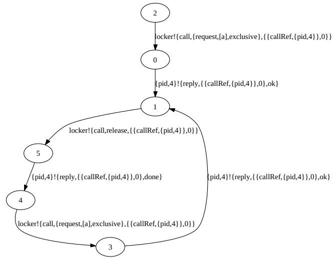
`McErlang` – a tool for model checking Erlang programs in Erlang

Lars-Åke Fredlund
Facultad de Informática, Universidad Politécnica de Madrid

Clara Benac Earle
Departamento de Informática, Universidad Carlos III de Madrid

What is model checking?

- Obtain an abstract representation – a **model** – of the program to check (often a labelled transition system)



- Provide a correctness property to check

$$\textbf{Always}\ \ (\neg \mathit{hasResource(Pid1)}\ \textbf{Or}\ \neg \mathit{hasResource(Pid2)})$$

- Using a model checking algorithm, prove that the model satisfies the correctness property (or a counterexample if not)

Why model checking?

- Push-button technology; in theory no manual proof steps
- Decent tools available: **SPIN**, **UPPAAL** (real-time systems), **Etomcrl** (Erlang), and many for hardware checking…

Why model checking?

- Push-button technology; in theory no manual proof steps
- Decent tools available: **SPIN**, **UPPAAL** (real-time systems), **Etomcrl** (Erlang), and many for hardware checking…
- Aha. So there are already many tools out there for model checking, and concretely **Etomcrl** is available for Erlang.
- Why do we need a new model checking tool?

Why model checking?

- Push-button technology; in theory no manual proof steps
- Decent tools available: **SPIN**, **UPPAAL** (real-time systems), **Etomcrl** (Erlang), and many for hardware checking…
- Aha. So there are already many tools out there for model checking, and concretely **Etomcrl** is available for Erlang.
- Why do we need a new model checking tool?
  - What language could be better for writing a model checker for Erlang than Erlang itself?
  - Writing a model checker means experimenting a lot with syntax and semantics – what language could be better than an untyped one?

Model Checking Programs

What is really needed to modelcheck an Erlang program against a correctness property?

- Compute transitions of a state s:

  `forall states` s' `and actions` α`:` $\quad s \xrightarrow{\alpha} s'$

- Compare program states for equality ($s \equiv s'$), to detect recurring states
- Inspect states or actions to determine whether they violate the correctness property being checked

Existing Tools for Erlang

- **Etomcrl** permits checking state equality, but the input language is rather restricted
- **QuickCheck** permits expressive programs, but cannot check equality between states (which is why it is a testing tool and not a model checking tool)
- What about the tools of Huch and Noll?
  And Hans Svenssons tool for generic servers?
- Is there some intermediate solution between **Etomcrl** and **QuickCheck**?

The `McErlang` approach to model checking

- So lets be *lazy*:
  we just execute Erlang functions, in Erlang, but try to access the combined system state as well
- The ideal solution would be to dig out the system state (queues, function contexts) for all processes from the Erlang runtime system

The `McErlang` approach to model checking

- So lets be *lazy*:
  we just execute Erlang functions, in Erlang, but try to access the combined system state as well
- The ideal solution would be to dig out the system state (queues, function contexts) for all processes from the Erlang runtime system
- Except we don't want to mess with the runtime system (written in C, complex, lots of other excuses...)
- Instead we develop a **new runtime system** for Erlang, in Erlang,
  with easy access to process state from Erlang,
  and execute the program to verify in the new runtime system

Erlang System State in New Runtime System

- A state is a tuple containing the processes, a map from atoms to pids (for `register`), and a set of pid tuples to implement process linking

$$\{\mathit{Processes}, \mathit{Register}, \mathit{Links}\}$$

- Each process is a tuple

$$\{\mathit{Status}, \mathit{Expr}, \mathit{Pid}, \mathit{Queue}, \mathit{CommQueue}, \mathit{Flags}\}$$

  - *Status* tells whether the process is runnable, has a receivable value, and so on
  - *Expr* is the expression to execute – a function application
  - *Pid*, *Queue* are standard
  - *Flags* controls some Erlang specific flags
  - *CommQueue* is used to implement distribution

Modifiying Code to use new Runtime

- We supply a new API to interact with the new runtime system: `evOS:send(Pid,Value)`, `evOS:link(Pid)`, `evOS:spawn(FunctionName,Arguments)`,...
- The new calls work on the new state structure instead of the old complex one
- For instance, Erlang processes are simulated only

Modifiying Code to use new Runtime

- We supply a new API to interact with the new runtime system: `evOS:send(Pid,Value)`, `evOS:link(Pid)`, `evOS:spawn(FunctionName,Arguments)`,...
- The new calls work on the new state structure instead of the old complex one
- For instance, Erlang processes are simulated only
- All this is fine, but the code still calls `link` instead of `evOS:link`

