two processes or a run-time event such as e.g. a notification of a process termination).

A node tuple

$$\langle name, processes, registered, monitors, node\_monitors, links \rangle,$$

is on the second hierarchical level. The $processes$ field contains the processes executing on the node, $registered$ implements the Erlang name server which maps (on a node basis) pids to symbolic names. The fields $monitors$, $node\_monitors$ and $links$ is used in the three different process linking mechanisms available in Erlang.

Each process is a tuple

$$\langle status, expr, pid, queue, dict, flags \rangle.$$

The field $status$ records the execution status of the process, e.g., whether it is blocked waiting on incoming messages, ready to run, or ready to receive an existing message. The $expr$ field describes the next piece of code to execute, concretely a named user-defined Erlang function and a set of actual parameters to invoke the function with. The pid field is the system-wide unique process identifier of the process, $queue$ contains the messages sent to the process that are available for reading (inter-node messages migrate from the $ether$ data structure to the $queue$ whereas intra-node messages are directly put in the $queue$ data structure, mimicking the different communication guarantees provided by the run-time system for inter-node compared to intra-node communication). Finally $dict$ contains a process dictionary (the equivalent of imperative variables in Erlang), and $flags$ describes the setting of various process options.

Although the exact manner in which states are physically stored or represented (e.g., on the stack of 'choice points' and in the table of states previously seen) during a model checking is fully configurable, the normal exact representation of a state ensures that states are *normalized*, i.e., nodes are sorted in some order, as are processes within an nodes, as are links (pairs of processes identifiers in a node) and so on, to ensure a rapid check for state equality.

5.4 Model Checker Semantics

The tool implements a major part of the core `erlang` module in the Erlang/OTP distribution omitting mainly functions to inspect the run-time system itself, to obtain process status, timing functions, and ports (which are used to interface with foreign, i.e. non–Erlang, code). In total we provide around 40 such API functions, the implementation of which constitutes a significant portion of the lines of code of the model checker.

The operational semantics implemented by McErlang comprise an interleaving transition relation between Erlang states whose actions are decorated by sequences of actions (i.e., a big-step operational semantics). States are comprised by stable systems (e.g., where all processes are waiting in receive statements or have just spawned) and transitions are caused by invoking a single enabled process to run which may cause many side effects until it again becomes stable (waiting in a receive statement).

The use of a big–step semantics means that some errors will go undetected which would be caught using a smaller-step semantics. For the typically large scale systems that we are interested in verifying with McErlang there is a trade-off here. One option is to have a very detailed execution model with all the possibility non-determinism inherent in the programming language.[3] This quickly leads to enormous state spaces with the result that only a very tiny part of such state spaces can be explored by a model checker. On the other hand, we can reduce the non-determinism in the specification language by slightly changing its semantics. The result is smaller state spaces, which we can verify a bigger part of, but there are possibly states that we can never check because they will never be generated by the model checker. In future work we aim to implement a more finely-grained semantics for intra-node Erlang to explore this issue in further detail.

Interestingly it turns out that we can recover a more finely-grained semantics in case each process communicates only with other remote processes (located on other nodes). Then a send, as well as any other side effect, will be arbitrarily delayed (since the node $ether$ data-structure is used, which essentially have separates queues for all pairs of communicating processes, see section 5.3 for details) compared to side effects caused by other processes, and so all interleavings of side effects are recovered.

5.5 Run-time Environment Modeling

Probably the most challenging part of developing a model checker for Erlang is to accurately model the environmental constraints put on a running Erlang program. For example: constraints on scheduling Erlang processes, the semantic impact of mapping processes onto remote processing nodes, the basic communication guarantees of Erlang, and on the frequency of failures in a running system.

Moreover the Erlang API has quite a few functions with side effects, whose actions cannot be understood as simply as sequences of lower-level primitives (send and receive) but are first-class citizens in any operational semantics.

As an example we consider below the implementation, which is a form of operational semantics, of the `erlang` API function `exit/2`. In Erlang, `exit(Pid,Reason)` is used to send a termination signal to the process referenced by `Pid`, which may be terminated as a result. The implementation has to handle the rather subtle interplay between fault-handling mechanisms (linking, monitors) and take into account process locality (on the same node, or not), etc. Moreover, its behaviour is very different depending on whether the process to terminate resides on the same node as the process executing the call or not.

