McErlang – a Model Checker for Erlang Programs

Lars-Åke Fredlund, Clara Benac Earle
Universidad Politécnica de Madrid

Hans Svensson, Chalmers

Facultad de Informática
Universidad Politécnica de Madrid

ProTest
property based testing

McErlang basics

- McErlang is useful for checking *concurrent software*, **not** for checking sequential software
- The Erlang runtime system for concurrency and communication is replaced with a new runtime system written in Erlang (`Pid!Value`, `spawn`, ... have been reimplemented)
- A concurrent program is checked under **all** possible schedulings
- McErlang is open source, available under a BSD license

McErlang In Practise: A Really Small Example

Two processes are spawned, the first starts an “echo” server that echoes received messages, and the second invokes the echo server:

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

start() ->
  spawn(fun() -> register(echo,self()), echo() end),
  spawn(fun() ->
           echo!{msg,self(),'hello␣world'},
           receive
             {echo,Msg} -> Msg
           end
        end).

echo() ->
  receive
    {msg,Client,Msg} ->
      Client!{echo,Msg}, echo()
  end.
```

Example under normal Erlang

Let's run the example under the standard Erlang runtime system:

```
> erlc example.erl
> erl
Erlang R13B02 (erts-5.7.3) ...

1> example:start().
<0.34.0>
2>
```

That worked fine. Let's try it under McErlang instead.

Example under McErlang

First have to recompile the module using the McErlang compiler:

```
> mcerl_compiler -sources example.erl -output_dir .
```

Example under McErlang

First have to recompile the module using the McErlang compiler:

```
> mcerl_compiler -sources example.erl -output_dir .
```

Then we run it:

```
> erl
Erlang (BEAM) emulator version 5.6.5 [source] [smp:2] ...

Eshell V5.6.5  (abort with ^G)
1> mce:apply(example,start,[]).
Starting McErlang model checker environment version 1.0 ...
...

Process ... exited because of error: badarg

Stack trace:
  mcerlang:resolvePid/2
  mcerlang:send/2
  ...
```

Investigating the Error

An error! Let's find out more using the McErlang debugger:

```
2> mce_erl_debugger:start(mce:result()).
Starting debugger with a stack trace; execution terminated
  user program raised an uncaught exception.

stack(@2)> showExecution().
0: process <node,1>:
 run function example:start([])
 spawn({#Fun<example.1.118053186>,[]},[]) --> <node,2>
 spawn({#Fun<example.2.76847815>,[]},[]) --> <node,3>
 process <node,1> was terminated
 process <node,1> died due to reason normal


1: process <node,3>:
  run #Fun<example.2.76847815>([])
  process <node,3> died due to reason badarg
```

Facultad de Informática Universidad Politécnica de Madrid

ProTest property based testing

Error Cause

- Apparently in one program run the second process spawned (the one calling the echo server) was run before the echo server itself:

  ```
  run #Fun<example.2.76847815>([])
  ```

- Then upon trying to send a message