Modifiying Code to use new Runtime

- We supply a new API to interact with the new runtime system: `evOS:send(Pid,Value)`, `evOS:link(Pid)`, `evOS:spawn(FunctionName,Arguments)`,...
- The new calls work on the new state structure instead of the old complex one
- For instance, Erlang processes are simulated only
- All this is fine, but the code still calls `link` instead of `evOS:link`
- Solution: do a source-source transformation of the code, replacing calls to `link` with calls to `evOS:link` etc

Modifiying Code to use new Runtime

- We supply a new API to interact with the new runtime system: `evOS:send(Pid,Value)`, `evOS:link(Pid)`, `evOS:spawn(FunctionName,Arguments)`,...
- The new calls work on the new state structure instead of the old complex one
- For instance, Erlang processes are simulated only
- All this is fine, but the code still calls `link` instead of `evOS:link`
- Solution: do a source-source transformation of the code, replacing calls to `link` with calls to `evOS:link` etc
- Everything is fine except for `receive` statements which are handled specially; more on this soon...

Transition Semantics

So what are the states s, s' and actions α in transitions $s \xrightarrow{\alpha} s'$?

- In `McErlang` transitions occur between *stable states* of the Erlang program
- A stable runtime state is when all processes are in stable states
  - A process is in a stable state when it is waiting in a receive statement, or
  - It has just been spawned
- Actions are the side effects (upon other processes) that a process causes between stable states (a sequence of side effects)

Transition Semantics, implementation details

What happens when we start the interpreter with a call of a function `f(V1,...,Vn)` in a process P given a state s?

- Probably `f` causes some side effects during the call (by `evOS:spawn` etc)
- These side effects are immediately recorded in s (a new process datastructure is created – but not run yet)

Transition Semantics, implementation details

What happens when we start the interpreter with a call of a function `f(V1,...,Vn)` in a process P given a state s?

- Probably `f` causes some side effects during the call (by `evOS:spawn` etc)
- These side effects are immediately recorded in s (a new process datastructure is created – but not run yet)

How can the function call return?

Transition Semantics, implementation details

What happens when we start the interpreter with a call of a function `f(V1,...,Vn)` in a process P given a state s?

- Probably `f` causes some side effects during the call (by `evOS:spawn` etc)
- These side effects are immediately recorded in s (a new process datastructure is created – but not run yet)

How can the function call return?

- The call either returns a value, signalling that the process finished normally (the process is removed from s)
- Or the call generates an exeption, signalling that the process finished abnormally (we let other linked processes know by putting a special message in their queue)
- Or `f` tries to `receive` a value
- Or `f` doesn't return at all...

Handling Receive

- If `f` tries to receive a value the function is in a stable state; we are ready to possible start running another process for a while
- An interleaving semantics, big-step
- How do we detect trying to `receive`?
- By a second source-source transformation so that a function instead of calling receive returns a special tuple

```
{recv, {M, F, [V1,...,Vn]}}
```

  - `M:F` refers to a function for checking whether a receive is possible, and a continuation in case a receive happens
  - `[V1,...,Vn]` is a list of variables needed

Receive Example

```
f(Pid) ->
  receive
    hello -> Pid!hello, f(Pid);
    Other -> f(Pid)
  end.
```

Receive Example

```
f(Pid) ->
  receive
    hello -> Pid!hello, f(Pid);
    Other -> f(Pid)
  end.
```

becomes

```
f(Pid) -> {recv, {?MODULE, f_0, [Pid]}}.

f_0(hello,[Pid]) ->
  {true,
   fun (hello,[Pid]) ->
     evOS:send(Pid,hello), f(Pid)
   end};

f_0(Other,[Pid]) ->
  {true, fun (Other,[Pid]) -> f(Pid) end}.
```

Other Special Constructs

- A choice construct for directly expressing non-determininsm
- A form of let expression for handling receives that occur in another expression:

```
g(Pid,V) ->
        V*f(Pid).
```

Other Special Constructs

- A choice construct for directly expressing non-determininsm
- A form of let expression for handling receives that occur in another expression:

```
g(Pid,V) ->
    V*f(Pid).
```

becomes

```
f() ->
    {letexp, {f(Pid), {?MODULE, g_1, [V]}}}.

g_1(Result, [V]) ->
    V*Result.
```

Consequences of Transition Semantics

- Side effect free functions are executed by normal Erlang interpreter (quick, but maybe a dedicated model checker is quicker)

Consequences of Transition Semantics

- Side effect free functions are executed by normal Erlang interpreter (quick, but maybe a dedicated model checker is quicker)
- Side effect functions (sending, linking) are executed by new API functions, also in normal Erlang interpreter (quick, but destructive state updates are a bit slow in Erlang)

Consequences of Transition Semantics

- Side effect free functions are executed by normal Erlang interpreter (quick, but maybe a dedicated model checker is quicker)
- Side effect functions (sending, linking) are executed by new API functions, also in normal Erlang interpreter (quick, but destructive state updates are a bit slow in Erlang)
- When receive is called, the interpreter stops executing and we can schedule another processes