Although the function may seem complicated, it is an intrinsic part of the Erlang language, which is used by programmers all the time (as invoked in through higher-level functions), and we have no choose but to model it faithfully if we wish to verify realistic Erlang software.

Implementation sketch:

1. First the arguments are checked; if `Pid` is not a process identifier an exception is raised.
2. The code then checks if `Pid` is a local pid (i.e., the corresponding process resides on the same node as the process which executes the `exit/2` call. If the process is remote, a signal (a message) is sent to the node on which the process resides containing a request to issue an `exit/2` call, and the function returns.
3. If it is a local process, the process flags are retrieved. The process traps exit messages if the flag `trap_exit` is set. If `trap_exit` is set, and the `Reason` argument is not `kill`, a message, `{'EXIT',self(),Reason}`, is put into its mailbox (where `self()` evaluates to the pid of the process that called `exit/2`), and the function returns.
4. If the process is local, and it is not trapping exits, and the `Reason` argument is `normal`, the process is not terminated (and no message is put in its message queue), and the function returns.
5. Otherwise (the process is local, the reason is `kill`, or...) the process is terminated, i.e., it is removed from the process table.
6. Moreover any registered names for the process are removed (by modifying the *registered* element in the *node*).
7. And any monitors the now terminated process has set up are removed (all nodes are searched for such monitors), and messages concerning terminated processes due to such monitorings are removed (from the *ether* element).
8. Then every process that has requested to monitor the terminated process (information present in the *monitor* field of the *node* structure) are sent a message informing them of the termination of the process they monitored, and the reason for termination.
9. Then all the links mentioning the terminated process are examined (recorded in the *links* field of the *node* structure). If a link mentions a remote process, then the remote process is sent a signal (message) informing it that one of its linked process has terminated. If the process is local, the linked process is itself a candidate to terminate immediately, and execution contains for the linked process with roughly step 3 above.

As is indicated in the last step, in Erlang the termination of a process can, through the link concept, cause the termination of more processes, and so on, in a chain reaction. Although at first

[3] As an extreme case, Erlang, for instance, does not fix the order of evaluation of arguments to functions, so a totally faithful semantics would generate all such orderings. As Erlang programmers can happily write code that cause side effects in the evaluation of function call arguments, generating all such orderings may be highly important in model checking. However, the number of extra states could be huge, although part of the overhead could be eliminated through use of intelligent reductions. In practise, however, the only available Erlang language implementation *does* fix the order of argument evaluation, and in our opinion this is very unlikely to ever change in the future of Erlang.

counter-intuitive, the idea is to use this behavior of the linking mechanism to write fault tolerant code. Essentially some processes are designated as supervisor processes, which are responsible for starting processes, and handling their termination by optionally restarting them. Such supervisor processes set the `trap_exit` flag to have termination message delivered to their message queues. Their clients on the other hand generally do not set the `trap_exit` flag, since they do not contain programming logic to handle faults.

Many Erlang programs are written to be fault-tolerant, using the linking or monitoring mechanism, and although using ready-made components[4] make the task easier, programming fault tolerant applications is still *hard*, and being able to check code under adverse run-time conditions using a tool such as our model checker is a significant help.

Ensuring Finite Models Clearly the efficacy of the model checking algorithm depends crucially on whether the checked Erlang program is finite state or not. However, note that for checking non-compliance this is not always necessary. For instance, we can easily code a monitor that raises an alarm whenever a process mailbox contains more than, say, N messages. Similarly, an abstraction (see the discussion in section 4.5) could simply cut the mailbox when it has grown too large.

Still, in model checking Erlang there are at least two sources of trivially infinite models that we need to avoid: the assigning of process identifiers to new processes, and the use of unique references to uniquely identify (generic server) calls. We solve both problems by consistently choosing the least *fresh* process identifier (or communication tag) absent from both the current program state and the correctness monitor.

6. Evaluation

To evaluate the use of McErlang we have used it on several non-trivial examples, ranging from a resource locker to a Video–on–demand server. Here we focus on two examples, first a simplified resource manager (or locker) originally implemented and verified by Arts et al. (2004b). Their locker is based on a real implementation in the control software of the AXD 301 ATM switch developed by Ericsson. The second example is an implementation of a leader election algorithm. The implementation is (loosely) inspired by an algorithm presented by Singh (1996). Also this example originates from the AXD 301 ATM switch, but the particular implementation we studied here (and which have been studied before by Arts et al. (2005)) is an open source version written by Wiger.