  ```
  echo!{msg,self(),'hello␣world'}
  ```

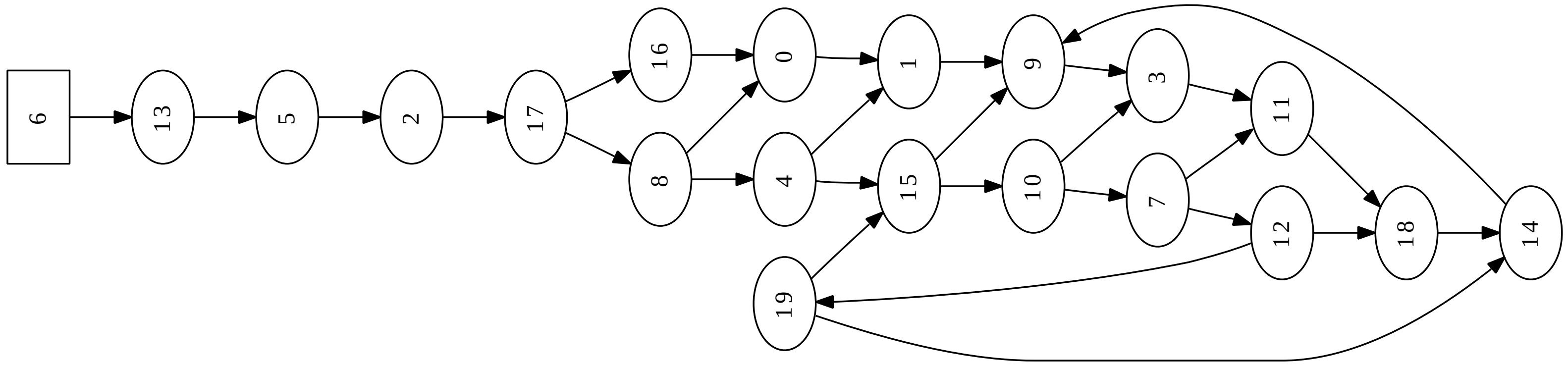
  the `echo` name was obviously not registered, so the program crashed

Presentation Outline

- What is model checking & a brief comparison with testing
- McErlang basics
- Integration with QuickCheck
- McErlang in practise: installing and usage
- Working with a larger example: a lift control system

Model Checking: Basics

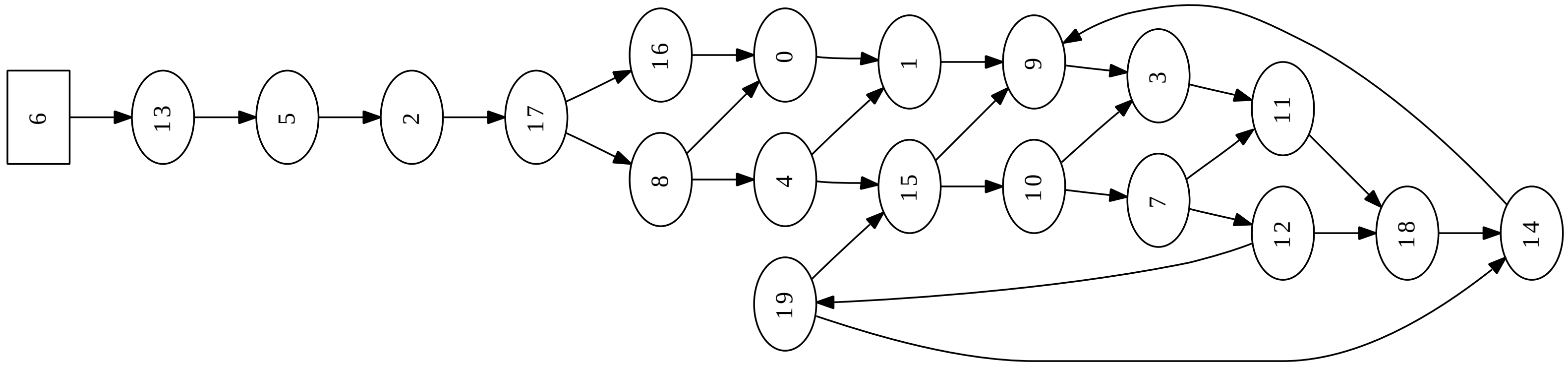
- **Construct** an abstract **model** of the behaviour of the program, usually a finite state transition graph



  - A node represents a **Program state** ($x = 0, y = 3$)
  - **Graph edges** represent computation steps from one program state to another

Model Checking: Basics

- **Construct** an abstract **model** of the behaviour of the program, usually a finite state transition graph



  - A node represents a **Program state** $(x = 0, y = 3)$
