Model Checking Erlang Programs: The Functional Approach

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

Clara Benac Earle
Departamento de Informática
Universidad Carlos III de Madrid
cbenac@inf.uc3m.es

Abstract

We present the new model checker McErlang for verifying Erlang programs. In comparison with the etomcrl tool set, McErlang differs mainly in that it is implemented in Erlang. The implementation language offers several advantages: checkable programs use "almost" normal Erlang, correctness properties are formulated in Erlang itself instead of a temporal logic, and it is easier to properly diagnose program bugs discovered by the model checker. In addition the model checker can easily be modified, thanks largely to the use of Erlang. The drawback of writing the model checker in Erlang is, potentially, severely reduced performance compared with model checking tools programmed in programming languages which permit destructive updates of data structures.

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

General Terms Verification

Keywords Code Verification, Concurrency

1. Acknowledgement

Thanks are due to Thomas Arts, Juan José Sánchez Penas, Hans Svensson and the anonymous referees.

2. Introduction

Early on in the development of the etomcrl tool set [1, 3] [1] its principal authors (Thomas Arts and Clara Benac Earle) discussed whether to implement their approach to model checking Erlang programs through the translation into μCRL [8], and use third-party model checking tools, or whether to implement the model checking algorithms directly in Erlang. The Erlang route was rejected because the authors thought that it would be more efficient to reuse existing quality tools for model checking.

This paper documents an attempt to follow the second route above, i.e., to provide an model checking framework for Erlang programs implemented in Erlang.

* The author was supported by a Ramon y Cajal grant from the Ministerio de Educacion Y Ciencia.

[1] comprising a translator from Erlang to μCRL, a state space generator for μCRL specifications, and the CADP state space analysis tools

Erlang'06 September 16, 2006, Portland, Oregon, USA.

The main current differences between McErlang and etomcrl, for a user, can be summarised as follows:

- McErlang supports a larger fragment of the Erlang language, especially with respect to the evaluation of functions without side effects (which in McErlang are evaluated by the normal Erlang run time system).
- McErlang provides support for the basic communication mechanisms of Erlang: i.e., sending and receiving, and process linking, whereas in etomcrl processes can only communicate with generic servers (in McErlang generic servers are, as in normal Erlang, implemented on top of the basic communication mechanism).
- The closer integration with McErlang and the runtime system for Erlang makes it easier to implement alternative semantics for Erlang, and to check a program under those semantic assumptions. For instance, although the node functions (e.g., spawning a process on a remote node) are currently not supported a user can choose to check programs either under the assumption that all processes execute on a single node, or under the assumption that all processes execute on different nodes. Similarly, for checking fault-tolerance, McErlang provides the option to at any moment kill an executing process.
- Program correctness properties in etomcrl are often formulated in a branching-time temporal logic over the actions of the Erlang program, and checked using the CADP [7] tool. In contrast, in McErlang correctness properties are written as ordinary Erlang functions that can inspect the current program state. Currently the McErlang tool supports only checking of safety properties (e.g., "the bad state A is never reached"); hopefully this will change in the near future.
- As the model checking part of the etomcrl tool set is implemented in C it can be expected to be much more efficient than the McErlang counterpart implemented in Erlang. Neither have the McErlang model checker tool been optimised for speed or memory efficiency (crucial properties in a model checker.

Given especially these efficiency drawbacks why is it worth considering writing a model checker in Erlang? Because of a key reason Erlang (and other scripting languages) is popular in the first place: an untyped programming language, where program text can be constructed and deconstructed on the fly, stored as data, and can be executed on demand (using apply) is a very capable experimental platform; especially so when experimenting with program analysis[2].

The hope is that the programming *flexibility* such a platform offers outweighs any execution efficiencies. For the McErlang tool,

[2] In our opinion Erlang is a very capable scripting language with a clearly very superior concurrency model compared to other such languages; it is just a pity that the rest of the world hasn't caught on yet...

although still just a prototype, early evidence points to such advantages with regards to the ease with which complex correctness properties can be formulated, the possibility to integrate powerful state abstractors for reducing the state spaces explored, and the rich specification language (normal Erlang, except that a number of side-effect inducing API functions are not available).

The main idea of the `McErlang` tool is to replace the part of the ERTS runtime system that implements process concurrency with a new runtime system suitable for model checking. In other words, the model checker replaces basic Erlang operations such as process spawning, communication and process linking, and corresponding data structures for recording processes, with its own data structures for recording processes and its own code for interacting with processes.