The two examples aims to show different aspects of McErlang, the locker example is comparing McErlang with `etomcrl` and does not use the distributed features of McErlang. On the other hand, the leader election example is distributed (and fault-tolerant) and the example shows that it is possible to find errors in a distributed application with McErlang.

Other case studies realized using McErlang include the verification of an implementation of the Chord peer–to–peer protocol (Stoica et al. 2001), another implementation of a leader election algorithm namely Stoller's leader election algorithm (Stoller 1997), and of the above mentioned Video–on–demand server (Fredlund and Sánchez Penas 2007).

6.1 Resource manager

The locker is responsible for a number of resources, to which it can give clients exclusive or shared access, and which can survive client failures. To compare performance with the `etomcrl` tool we here focus on checking a single property[5]: is the locker safe with regards to mutual exclusion? That is, if a client requests exclusive access to a resource, and is granted access, then no other client will access the resource.

The source code of the example is split into four Erlang modules (files): (1) a module implementing a (parametric) client repeatedly accessing the locker using the `gen_server` OTP client-server component, (2) the source of a fault–tolerant locker, (3) a module implementing a supervisor process for starting the clients (using the `supervisor` OTP component), and (4) a supervisor that starts both the server and the client supervisor. In total around 430 lines of Erlang code.

The mutual exclusion monitor is provided in a separate Erlang module (around 60 additional lines of code); it checks whether multiple clients think they have access to the same resource, and at least one client has exclusive access (a mutual exclusion failure). In the client source we make visible the property of having access to resource by introducing a state using the `pause` value: {`pause`,{`?MODULE`,`inUse`,[*Resources*]}} which documents the resources and lock types the client thinks it has acquired.

Results As a comparison with `etomcrl` we present some figures for the checking of the locker example in table 1 below. The configuration column indicates, in a schematic manner, the model checking scenario used. For instance aEaEaEaEaS is a configuration with four clients requesting exclusive access to the resource a, and one client requesting shared access. The timing column shows the time for generating the transition system (for `etomcrl`, via the instantiator tool) and both the time to generate the transition system and check the mutex property for McErlang. The states column represents the number of states in the generated models. Note that for McErlang we use a non-lossy hash-table to store the state table.

	`etomcrl`		McErlang	
configuration	time	states	time	states
aEaEaEaEaE	52s	34282	17s	52197
aEaEaEaEaS	36s	28014	17s	50805
aEaEaEaSaS	39s	30814	18s	56313
aEaEaSaSaS	1m 4s	51928	25s	75801
aEaSaSaSaS	2m 49s	135038	42s	130101
aSaSaSaSaS	9m 29s	466702	1m39s	284277

Table 1. Comparison of `etomcrl` and McErlang

The table shows that in less complex scenarios, `etomcrl` creates smaller state spaces than McErlang. However, in complex scenarios (a scenario with more sharing is more complex, since many processes can request and succeed in getting a sharing lock on a resource at the same time) the difference in number of states evens out. The tool experiments were performed on a HP xw6400 workstation with four Intel Xeon CPUS each running at 1.60GHz (although neither tool made us of more than one CPU) and with 2 GB of memory, running Ubuntu 7.04.

It is hard to draw firm conclusions from the performance figures, although it is a promising sign that the time needed to generate the transition system using McErlang is competitive with the instantiator tool (Wouters 2001), as the instantiator is written in C and can be expected to be heavily optimized by now.[6]

[4] Such as, for example, the OTP supervisor pattern and the OTP generic server that are prepared to handle errors.

[5] Since `etomcrl` in contrast with McErlang does not support checking fault tolerance we did not introduce failures in the checked model; this was done in a separate experiment.

[6] Version 2.17.13 of the μCRL toolset was used.

6.2 Leader election

The objective of the leader election algorithm is to elect a leader among a fixed set of participants. This may seem trivial at first, but in a distributed and fault tolerant setting there are many subtle things that makes it a hard problem (and a well studied problem (Lynch 1996; Dolev et al. 1997) as well). Each *node* has a single leader election process, and the processes communicate with messages and also uses monitors to detect failures of other processes. There are two basic properties for leader election:

- **Safety** – two processes can never be elected as leaders at the same time.
- **Liveness** – eventually a process must be elected as the leader (or there is an infinite sequence of processes dying and restarting).