Consequences of Transition Semantics

- Side effect free functions are executed by normal Erlang interpreter (quick, but maybe a dedicated model checker is quicker)
- Side effect functions (sending, linking) are executed by new API functions, also in normal Erlang interpreter (quick, but destructive state updates are a bit slow in Erlang)
- When receive is called, the interpreter stops executing and we can schedule another processes
- We can easily inspect the global system state (actually implemented as a generic server process)

Consequences of Transition Semantics

- Side effect free functions are executed by normal Erlang interpreter (quick, but maybe a dedicated model checker is quicker)
- Side effect functions (sending, linking) are executed by new API functions, also in normal Erlang interpreter (quick, but destructive state updates are a bit slow in Erlang)
- When receive is called, the interpreter stops executing and we can schedule another processes
- We can easily inspect the global system state (actually implemented as a generic server process)
- We can check for state equality (normal Erlang equality "==")

Consequences of Transition Semantics

- Side effect free functions are executed by normal Erlang interpreter (quick, but maybe a dedicated model checker is quicker)
- Side effect functions (sending, linking) are executed by new API functions, also in normal Erlang interpreter (quick, but destructive state updates are a bit slow in Erlang)
- When receive is called, the interpreter stops executing and we can schedule another processes
- We can easily inspect the global system state (actually implemented as a generic server process)
- We can check for state equality (normal Erlang equality "==")
- We have easy and great power over the execution of a system: we can kill processes randomly, we can break communication links, . . .

Translator Tool

- There is a prototype translation tool to replace the calls to `link` with `evOS:link`, and `receive` with returning a value, etc
- We use the `syntax_tools` in the translation tool
- Handling Erlang variable bindings are a bit complex; it would be nice to have access to binding information directly in the `syntax_tools`
- Although we use basic Erlang communication primitives, OTP behaviours `gen_server`, `supervisor` are available as library functions defined using the standard basic communication primitives

Correctness Properties

- Ok, we can compute the state transition relation
- Next we need a language for expressing correctness properties

Correctness Properties

- Ok, we can compute the state transition relation
- Next we need a language for expressing correctness properties
- We pick Erlang of course (similarly to QuickCheck)
- A *monitor* is an Erlang function with two arguments: a new Erlang system state to check, and its own saved monitor state
- A monitor is properly an automaton, also has an internal state
- The monitor has full power to inspect the current state, and the actions leading to the current state
- If everything is ok with the Erlang state, the monitor returns a new monitor state; otherwise it signals an error

Monitor to detect deadlocks

```
-module(monDeadlock).
-export([init/1,stateChange/2]).
-include("state.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 ->
      error
  end.

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

Model Checking

- Next to do correctness checking we simply run an Erlang correctness "monitor" in lock-step with the Erlang program

$$\textit{Program} \mid\mid \textit{Monitor}$$

- That is, when the program takes a step the model checker offers the monitor to also take a step (with the new program state as argument), or halt signalling an error
- A simple depth-first state-generation algorithm is used to explore all the combined states of the program and monitor

Model Checking

- Next to do correctness checking we simply run an Erlang correctness "monitor" in lock-step with the Erlang program
  Program || *Monitor*

- That is, when the program takes a step the model checker offers the monitor to also take a step (with the new program state as argument), or halt signalling an error
- A simple depth-first state-generation algorithm is used to explore all the combined states of the program and monitor
- To detect recurring states we keep a hash table storing visited states

Model Checking

- Next to do correctness checking we simply run an Erlang correctness "monitor" in lock-step with the Erlang program

$$\mathit{Program} \mid\mid \mathit{Monitor}$$

- That is, when the program takes a step the model checker offers the monitor to also take a step (with the new program state as argument), or halt signalling an error
- A simple depth-first state-generation algorithm is used to explore all the combined states of the program and monitor
- To detect recurring states we keep a hash table storing visited states
- And we can also abstract (simplify or generalise) program states For example, replace values by types ($\texttt{2} \rightarrow \texttt{int}$)

Ensuring Finite Models

- To detect a property violation it may not matter if the model is finite
- On the hand, to prove correctness we need finite models
- New pid creation is one typically operation that use causes infinite models – here we choose *fresh* pids
- Similar handling of return tokens needed for generic server calls

Tool status and Conclusions

- Reasonable speed (we can certainly check the locker) – some 300000 states in 2 minutes
- Implementation not complex
- Programs to be checked can use complex data and complex side effect free functions without problems – we just execute them – no translation problem
- Nice to have a semantics of Erlang implemented in Erlang!

Near Future Work

- Perhaps have a less coarse transition semantics; break the execution for every side effect (e.g., spawns)
- Handle full temporal logic by implementing algorithms for checking Buchi automatons
- Add some model checking optimizations:
  reducing the storage needed for a state, and removing unnecessary states
- Handling a bigger piece of Erlang: monitors, nodes, . . .
- In a sense the approach is a really nice framework for doing model checking of other languages as well. We have a WS-CDL interpreter implemented in Erlang as well!