  - **Graph edges** represent computation steps from one program state to another

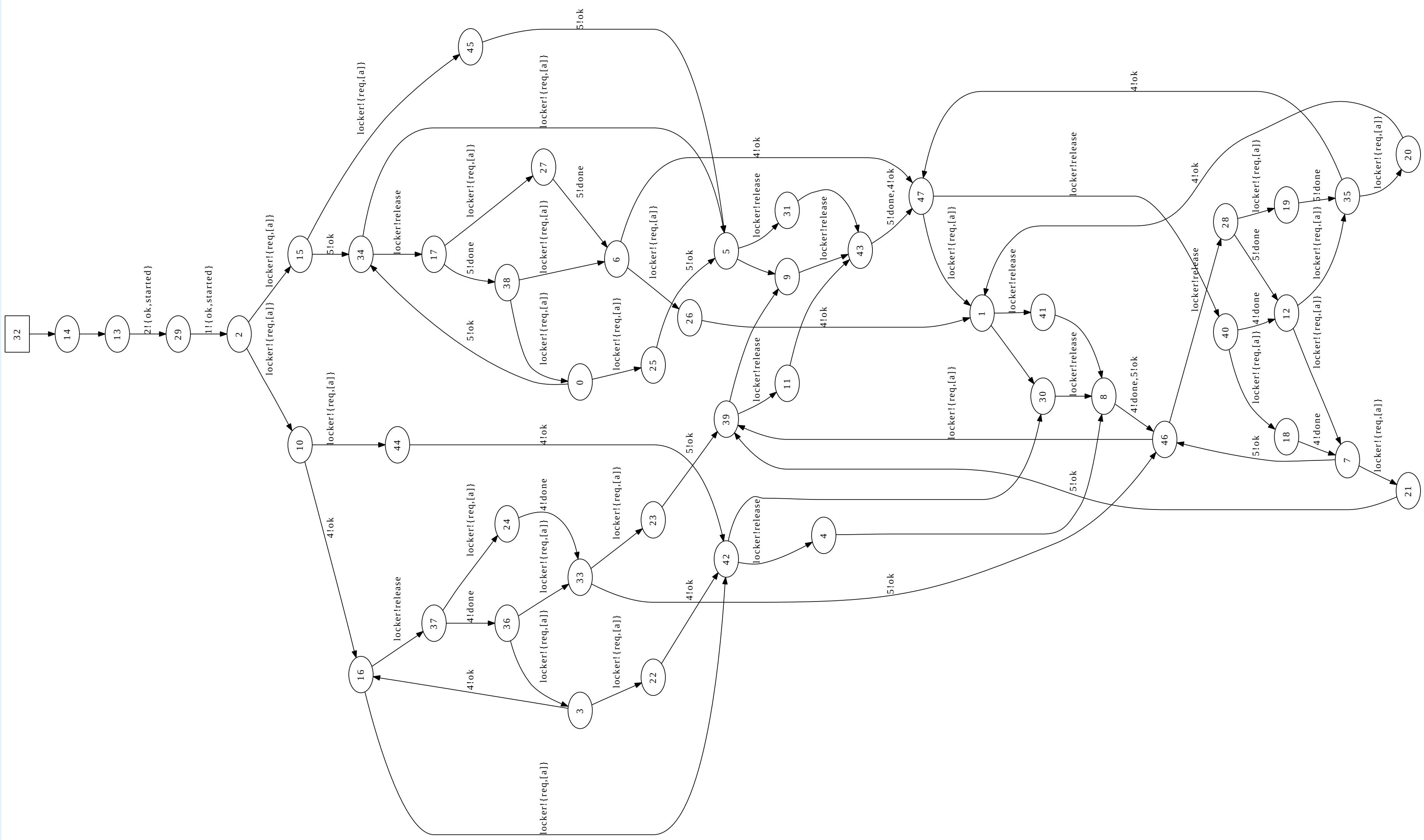
- **Check** the abstract model against some description of desirable/undesirable model properties usually specified in a **temporal logic**: *Always* $x \geq 0$

Testing Concurrent Programs

Why is (random) testing of concurrent programs difficult?

Testing Concurrent Programs

Consider the state space of a small program:

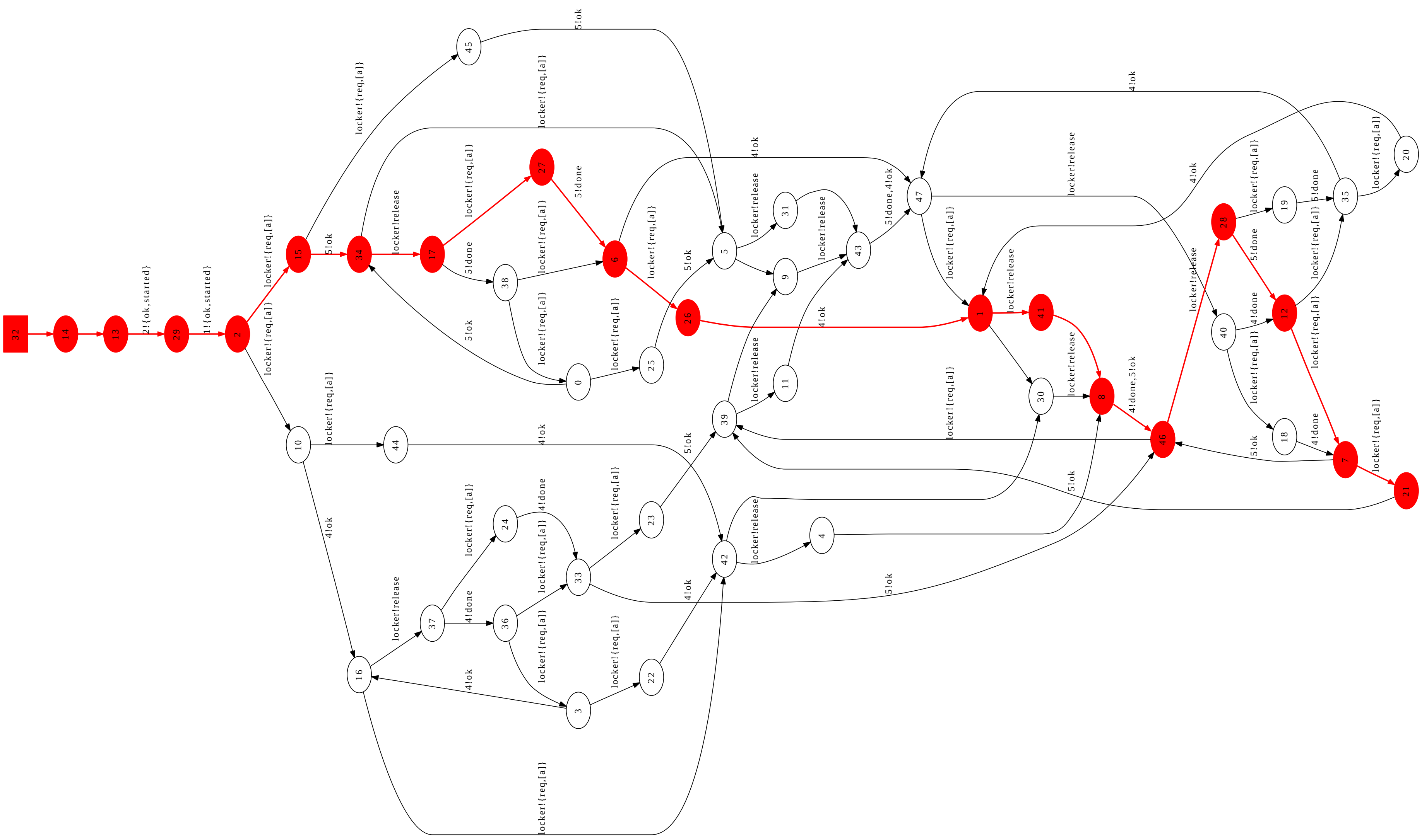


Facultad de Informática
Universidad Politécnica de Madrid

ProTest
property based testing

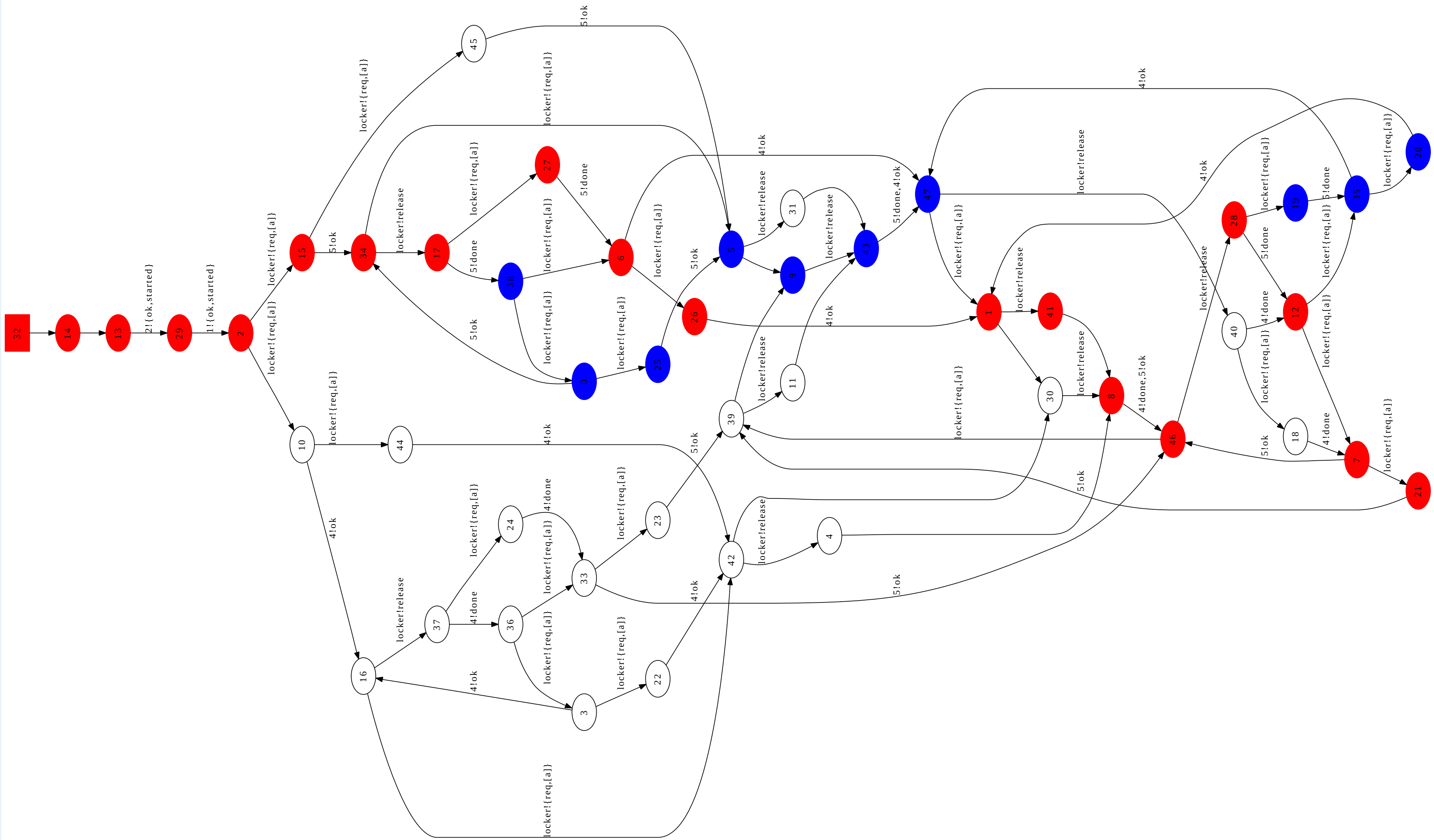
Testing Concurrent Programs

Random testing explores **one** path through the program:



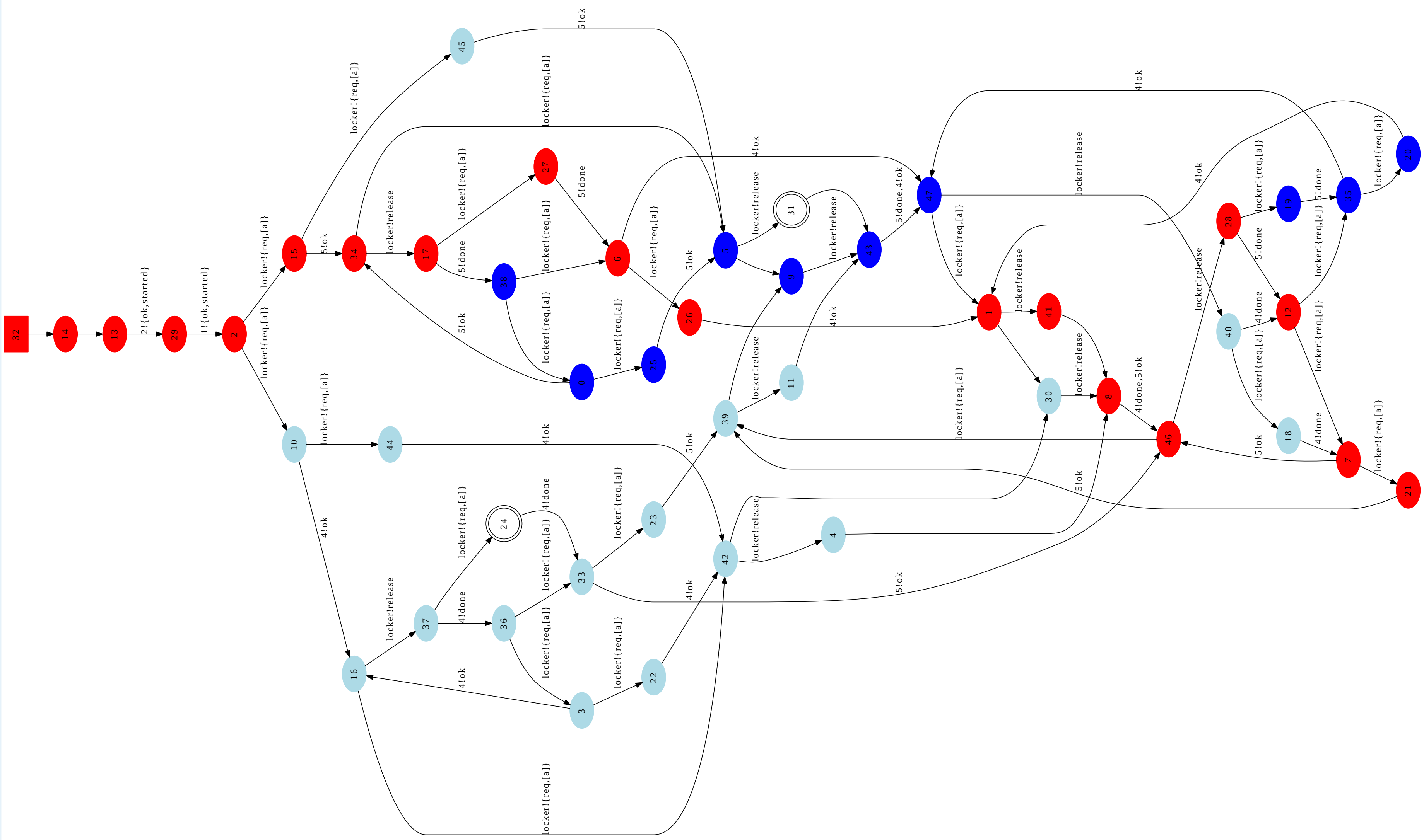
Testing Concurrent Programs

With repeated tests the coverage improves:



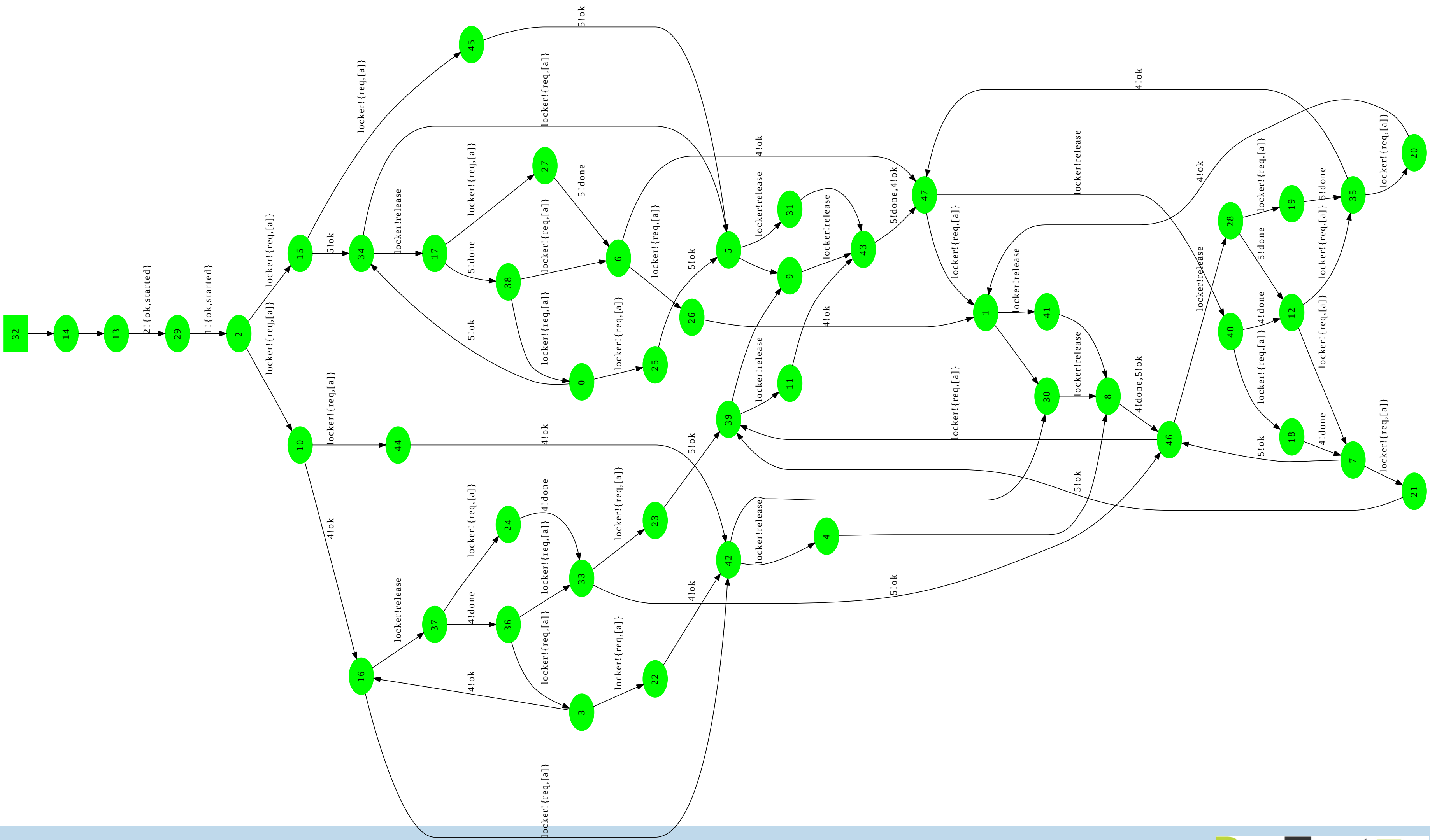
Testing Concurrent Programs

A lot of testing later (note the states not visited):