Model checking a program is achieved by running the program, in the new runtime system, in lock-step with an automaton (also implemented in Erlang) that represents the correctness property to check. The automaton has the power to inspect the state of the running Erlang program, and will halt the execution of the program if a correctness violation is detected.

Let us define an Erlang state as the collection of currently executing Erlang processes, together with their process identifiers, process mailboxes, the links connecting processes, as well as any other data structure needed during the execution of an Erlang application.

The basic new operations upon such Erlang states that the model checker requires are:

- From a state s, the possibility to compute all next states $s_1, \ldots, s_n$. Multiple next states are possibly due to the interleaving notion of concurrency we use, as well as the introduction of an explicit non-deterministic construct in our Erlang variant.
- Checkpoint and restore the current state. That is, the possibility to store the current state (and next states) in (Erlang) data structures, and later retrieve them from these data structures to compute next states.
- The possibility to treat a system state as an ordinary Erlang data item in order to compute abstractions over the state (e.g. to compute a hash value for use in the state table implementation, and to check whether two states are equal up to some notion of state equality), as well as general state inspection functions used in program monitors checking whether a state complies with an associated correctness property.

One option would have been to add this functionality directly to ERTS. Another option would have been to use a language reflection capability, should a good such facility have existed for Erlang. Presumably calls to Erlang kernel functions such as `link` could have been redirected; however the handling of the `receive` statement would likely present problems.

To avoid having to modify Erlang we opted instead to modify the program to check; to replace in the source code calls to functions with side-effects with calls to the runtime system of the model checker, and to replace instances of `receive` constructs with a new model checking construct.

In section 3 we explore the technical underpinnings and details of the execution model of the `McErlang` tool, section 4 describes the model checking features of the tool while section 5 contains a case study on the checking of the (classical) locker example of Arts and Benac Earle in `McErlang`. Finally section 6 draws some preliminary conclusions about the use of the tool and its future.

3. Execution Model

Here we describe the execution model of Erlang programs running under the runtime system of the model checker `McErlang`.

3.1 Model Checker Runtime System

As previously described the `McErlang` checker replaces the concurrent part of the normal Erlang runtime system which its own. The new runtime system contains data structures for recording processes and process related data structures.

3.1.1 Processes

Informally, a process is a tuple having five components `{Status,Expr,Pid,Queue,CommQueue,Flags}` where `Status` records whether a process is `runnable`, `receivable`, `blocked` and so on (e.g., a process is `blocked` when it is waiting in a `receive` statement and no message in its input queue can be received). `Expr` either contains the next function to be invoked (a triple `{Mod,Fun,Args}`), or the equivalent of a receive statement (see section 3.2.1 below) or a non-deterministic choice. The field `Pid` is the unique process identifier of the process, `Queue` is the process mailbox and `Flags` records, for instance, whether exit messages should be received by the process in its mailbox. `CommQueue` is used when a program is checked under the assumption that all processes are located on different nodes. In that case messages directed to a process are not delivered directly in its mailbox, instead they are stored first in `CommQueue` which is a data structures containing tuples `{SendingPid,Messages}` where `Messages` is a queue sorted in arrival order (to guarantee the proper Erlang message semantics [6]). Non-deterministically a process can, at any time, choose to move a message from `CommQueue` to `Queue`.

3.1.2 System

A system in the model checker runtime system is a set of processes, a data structure for associating names with pids (to implement `register`), and a set of tuples `{Pid1,Pid2}` (for implementing `link`). Still absent in the checker is, for instance, monitors. Currently this global system state is kept in a separate Erlang `gen_server` process, and modifications to it, or queries, are made through the `gen_server` API interface.

3.1.3 API

Erlang programs running in the model checker invoke the runtime system normally, except that for instance calls to spawn a new processes should be made to `evOS:spawn`, instead of just `spawn`. The module `evOS` currently provides a subset of the normal Erlang functions for process spawning, process linking, sending, `self`, `register`, `exit`, `process_flag`, etc.

3.2 Execution Model

To invoke the model checker the user should provide:

- the name of the initial function to run, and its arguments
- the name of a "monitor module" that implements the correctness property to check
- the name of an "abstraction module" that abstracts program states, and provides a state table
- a set of checking options, such as e.g. whether the remote semantics should be used, whether processes should be crashed non-deterministically, etc.

The role of the monitor and the abstraction modules in the model checking process will be explained in section 4. Here only the execution of the Erlang program will be informally described.

The model checker begins by creating a new process (note: not an Erlang process but a simulated process in its own runtime system), marks it as runnable and begins executing the initial function using the normal Erlang function `apply`. The execution of the function may cause side effects which are implemented by the `evOS`

module; an example is a call to `evOS:send(Pid,Value)` to send a message to another process. Such API calls result in immediate changes to the global model checking state, via communication with the `gen_server` process keeping the global state.