Both can easily be expressed as LTL-formulas (and hence as Büchi automatons). Here we focus mainly on the safety property.

To illustrate the typical organization of a verification we provide some details regarding the concrete files involved. The source code of the example is split into three Erlang modules (files): (1) a module implementing the leader election algorithm, (2) an environment for the leader election algorithm, and (3) a module that contains the monitor for the safety property. The test scenario is schematically illustrated in Fig. 13.

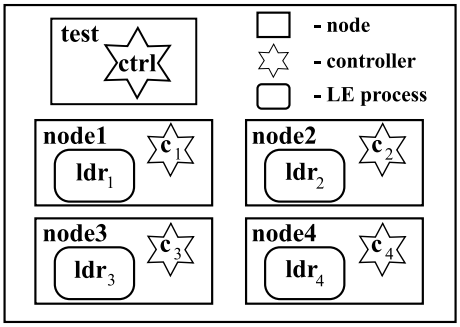

Figure 13. Leader election example organization

The environment module consists of code that initiates a set of nodes and starts a leader election process on each node. The environment also spawn controller processes (one for each node) that are responsible for killing and restarting the local leader election process. The controller processes in turn are dictated by a central stimuli generator (located on a separate node). The central controller sends messages to the local controller processes, which then enforces the order from the central controller (i.e., either killing or restarting the leader election process). All communication between conrollers are normal Erlang communication and it is all part of the model checking experiment. The reason for this somewhat strange stimuli generation structure stems from earlier testing, where we used *tracing* in a way which worked best with this structure. However, this is a good example of one of the strengths of the everything–in–Erlang approach, where the code from testing can be re-used (almost as is) as the environment description in verification. Also, the flexibility of having the environment in a separate module (which consists of ordinary Erlang code) is that we could easily do a verification of only the start-up phase (or some other part of the state space, such as just killing the process with highest priority) by just changing the module with the stimuli code. Originally, the test code provides random stimuli, which is not very suitable for model checking. The randomness is removed in our example by setting the pseudo-random generator seed to a fixed value.

The monitor for the safety property is not very complicated, it only consists of a check if there are two leaders elected in the current system state. The property monitor is listed in Fig. 14. One

```
-module(monNotTwoLeaders).

init(State) ->
  {ok,{safety,State}}.

stateChange(State,MonState,_) ->
  case notTwoLeaders(stRecords(allProcs(State))) of
    true ->  {ok,State};
    false -> {error,stRecords(allProcs(State))}
  end.

allProcs(State) ->
  lists:flatmap
    (fun (Node) -> Node#node.processes end,
     State#state.nodes).

stRecords([]) -> [];
stRecords([P|Rest]) ->
  case P#process.expr of
    {recv,{ev_gen_server2,_,{Rec,_}}} ->
      [Rec|stRecords(Rest)];
    _ ->
      stRecords(Rest)
  end.

isLeader({P,{_,State}}) ->
  Ldr = State#data.leader,
  P#process.pid == Ldr.

notTwoLeaders(States) ->
  length(lists:filter(fun isLeader/1,States)) < 2.
```

Figure 14. Safety property monitor – NotTwoLeaders

thing that is clear from the listing in Fig. 14 is the need for a set of convenience functions for accessing the states and retrieving information from the state.

Results If the example is run in McErlang using the safety algorithm, and the `NotTwoLeaders` monitor the result is a counter example. The time it takes to reach a counter example is only a few seconds (depending on the seed choosen it can take longer or shorter time) on a fairly modest workstation. The size/length of the counter example includes around 50 transitions. The existence of a counter example is not surprising, since other studies of the same algorithm (Arts et al. 2005) have revealed errors. (The counter example described below is actually exactly the same as the one labeled 'The first serious bug' in that paper)

The counter example scenario is described in Fig. 15. The problem in the scenario is that some communication is slower than other. Since in the protocol only a majority of the involved processes needs to accept a candidate it is possible that an existing leader (B in the scenario) could be outnumbered by newly started and fast communicating processes (A and C in the scenario).