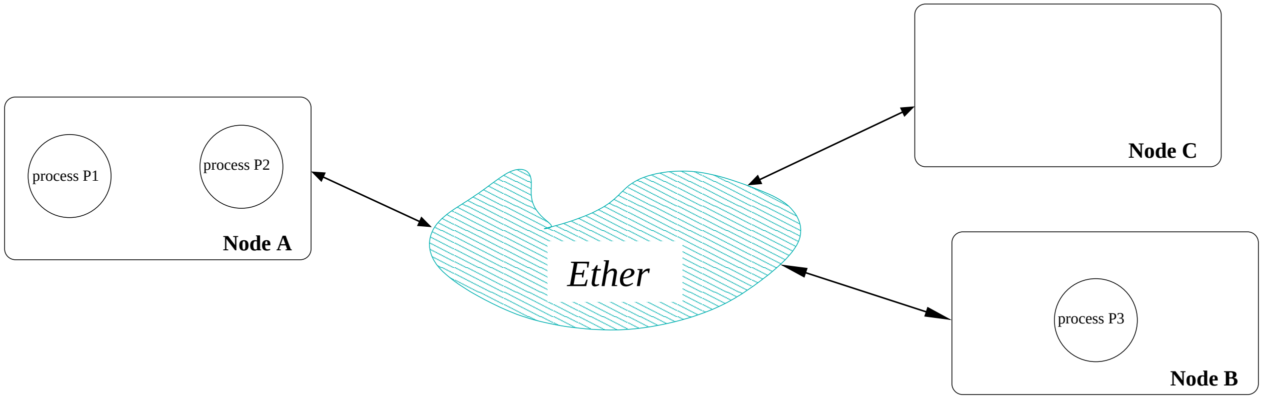
Testing Concurrent Programs

Model checking can guarantee that all states are visited, without revisiting states



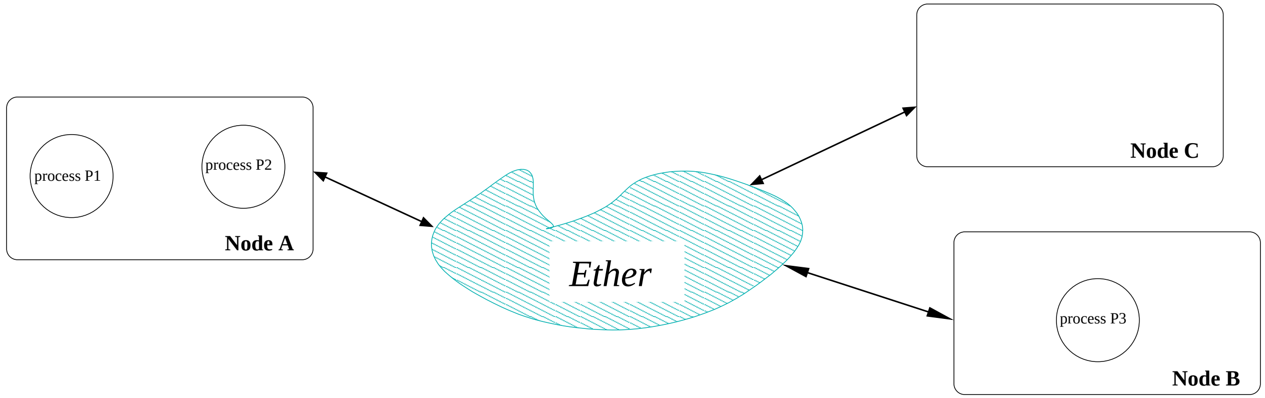
Step-by-step execution of Erlang Programs

- To be able to visit **all** the states of an Erlang program we need the capability to take a **snapshot** of the Erlang system
  - A **snapshot/program state** is: the contents of all process mailboxes, the state of all running processes, messages in transit (the ether), all nodes, monitors, . . .



Step-by-step execution of Erlang Programs

- To be able to visit **all** the states of an Erlang program we need the capability to take a **snapshot** of the Erlang system
  - A **snapshot**/**program state** is: the contents of all process mailboxes, the state of all running processes, messages in transit (the ether), all nodes, monitors, . . .



- Save the snapshot to memory and forget about it for a while