When any such application returns[3] its return value is checked. If it is a normal value, or an exception was raised, the process has terminated and linked process are notified of this fact. The model checker then selects another ready process to run.

Apart from immediately terminating, a function may also return one of the so called "special return values" that signify that the associated process should not yet terminate, but that it has reached a "stable state".

Special Return Values These special return values are:

- `{recv, {Mod,RecvFun,Args}}`
- `{letexp, {Expr, {Mod,LetFun,Args}}}`
- `{choice, [{Mod,Fun,Args}|...]}`
- `{pause, {Mod,Fun,Args}}`

3.2.1 Handling Receives

Instances of `receive` statements in Erlang code to be model checked have to be replaced with code that instead returns a `{recv, {Mod,RecvFun,Args}}` tuple above. When an invoked Erlang function, in an Erlang process, returns such a `recv` tuple the runtime system 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 runtime system can schedule another ready process.

The transformation of an Erlang program containing a `receive` into one using a `recv` expression will be explained by a small example below.

```
doRequest(State) ->
  receive
    {call, F, Pid} when is_pid(Pid) ->
      {Reply, NewState} = apply(F, State),
      Pid!{reply, Reply},
      {ok, NewState}
  end.
```

The code defines a function which guards some private state data. The state can be changed by sending a call message to the server process, containing a function that should return a new state and a return value. The transformation of the above function yields the following Erlang fragment:

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

f_0({call, F, Pid}, [State]) when is_pid(Pid) ->
  {true,
   fun ({call, F, Pid}, [State]) ->
     {Reply, NewState} = apply(F, State),
     evOS:send(Pid,{reply,Reply}),
     {ok, NewState}
   end};
f_0(_, _) ->
   false.
```

Thus, a call to `doRequest(State)` will immediately return a tuple `{recv, {?MODULE, f_0, [State]}}` which is recognised by the model checker. The function argument to `recv` should refer to a function that accepts two arguments, a message from the queue which should be tested whether it is readable, and a set of variables needed in the evaluation of the `receive`.

The function should return `{true, AF}`, where `AF` is an anonymous function, if the message matches. If the message doesn't match, `false` should be returned. The function `AF` receives the same argument as the original function, and contains the body of the `receive` clause.

The separation of a `receive` clause into two functions serves to separate the testing whether a message is receivable from the actual reading of the message from the queue. Moreover, the use of an actual named function in the `recv` tuple helps during the checking of state equality in the model checker.

3.2.2 Receives in an Expression Context

The special return value `{letexp, {Expr, {Mod,LetFun,Args}}}` is useful in the situation when a receive statement occurs in an expression context (not in a tail-recursive position). Consider a recursive function `server` which acts similarly to a `gen_server`:

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

This should be translated into:

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

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

The function referenced in the `letexp` is called when the inner function has returned a value, and receives as parameters the returned value as first argument and as second argument a list of variables necessary to compute the continuation.

Note that all non tail recursive calls to functions that contain a `receive` in their body will have to be guarded using a `letexp` in a manner similar to the above code fragment.

3.2.3 Other Special Return Values

The special return value `{choice, [{Mod,Fun,Args}|...]}` introduces explicit non-determinism in Erlang; the model checker will non-deterministically select the continuation function from the list of functions in the parameter to `choice`.

Finally `{pause, {Mod,Fun,Args}}` is just short hand for a choice with a single continuation function. The `choice` construct is mostly useful to facilitate detection of interesting states in the correctness properties.

3.3 Translation

A prototype translation exists for translating a sizable fragment of Erlang code to Erlang code acceptable as input to the `McErlang` model checker, according to the translation schema illustrated in the previous subsection. The translation takes one module as input and returning a translated one.

Note that the resulting Erlang modules, for use in the model checker, represent syntactically completely legal Erlang modules and can thus be compiled by the normal Erlang compiler (and of course such a compilation is necessary to use the model checker).

The translator substitutes calls to functions as `spawn` to the corresponding ones in the `evOS` module, it replaces `receive` statement in expression contexts with `recv` and `letexp` tuples, and so on. Although the translator using the `erl_syntax` libraries to streamline the translation, not all translation rules are trivial due to the semantics of Erlang variable binding which can export variable bindings from deep inside a sub-expression to the context of the sub-expression.

[3] if it doesn't return the model checker will hang forever

Similarly to the case for `etomcrl`, an integral part of the translation is the (transitive) computation of the Erlang functions that have side effects (i.e., contains a receive statement). That is, if a function `g` calls a function `f` that has side effects, then indeed calling `g` may also cause side effects and so `g` is also marked as a side-effect inducing function, and so on. In essence, we are computing the least fixed point of the set of functions containing side effects, due to their calling graphs.

3.4 Semantics

The approach to model checking embodied by the `McErlang` tool can be described as *pragmatic*. We consider verification using model checking to be, fundamentally, a debugging activity which should complement other debugging activities such as using `Dialyzer` [10], `QuickCheck` [5] and tracing tools such as `TraceTool` [4, 2].

The semantic model, although not fully worked out yet, for an Erlang program executed by the `McErlang` tool is a (possibly infinite) state graph.

The structure of an Erlang system was covered in section 3.1.2; transitions between states are the result of choosing a runnable Erlang process (one that is ready to execute a function, or is ready to receive a message from the mailbox) and executing until it either terminates or returns a special return value (signalling a desire to receive another message, or that a non-deterministic choice needs to be resolved).

Since in general multiple processes are enabled in any given system state the model is non-deterministic.

If we were to label these transitions by, for instance, the communication actions that the invoked process initiates, these transitions would be labelled not by singleton actions, but rather by sequences of such actions.

Compared to a more fine-grained execution semantics, which would offer to pause the running process when any action with a side-effect (effecting another process) takes place, our semantics contains much fewer states. This means, possibly, that some bugs will go undetected in our model checker, but would be caught in another model checker. Similarly the process scheduler used in the model checker will never interrupt a running process; with this choice again we risk failing to detect a program bug. Moreover the program is checked using only a fixed ordering of sub-expression evaluation, and one which is not guaranteed to always to correspond to the de-facto left-to-right ordering used in Erlang compilers.

The reason for choosing such coarse execution steps is that (i) we expect to gain quite a lot in performance and model complexity (execution of Erlang code will be quicker since the model checker process scheduler will have to be invoked less often, and (ii) the size of the model (its number of states) will decrease considerably, and (iii) we can anyway recover the more finely-grained semantics using the so called "remote semantics" option (see next subsection), and (iv) repeating again, verification using model checking is a pragmatic activity concerned with helping debugging programs, not primarily concerned with providing absolute guarantees regarding correctness.

It remains an item for future development and research to extend the special return values to include any call to an Erlang function that has a side effect, in order to refine the resulting semantics.

3.4.1 Remote Semantics

Remote Semantics is an execution option for the `McErlang` model checker which (partly) implements the communication semantics of Claessen and Svensson [6]. If enabled, all communications (including sending messages, establishing links, and delivering `exit` messages) between Erlang processes will be subject to non-deterministic transmission delay (as modelled by having an intermediate communication buffer `CommQueue` in the process state, see explanation in section 3.1.1). The only communication guarantee is that between two processes P and Q, messages are delivered in order if they are delivered at all.

Naturally enabling this option greatly enlarges the state space of the analysed Erlang program.

Interestingly, enabling this option removes the need to refine the execution model to interleave processes after any side effect (say, a send), since the non-deterministic communication delay introduced by the communication buffer will have the same effect.

3.4.2 Process Termination

Remote Semantics is an execution option which is useful for testing the fault tolerance of a program. If enabled, the runtime system will in any program state include the possibility of killing any process.

3.5 Handling OTP libraries

The ideal situation for handling the OTP components would be to simply translate them, from their source code representation, to the core language of the model checker. However, the `McErlang` support libraries are not complete enough yet to make this a practical task. In addition, there is a strong possibility that by using a specially developed library we will be able to shrink the state space of typical Erlang programs using OTP libraries.

Currently `McErlang` only provides two OTP components, a severely cut down `supervisor` component for spawning processes (the support for monitoring and restarting processes is not yet implemented) and a generic server component (`gen_server`) for client–server communication. The `gen_server` library in `McErlang` is also limited in functionality, for instance there is no support for `multi_call` yet.

For completeness we include the source code of the `gen_server` module that we use in Appendix B. Note that for presentation purposes we have cut out some functionality present in the module such as creating a generic server without registering a name.

4. Model Checking

In this section we describe the approach to model checking used in `McErlang`, and how correctness properties can be formulated.

4.1 Correctness Properties

Program correctness properties are described directly in Erlang. Currently only safety properties can be expressed, i.e., properties concerning the absence of a bad state or action ("the bad state A is never reached", "action A is never invoked"). These are properties that needs to be checked in every reachable program state. In contrast, progress properties ("after the reception of a message, eventually a reply is sent") requires the usage of a slightly more complex correctness automata[4]; see [9] for details.

A correctness property is defined as a program monitor, i.e., essentially an Erlang function which is checked in every reachable program state. Intuitively a monitor is a simple automaton (the monitor also has its own state) that simply reacts to program transition.

An Erlang monitor must define two callback functions: `init(Parameters)` and `stateChange(ProgramState,MonitorState)`. The `init` function returns `{ok,MonState}` where `MonState` is the initial state of the monitor (as in a `gen_server` a state is an arbitrary Erlang value).

[4] Intuitively, we need to distinguish progressing loops from non-progressing loops

The function `stateChange(ProgramState,MonitorState)` is called when the model checker encounters a new program state `ProgramState` and the current monitor state is `MonitorState`. If the 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; any other value signals a violation of the correctness property implemented by the monitor.

As an example, Figure 1 contains a simple monitor that that guards against program deadlocks.