What is important to note is that the error found is only present in a distributed and fault tolerant semantics. That is, we could not have found this error using a model checker (or other verification tool) that does not support the distributed semantics of Erlang. We also have the possibility to search for the shortest path leading to an error (again what is the shortest vary due to the introduced randomness). Having the shortest counter example is often desirable since it includes the least amount of unnecessary information. A search for the shortest path to an error is of course slower, sometimes several order of magnitudes slower. In one of our examples a search took about 30 minutes, and explored somewhere around 10 million states.

```
Three processes A,B,C (with priority A > B > C):
B is started
B: Send 'capture' to A,C and monitor A,C.
B: Receive 'Down' from A.
B: Receive 'Down' from C, broadcast 'elected'.
B is the leader
A is started
C is started
A: Send 'capture' to B,C and monitor B,C.
C: Receive 'capture' from A, Send 'accept' to A.
A: Receive 'accept' from C, broadcast 'elected'.
A is the leader
```

Figure 15. Counter example from leader election

7. Discussion

In this section we want to discuss some alternative implementation aspects. As mentioned in the introduction we made some experiments with a prototype implementation of the distributed Erlang semantics in Haskell. The prototype consisted of an Erlang parser and a layered run-time system with flexible control of path choice, etc. It supported all the distributed features of Erlang, but a lot of the more basic pure functional things were missing.

We asked ourselves if it would be possible to use such an implementation as the starting point for a model checker for Erlang as well. Much of the work with McErlang has gone into accurately modeling the node level semantics of Erlang. Starting instead with an implementation of the distributed semantics that task would be much simpler. We also think that a lot of the modular structure of McErlang could be the same in a Haskell implementation.

We have identified some advantages with a Haskell approach as well as some drawbacks. One of the major drawbacks is that one looses the ability to re-use the existing evaluation mechanisms for the purely functional part. This means that every lower-level built-in pure function and data structure has to be dealt with in the implementation. To implement this is perhaps not a very complicated task, however we deemed it as far too time-consuming for a research project. On the other hand, by having full control of the whole run-time system we could omit the Erlang–to–Erlang compilation phase discussed in section 5.2. It would also be trivial to switch from a *big-step* semantics to a *small-step* semantics since we could easily turn other syntactic constructions into choice points. A final drawback is of course also that we miss the "all-in-Erlang" aspect, since we involve Haskell. This could be a hinder for an experienced Erlang programmer with limited Haskell knowledge.

Our conclusion is that it is certainly possible to implement the same type of model checker in Haskell. However, it seems to be a lot more time-consuming, and it is not obvious that the end result would be any better than McErlang.

8. Related Work

Software model checking is a very active research field, which means that there exist an overwhelming amount of related works. We try to mention the most important and the ones which have provided inspiration for McErlang.

For Erlang the `etomcrl` toolset (Arts et al. 2004a) already provides a model checking capability. Although it is more restricted, covering a smaller subset of Erlang, for instance lacking the concept of distribution and fault tolerance (i.e. nodes, processes, links, monitors, ...). Other verification tools for Erlang include Huch's abstract interpretation model checker (Huch 1999) which uses abstract interpretations to reduce the size of the state space. We also have the "*Verification of Erlang Programs*"-project (Fredlund et al. 2003) which uses theorem proving technology. Further there is the interesting QuickCheck tool for Erlang by Arts and Hughes (2003), which however is more of a testing tool than a verification tool as it cannot detect recurring states.

The work on tracing for Erlang, in particular the approaches that have used *abstractions* to handle the size of the traces, by Arts and Fredlund (2002) and by Arts et al. (2005) was also a source of inspiration for the abstraction part of the McErlang implementation.

A lot of the inspiration for this work naturally comes from the work on the SPIN tool by Holzmann (1991) and the CADP toolset (Fernandez et al. 1996), as they both constitute very capable language based platforms for the verification of software, and for testing new verification algorithms.

The VeriSoft tool by Godefroid (1997) is one of the earlier examples of providing a verification functionality to a real, complex, programming language (such as C or C++) instead of a simpler specification language. Another successful example of such a verification project is the Modex tool (Holzmann and Smith 2002) which is closely connected to SPIN. A recent work on the verification of complex concurrent program code is the work on model checking file system implementations by Yang et al. (2004). Another recent work is the Zing model checker by Andrews et al. (2004) which aims at checking concurrent systems.