- Later continue the execution from the snapshot

Fundamental Difficulties of Model Checking

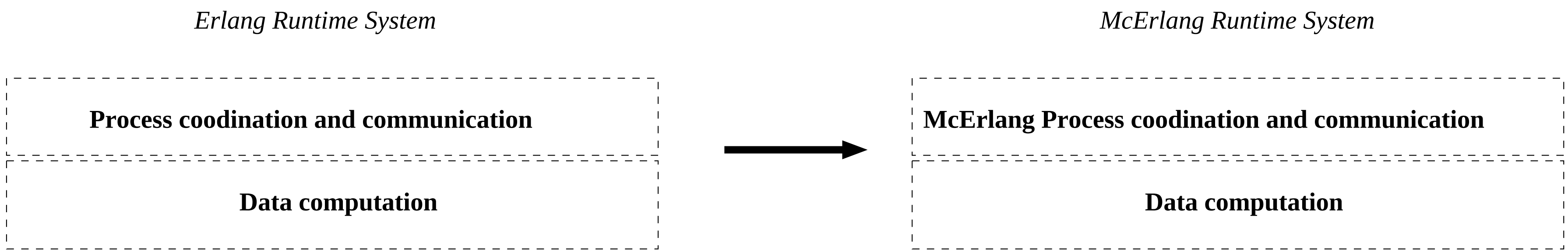
- Too many states (not enough memory to save all snapshots)
- Checking all states takes too much time
- We have to a snapshot of things outside of Erlang
  (hard drives due to disk writes and reads,. . .)

The McErlang approach to model checking

- The lazy solution: just execute the Erlang program to verify in the normal Erlang interpreter
- And extract the system state (processes, queues, function contexts) from the Erlang runtime system

The McErlang approach to model checking

- The lazy solution: just execute the Erlang program to verify in the normal Erlang interpreter
- And extract the system state (processes, queues, function contexts) from the Erlang runtime system
- Too messy! We have developed a **new runtime system** for the process part, and still use the old runtime system to execute code with no side effects



Adapting code for the new runtime environment

Erlang code must be “compiled” by the McErlang “compiler” to run under the new runtime system:

- API change example: call `mcerlang:`**`spawn`** instead of `erlang:`**`spawn`**

Adapting code for the new runtime environment

Erlang code must be “compiled” by the McErlang “compiler” to run under the new runtime system:

- API change example: call `mcerlang:spawn` instead of `erlang:spawn`

- These transformations are implemented on HiPE Core Erlang code (a compiler intermediate language)

Full Erlang Supported?

- Virtually the full core language supported:
  - Processes, nodes, links, all data types
  - Higher-order functions

  Many libraries at least partly supported:
  - supervisor, gen_server, gen_fsm, ets
  - **Not supported:** gen_tcp, . . .

Full Erlang supported?

No real-time model checking implementation yet