```
-module(monDeadlock).
-export([init/1,stateChange/2]).

-include("state.hrl").
-include("process.hrl").

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

stateChange(State,MonState) ->
  case lists:any(fun (P) -> not_deadlocked(P) end,
                 State#state.processes) of
    true ->
      {ok, MonState};
    false ->
      {deadlock, MonState}
  end.

not_deadlocked(P) ->
  P#process.status=/=blocked.
```

Figure 1. A monitor to detect deadlocks

4.2 Model Checking Algorithm

Currently the `McErlang` tool only offers a very simple depth-first state traversal model checking algorithm. In pseudo-code Erlang the algorithm can be described as in figure 2.

Given a starting call `m:f(P1,...,Pn)`, an initial monitor state `M` and an empty state table `T`, the checking algorithm should be invoked: `check([[{mkProc(m,f,[P1,...,Pn]),M,T}]],Mon,Abs,Tab)` where `mkProc` constructs a runnable model checking process, `Mon` is a monitor module, `Abs` is a module used for abstracting states (see next subsection), and `Tab` is a module implementing a state table.

The first argument to `check` is essentially a stack represented as a list where the stack levels represents the path from the first program state to the current one, and on each level, a list of alternative next states not yet explored.

As seen in the figure, model checking states are composed of a program state, a monitor state, and a state table. Newly generated program states are checked against the associated 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 truly new, the set of next states is computed using the function `doExec`. The set of next states includes the case where each runnable process will take at least a step (more, if there is a non-deterministic choice in the process). The checking then continues using a new stack frame containing the new model checking states, where the new program states have been coupled with the new monitor state and the new state table.

As can be seen, the program and the correctness property monitor essential run in lock-step during model checking.

```
% Execution finished
check([],Mon,Abs,Tab) -> ok;

% Backtrack to earlier states
check([[]|Earlier],Mon,Abs,Tab) ->
  check(Earlier,Mon,Abs,Tab);

check([[State|Alts]|Earlier],Mon,Abs,Tab) ->
  {ProgState,MonState,StateTab} = State,
  % Check monitor
  {ok,NewMonState} =
    apply(Mon,stateChange,[ProgState,MonState]),
  % Abstract state, and check whether seen
  {ok, AbsState} =
    apply(Abs,abstract_state,
          [ProgState,NewMonState]),

  case apply(Tab,permit_state,
             [AbsState,StateTab]) of
    no ->
      check([Alts|Earlier],Mon,Abs,Tab);