9. Conclusion and Future Work

As we have seen, adopting an "everything-in-Erlang" approach to model checking has certain advantages. It is easy to provide a rich specification language, and to use the same language for formulating correctness properties as for programming is convenient. Moreover much of the basic execution machinery can be reused (e.g., McErlang uses the normal Erlang run-time system extensively). The result is a model checker for Erlang, which supports all aspects of distribution and fault tolerance. This is especially important since distributed and fault-tolerant implementations are known to be error prone and hard to test and debug. It is our hope that McErlang is also simple enough to use, such that it can be used by the ordinary Erlang programmer.

With two examples we have compared McErlang with the existing `etomcrl` tool set and also showed that it is indeed possible to find errors in a distributed program using McErlang. The performance of McErlang looks promising, and the trade-off between expressive power and efficiency seems positive. However, more case studies are needed before we can be certain about the capacity of McErlang.

Another good property of McErlang, is the clearly separated input. We can easily experiment with different environment constraints for a program under test. This is particularly useful if one is only interested in part of the complete state space, since the search space could easily be altered by changing the environment constraints as we saw in the leader election example in section 6.2.

We have also experimented with an alternative implementation approach using Haskell. There we concluded that although it is a possible alternative it is far from obvious that the result would be better than McErlang.

During the development of the McErlang tool we also realized that a (dynamically typed) functional language offers several advantages over traditional languages like C as a general framework for implementing formal verification tools (e.g., quick prototyping, clean higher-order functions, separating functionality cleanly into modules, seamless composition of modules, and so on). Thus we have started experimenting with the use of the McErlang tool as a general framework for building model checkers for various target languages. Essentially this involves provides an executable operational semantics for the target language in question, together with the glue necessary (state parsers and unparsers, and so on). As a small experiment we implemented a simple interpreter and model checker for the WS-CDL web choreography language (W3C 2005).

Future work The tool is far from finished, there are many things that we want to investigate further, the following list indicates some of these areas:

- We would like to experiment with partial-order verification algorithms for the model checker. Clearly such reductions are normally quite language specific, and it will be instructive to see whether we can express their enabling conditions cleanly in Erlang. Moreover we can hope to benefit from the fact that standard components are heavily used in Erlang, which should results in more regular communication exchanges, i.e., which are more amenable to partial order reductions.
- To use McErlang on a larger body of programs we need to support a slightly richer Erlang fragment (e.g. the `port` construct for communicating with the external world). In particular it would be interesting to have a normal Erlang node communicate with nodes in our "modeled" Erlang environment.
- We should provide the option of changing the Erlang semantics implemented in the tool to re-schedule processes not only when a receive statement is encountered, but to do so for every side-effect inducing operation (e.g. message sends). This will result in a small-step semantics option that may detect new program bugs.
- Since many aspects of Erlang (asynchronous message passing, rich error detection mechanisms and process fairness) closely match standard implementation environments for distributed algorithms. Therefore, it seems reasonable to think that McErlang can be really useful also for verification of general distributed algorithms. The *leader election algorithm* example, presented in section 6, could be seen as one example of such an algorithm.
- We would like to develop a library of useful state abstractors for Erlang to enable this part of the tool to see wider use.

Acknowledgement

Thanks are due to Clara Benac Earle, Juan José Sánchez Penas, Koen Claessen and Thomas Arts.

References

T. Andrews, S. Qadeer, S.K. Rajamani, J. Rehof, and Y. Xie. Zing: A model checker for concurrent software. In *Lecture Notes in Computer Science*, volume Vol. 3114, pages 484 – 487, Jan 2004.

J. Armstrong. *Programming Erlang – Software for a Concurrent World*. The Pragmatic Programmers, http://books.pragprog.com/titles/jaerlang, 2007.

J. Armstrong. *Making reliable distributed systems in the presence of software errors*. PhD thesis, Royal Institute of Technology, Stockholm, Sweden, December 2003.

J. Armstrong, R. Virding, C. Wikström, and M. Williams. *Concurrent Programming in Erlang*. Prentice-Hall, 1996.

T. Arts and L. Fredlund. Trace analysis of Erlang programs. *SIGPLAN Not.*, 37(12), 2002. ISSN 0362-1340.

T. Arts and J. Hughes. QuickCheck for Erlang. In *Proceedings of the 2003 Erlang User Conference (EUC)*, 2003.

T. Arts, C. Benac Earle, and J. J. Sánchez Penas. Translating Erlang to mucrl. In *Proceedings of the International Conference on Application of Concurrency to System Design (ACSD2004)*. IEEE Computer Society Press, June 2004a.