    {ok, NewStateTab} ->
      % Compute new states
      NewStates =
        [{S,NewMonState,NewStateTab} ||
           S <- doExec(ProgState)],
      check([NewStates,Alts|Earlier]],Mon,Abs,Tab)
  end
end.
```

Figure 2. Basic Model Checking Algorithm

If the monitor detects a violation of its associated correctness property it prints out a report describing the problem, and a trace from the initial state of the program leading to the problem state.

4.2.1 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 [4]. 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, signalling whether the integer is less than zero. Of course, there is in general no guarantee that such abstractions are safe.

4.2.2 State tables

A state table in `McErlang` implements a state table which record states encountered during the model checking (a state here is a combination of a program state and monitor state). The state table is used to halt the continued exploration of a state when an earlier identical state already has been checked.

4.2.3 Ensuring Finite Models

Clearly the efficacy of the model checking algorithm depends crucially on whether the Erlang program checked have finite state spaces 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.

Then, even if mailboxes grow without bound, we will be able to detect the problem. Similarly, an abstraction could simply cut the mailbox when it has grown too large.

Still, in model checking Erlang programs there are at least two sources of trivially infinite models that we normally need to take into account: process identifiers and unique references.

Process identifiers are assigned in the `spawn` function call, and a trivial implementation would simply assign increasing integer values in successive calls to `spawn` to distinguish different process identifiers. Such a discipline however directly leads to infinite models. Instead, we let `spawn` enumerate the process identifiers in use in either the current program state, or the current monitor state, and select the lowest process identifier (a natural number starting with 0) not currently in use. This is clearly an example where it is very useful having easy programmatic access to both the program state and the monitor state in the model checker; a strong argument for the "everything-in-Erlang" approach followed in this paper.

A similar re–use discipline is used to combat the problem of unique `gen_server` call tags generated by the `gen_server` library.

5. Example

To illustrate the use of `McErlang` on a non-trivial example, we use a simplified resource manager, the so called locker, implemented by Arts and Benac Earle [1]. The locker is based on a real implementation in the control software of the AXD 301 ATM switch developed by Ericsson and it was used as a case-study for the `etomcrl` tool.

5.1 Locker Implementation

The locker case-study consists of a server process (the locker) that provides exclusive or shared access to an arbitrary number of resources for an arbitrary number of client processes. As in the real software, the whole locker application is started as a supervision tree where there is a supervisor for the server and another supervisor for the clients.

The locker is implemented as a generic server callback module. The state of the locker contains the following information:

- a list containing the locks in the system. Each lock stores information about a resource, the clients accessing it with a particular type of access (shared or exclusive) and the clients waiting to get access to it.
- two lists, for storing the clients that request exclusive and shared access to the resources.

The code corresponding to the handle_call function for a request message sent by a client to the locker is given in figure 3.

A client requesting access to a list of resources will only obtain access to them when they are all available. If they are not available the client is put in the pending list of each lock, added to the corresponding list (Excls or Shared), and suspended until the resources are released.

5.2 Clients

The example client repeatedly asks for a set of resources, and then releases them, see figure 4. Note that a `pause` is inserted to indicate to the monitor implementing the correctness property that the client is in its critical region.

Both the server and the set of clients are started from a supervisor tree description.

5.3 Model Checking the Locker

In this small example we only check a single property: whether the locker is safe with regards to mutual exclusion. That is,

```
handle_call({request,Resources,Type},Client,
  {Locks,Excls,Shared}) ->
  case check_availables(Resources,Type,Locks) of
    true ->
      NewLocks =
        map(fun(Lock) ->
          claim_lock(Lock,Resources,Type,Client)
            end,Locks),
      {reply, ok,{NewLocks,Excls,Shared}};
    false ->
      NewLocks =
        map(fun(Lock) ->
          add_pending(Lock,Resources,Type,Client)
               end,Locks),
      case Type of
        excl ->
          {noreply,
            {NewLocks,Excls++[Client],Shared}};
        share ->
          {noreply,
            {NewLocks,Excls,Shared++[Client]}}
      end
  end;
```

Figure 3. implementation of the function `handle_call` for the `request` message

```
start_link(Locker,Resources,Type) ->
  {ok,spawn_link(?MODULE, loop,
                 [Locker,Resources,Type])}.

loop(Locker,Resources,Type) ->
  gen_server:call(Locker,
                  {request,Resources,Type}),
  {pause,{?MODULE,inUse,[Locker,Resources,Type]}}.

inUse(Locker,Resources,Type) ->
  gen_server:call(Locker, release),
  loop(Locker,Resources,Type).
```

Figure 4. a simple locker client

that if a client requests exclusive access to a resource, and is granted access, no other client can access the resource at the same time. As was explained above we instrument the client process to have a dedicated state, through a function named `inUse(Locker,Resources,Type)`. The fact that a process is pausing in this function signals to the monitor implementing the mutex correctness property that the client is in its critical region, and has access to its `Resources` under the access type (`share` or `excl`) signalled by the `Type` parameter.

The mutual exclusion correctness property is checked by the monitor in Appendix A. Although the code might appear complicated, the idea is simple. The monitor looks for all processes that are waiting in the function call `inUse` (`calculateResources`) and creates a list structure such that each of its elements is a tuple, `{Type,Name,Pids}` where `Type` is `excl` or `share`, `Name` is the name of the resource and `Pids` are the pids of the processes that have locked the resource.

In case a mutual exclusion violation is detected by the monitor (e.g., two processes holding an exclusive lock) the monitor returns an error indication.

	etomcrl		McErlang	
configuration	time	states	time	states
aEaEaEaEaE	1m 6s	9997	44s	52197
aEaEaEaEaS	45s	6033	42s	50805
aEaEaEaSaS	47s	6315	47s	56313
aEaEaSaSaS	1m 16s	14215	1m8s	75801
aEaSaSaSaS	4m 31s	59073	2m16s	130101
aSaSaSaSaS	21m	298437	7m48s	284277

Table 1. A comparison of `etomcrl` and `McErlang` using the locker example

As an experiment we modified one line of the locker `gen_server` implementation, changing the line