T. Arts, C. Benac Earle, and J. Derrick. Development of a verified Erlang program for resource locking. *International Journal on Software Tools for Technology Transfer (STTT)*, 5(2–3):205–220, March 2004b.

T. Arts, K. Claessen, and H. Svensson. Semi-formal development of a fault-tolerant leader election protocol in Erlang. *Lecture Notes in Computer Science*, 3395:140 – 154, January 2005.

S. Blau and J. Rooth. AXD 301 - a new generation ATM switching system. *Ericsson Review*, 1:10–17, 1998.

K. Claessen and H. Svensson. A semantics for distributed Erlang. In *Proceedings of the ACM SIPGLAN 2005 Erlang Workshop*, 2005.

S. Dolev, A. Israeli, and S. Moran. Uniform dynamic self-stabilizing leader election. *IEEE Trans. Parallel Distrib. Syst.*, 8(4):424–440, 1997. ISSN 1045-9219. doi: http://dx.doi.org/10.1109/71.588622.

Erlang 5.5/OTP R11B. The Erlang/OTP Team. URL `http://www.erlang.org/doc/doc-5.5/doc/highlights.html`.

J.-C. Fernandez, H. Garavel, A. Kerbrat, L. Mounier, R. Mateescu, and M. Sighireanu. CADP: A protocol validation and verification toolbox. In *Proceedings of the 8th Conference on Computer-Aided Verification*, volume 1102 of *Lecture Notes in Computer Science*, pages 437–440. Springer, 1996.

L. Fredlund. *A Framework for Reasoning about Erlang Code*. PhD thesis, Royal Institute of Technology, Stockholm, Sweden, 2001.

L. Fredlund. Implementing WS-CDL. In *Proceedings of the second Spanish workshop on Web Technologies (JSWEB 2006)*. Universidade de Santiago de Compostela, November 2006.

L. Fredlund and J.J. Sánchez Penas. Model checking a VoD server using McErlang. In *In proceedings of the 2007 Eurocast conference*, Feb 2007.

L. 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 (STTT)*, 4(4):405 – 420, Aug 2003.

P. Godefroid. Verisoft: A tool for the automatic analysis of concurrent reactive software. In *Computer Aided Verification*, pages 476–479, 1997.

G. Holzmann. *Design and Validation of Computer Protocols*. Prentice-Hall, 1991.

G. Holzmann, D. Peled, and M. Yannakakis. On nested depth first search. In *Proc. Second SPIN Workshop*, pages 23–32. American Mathematical Society, 1996.

G. J. Holzmann and M. H. Smith. An automated verification method for distributed systems software based on model extraction. *IEEE Trans. Softw. Eng.*, 28(4):364–377, 2002. ISSN 0098-5589.

F. Huch. Verification of Erlang programs using abstract interpretation and model checking. In *Proceedings of the 1999 ACM SIGPLAN International Conference on Functional Programming*, 1999.

N. Lynch. *Distributed Algorithms*. Morgan Kaufmann Publishers, 1996.

G. Singh. Leader election in the presence of link failures. In *IEEE Transactions on Parallel and Distributed Systems, Vol 7*. IEEE computer society, 1996.

I. Stoica, R. Morris, D. Karger, F. Kaashoek, and H. Balakrishnan. Chord: A scalable Peer-To-Peer lookup service for internet applications. In *Proceedings of the 2001 ACM SIGCOMM Conference*, pages 149–160, 2001. URL `citeseer.ist.psu.edu/stoica01chord.html`.

S. D. Stoller. Leader election in distributed systems with crash failures. Technical Report 481, Computer Science Dept., Indiana University, May 1997. Revised July 1997.

S. Torstendahl. Open telecom platform. *Ericsson Review*, 1, 1997.

M. Y. Vardi and P. Wolper. An automata-theoretic approach to automatic program verification. pages 332–344, 1986.

W3C. Web Services Choreography Description Language, Version 1.0 – W3C candidate recommendation 9 november 2005. Technical report, W3C, November 2005.

U. Wiger. Fault tolerant leader election. URL `http://www.erlang.org/`.

A.G. Wouters. Manual for the μCRL toolset. Technical report, CWI, Amsterdam, 2001.

J. Yang, P. Twohey, D. Engler, and M. Musuvathi. Using model checking to find serious file system errors. In *Sixth Symposium on Operating Systems Design and Implementation*, pages 273–288. USENIX, 2004.