```
{reply, ok, {NewLocks,Excls,Shared}};
```

to read instead

```
{reply, ok, {Locks,Excls,Shared}};
```

that is, forgetting to update the lock structure. Trying to check the new locker, using more than one client attempting to access the same resource, the monitor rapidly signals that it has found a counterexample, and prints out a trace back to the starting state. As an option, it is possible to search for the shortest such counterexample path. For the locker example with above modification the shortest path was 11 transitions.

5.3.1 Preliminary Comparison with `etomcrl`

As a very preliminary comparison with `etomcrl` we present some figures for the checking of the locker example in a number of client configurations in table 1. The configuration column indicates, in a schematic manner, the model checking scenario used. For instance aEaEaEaEaS indicates a configuration with four client processes requesting exclusive access to the resource a, and one client process requesting shared access. The timing column indicates the time needed to generate 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. For `etomcrl` the timings were obtained on a SUN E450 with 4 297MHz UltraSparc II CPU's, with 2Gb of RAM, and 6Gb of Swap. For `McErlang` the timings were obtained on a Dell PC running Linux 2.6.8-3 on 4 2GHz XEON processors, with 2Gb of memory.

As can be seen in the table, `etomcrl` creates much smaller state spaces than `McErlang` in the smaller scenarios. However, in more complex scenarios[5] the difference in number of states evens out. In terms of execution speed the tools are roughly equal, except again, `McErlang` requires less time to complete the larger scenarios.

It is hard to draw firm conclusions from the above figures since they represents measurements on a very early version of `McErlang`. It appears, however, that `McErlang` generates more states; it could possibly be a result of the manner in which we spawn processes through the supervisor component, which is not optimised. Further investigations are required here.

It is a hopeful sign that the execution time for generating the transition system using `McErlang` is competitive with the instantiator tool [11], as the instantiator is written in C and can be expected to be heavily optimised by now.

[5] 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

6. Conclusions and Future Work

As we have seen adopting a "everything-in-Erlang" approach to model checking has certain advantages: it is easy to provide a richer specification language, and to use the same language for formulating correctness properties as for programming, and much of the basic execution machinery can be reused (e.g., the model checker uses the normal Erlang expression evaluation mechanism).

Still the current tool is just a prototype; there are many drawbacks to its use that need to be addressed. We enumerate a number of issues below:

- Finalising the translation from standard Erlang to the fragment supported by the model checker (e.g., replacing receive statements with receive tuples).
- Support for a richer Erlang fragment, e.g., including process monitors, nodes, and so on. Ideally we should be able to directly use the sources of standard Erlang components such as `gen_server`; however this may induce a cost in an increased number of states.
- The monitor model is too weak. We need to be able to specify/program Büchi automatons, to enable formulation and checking of liveness and fairness properties.
- We should investigate the possibility of changing the semantics; to provide the option not only of halting the execution of the current process when a receive is encountered, but to do so for every side-effect inducing operation (e.g., also for sends).
- Clearly a model checker is a highly performance critical tool, and probably some part of its critical functionality (maintaining a state table) represents a program task for which Erlang is not very well suited (lacking destructive data structures). Adding a so called bit-state hash table [9] option to `McErlang` for instance (a hash table which just keeps one bit of information for every item) would probably have a prohibitive cost if implemented using `ets` tables. An option would be to provide such an implementation in C instead.

  Interestingly, on the `erlang-questions` mailing list Sagonis leaked information on 23/04/06 about a destructive array feature available in HiPE:

  > ... I've kept quiet till now, because I did not want to reveal the HiPE magic to the world, but since I see that all Erlang solutions are ets-based, I find little reason not to send this post...
  >
  > Try the program below. Currently, it uses byte arrays rather than bits, but converting it to use bits should be straightforward. It is about 20 times faster than Yffe's [sic] program and about 10 times faster than Richard's.
  > ...

  We look forward to trying out the HiPE byte arrays!

References

[1] T. Arts and C. Benac Earle. Development of a verified Erlang program for resource locking. In *Proc. FMICS 2001, GMD Report No.91*, pages 109–122, 2001.

[2] 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.

[3] 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 2004.

[4] T. Arts and L. Fredlund. Trace analysis of erlang programs. *SIGPLAN Not.*, 37(12):18–24, 2002.

[5] T. Arts and J. Hughes. Quickcheck for erlang. In *Proceedings of the 2003 Erlang User Conference (EUC)*.

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

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

[8] J.F. Groote and A. Ponse. The syntax and semantics of μCRL. Technical Report CS-R9076, CWI, Amsterdam, 1990.

[9] G. Holzmann. *Design and Validation of Computer Protocols*. Prentice-Hall International, 1991.

[10] T. Lindahl and K. Sagonas. Detecting software defects in telecom applications through lightweight static analysis: A war story. In Chin Wei-Ngan, editor, *Programming Languages and Systems: Proceedings of the Second Asian Symposium (APLAS'04)*, volume 3302 of *LNCS*, pages 91–106. Springer, November 2004.

[11] A.G. Wouters. Manual for the μCRL toolset (version 2.07). Technical Report To appear, CWI, Amsterdam, 2001.

A. Appendix: Mutual Exclusion Monitor

```
-module(monMutex).
-export([init/1,stateChange/2]).

-include("state.hrl").
-include("process.hrl").

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

stateChange(State,MonState) ->
  try
    ResMap =
      calculateResources
      (State#state.processes),
    {ok, MonState}
  catch
    throw:Term -> {badmon, Term}
  end.

%%% Retrieves resources for processes
%%% in critical region
calculateResources(Procs) ->
  lists:foldl
  (fun (P,ResMap) ->
     case find_innermost(P#process.expr) of
       {pause, {_, inUse, [_,Resources,Type]}} ->
         addResources
         (P#process.pid,
          Resources,
          Type,
          ResMap);
        _ -> ResMap
     end
   end, [], Procs).

find_innermost({letexp,{Expr,_}}) ->
  find_innermost(Expr);
find_innermost(Expr) -> Expr.

%%% Returns a list of process locking a resource,
%%% or an exception if locking discipline violated

addResources(Pid,Resources,Type,Map) ->
  lists:foldl
  (fun (RName,RMap) ->
     addResource(Pid,RName,Type,RMap)
   end, Map, Resources).

addResource(Pid,Name,Type,[]) ->
  [{Type, Name, [Pid]}];
addResource(_,Name,excl,[{_,Name,_}|_]) ->
  throw(mutex);
addResource(_,Name,share,[{excl,Name,_}|_]) ->
  throw(mutex);
addResource(Pid,Name,share,[{share,Name,L}|Rest]) ->
  [{share,Name,[Pid|L]}|Rest];
addResource(Pid,Name,Type,[First|Rest]) ->
  [First|addResource(Pid,Name,Type,Rest)].
```

B. Appendix: `gen_server` component

```
-module(gen_server).
-export([start_link/4call/2,cast/2,reply/2,doStart/4]).

start_link({local, Name}, Module, Args, Options) ->
  Pid = spawn_link(?MODULE, doStart,
                   [Name, Module, Args, evOS:self()]),
  receive started -> {ok, Pid} end.

doStart(Name, Module, Args, ParentPid) ->
  {ok, State} = apply(Module, init, [Args]),
  register(Name,self()),
  ParentPid!started,
  loop(State, Module).

loop(State, Module) ->
  receive
    Msg ->
      case Msg of
        {call, Data, ReplyId} ->
          checkResult
          (Module,
           apply(Module, handle_call, [Data,ReplyId,State]),
           ReplyId);
        {cast, Data} ->
           checkResult
           (Module,
            apply(Module, handle_cast, [Data,State]),
            void);
        {'EXIT',From,Reason} ->
           checkResult
           (Module,
            apply(Module, handle_info, [Msg,State]),
            void)
      end
  end.

checkResult(Module,{reply,Data,State},ReplyId) ->
  reply(ReplyId,Data),
  loop(State, Module);
checkResult(Module,{stop,Reason,Data,StopState},ReplyId) ->
  reply(ReplyId,Data),
  terminating(Module,Reason,StopState);
checkResult(Module,{noreply,State},ReplyId) ->
  loop(State, Module);
checkResult(Module,{stop,Reason,StopState},ReplyId) ->
  terminating(Module,Reason,StopState).

reply({{callRef,CallPid},CallRef}, Reply) ->
  CallPid!{reply, {{callRef,CallPid},CallRef}, Reply},
  true.

terminating(Module, Reason, State) ->
 apply(Module, terminate, [Reason,State]), ok.

call(Server, Data) ->
  CallRef = references:getNewValue({callRef,self()}),
  Server!{call, Data, CallRef},
  receive
      {reply, CallRef, ReturnData} -> ReturnData
  end.

cast(Server, Data) -> Server!{cast, Data}, ok.
```