```
receive
  X -> X
  after 20 -> ...
end
```

behaves the same as

```
receive
  X -> X
  after 20000 -> ...
end
```

Full Erlang supported?

No real-time model checking implementation yet

```
receive
  X -> X
  after 20 -> ...
end
```

behaves the same as

```
receive
  X -> X
  after 20000 -> ...
end

but is different from
receive
  X -> X
end
```

Extensions to Erlang in McErlang

- Non-determinacy:

```
mce_erl:choice
  ([fun () -> Pid!hi end,
    fun () -> Pid!hola end]).
```

  sends either `hi` or `hola` to `Pid` but not both

Extensions to Erlang in McErlang

- Non-determinacy:

```
mce_erl:choice
  ([fun () -> Pid!hi end,
    fun () -> Pid!hola end]).
```

sends either `hi` or `hola` to `Pid` but not both

- Convenience:

```
mcerlang:spawn
  (new_node,
   fun () -> Pid!hello_world end)
```

The node `new_node` is created if it does not exist

Compiling/preparing code for running under McErlang

- *All* source code modules of a project must be provided to the McErlang compiler
- *Some* OTP behaviours/libraries are automatically included at compile time
- Example: `mcerl_compile -sources *.erl`
- The translation is controlled by the `funinfo.txt` file (can be customised)
- The result of the translation is a set of `beam` files (and Core Erlang code for the translated modules)

Controlling Translation

- The file `funinfo.txt` controls the remapping of functions and describes side effects:

```
[
 {gen_server,[{translated_to,mce_erl_gen_server}]},
 {supervisor,[{translated_to,mce_erl_supervisor}]},
 {gen_fsm,[{translated_to,mce_erl_gen_fsm}]},
 {erlang,[{rcv,false}]},
 {{erlang,spawn,4},
     [rcv,
      {translated_to,{mcerlang,spawn}}]},
 {{erlang,send,2},[{translated_to,{mcerlang,send}}]},
 ...
]
```

- A verification project can use its own `funinfo.txt`

Choice of Libraries

- McErlang has tailored versions of some libraries: `supervisor`, `gen_server`, `gen_fsm`, `gen_event`, `lists`, `ets`, . . . which are automatically included
- It may be possible to use the standard OTP libraries instead

Running programs under McErlang

- Starting McErlang:

```
mce:start
  (#mce_opts{program={Module,Fun,Args},
             algorithm={Module,InitArgs},
             monitor={Module,InitArgs})
```

- Example: starting the `Echo` program

```
mce:start
  (#mce_opts{program={example,start,[]},
             algorithm={mce_alg_safety,void},
             monitor={mce_mon_test,void})
```

- The result of a model checking run can be retrieved using `mce:result()` (a program trace leading to the bug)

McErlang runtime options

More #`mce_opts{}` record options:

- `shortest = true() | false()`
  Compute the shortest path to failure? (false)
- `fail_on_exit = true() | false()`
  Stop a model checking run if a process terminates abnormally due to an uncaught exception (true)
- `time_limit = seconds`
  Halts verification after reaching a time limit
- And many more . . .

Algorithms

An algorithm determines the particular state space exploration strategy used by McErlang:

- `mce_alg_simulation`
  Implements a basic simulation algorithm –
  following a single execution path

- `mce_alg_safety`
  Checks the specified monitor on *all* program states

- `mce_alg_combine`
  Combines simulation and model checking to reduce state space

What to check: Correctness Properties

Ok, we can run programs under the McErlang runtime system.
Next we need a language for expressing correctness properties:

What to check: Correctness Properties

Ok, we can run programs under the McErlang runtime system.
Next we need a language for expressing correctness properties:

- We pick Erlang of course!

  A *safety monitor* is an user function with three arguments:

```
stateChange(State, MonitorState, Action) ->
  ...
  {ok, NewMonitorState}.
```

What to check: Correctness Properties

Ok, we can run programs under the McErlang runtime system.
Next we need a language for expressing correctness properties:

- We pick Erlang of course!

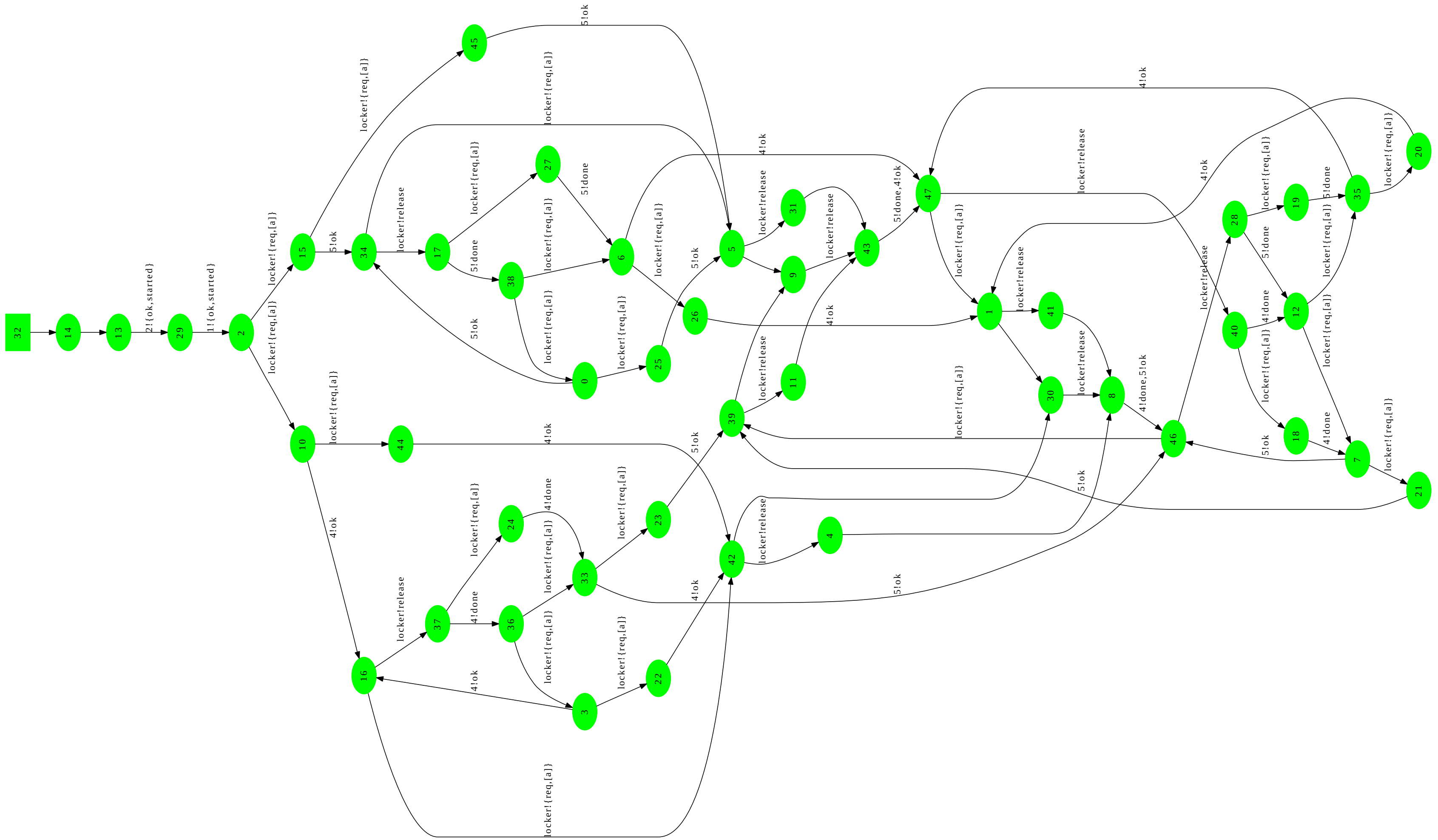
  A *safety monitor* is an user function with three arguments:

  ```
  stateChange(State, MonitorState, Action) ->
    ...
    {ok, NewMonitorState}.
  ```

- A program is checked by running it in lock-step with a monitor
- The monitor can inspect the current state, and the side effects (actions) in the last computation step
- The monitor either returns a new monitor state `{ok,NewMonitorState}`, or signals an error

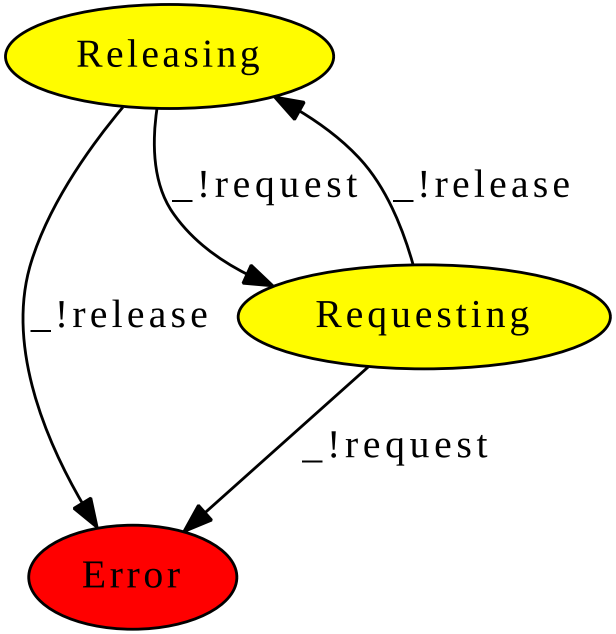
Safety Monitors

- Safety Monitors check that *nothing bad ever happens*
- They must be checked in *all* the states of the program:



A monitor example

- We want to implement a monitor to check that a program alternates between sending `request` and `release`
- As an automaton:



A monitor example implemented in Erlang

```
-module(req_rel_alternate).
-export([init/1,stateChange/3,monitorType/0]).
-behaviour(mce_behav_monitor).

monitorType() -> safety.
init(_) -> {ok,request}.

stateChange(ProgramState,request,Action) ->
  case get_action(Action) of
    {ok,request} -> {ok,release};
    {ok,release} -> not_alternating
    _ -> {ok,request}
  end; ...

get_action(Action) ->
  case mce_erl_actions:is_send(Action) of
    true -> {ok,mce_erl_actions:get_send_msg(Action)};
    false -> no_action
  end.
```

What can monitors observe?

- Program **actions** such as sending or receiving a message
- Program **state** such as the contents of process mailboxes, names of registered processes
- The values of some program variables
(can be tricky)
- Programs can be instrumented with special *probe actions* that are easy to detect in monitors
(e.g. calling `mce_erl:probe(requesting)`)
- Programs can be instrumented with special *probe states*, which are *persistent* (actions are transient)
(e.g. calling `mce_erl:probe_state(have_requested)`)

Some Predefined Monitors

- `mce_mon_deadlock`
  Checks that there is at least one non-deadlocked process
- `mce_mon_queue`
  Checks that all queues contain at most `MaxQueueSize` elements.

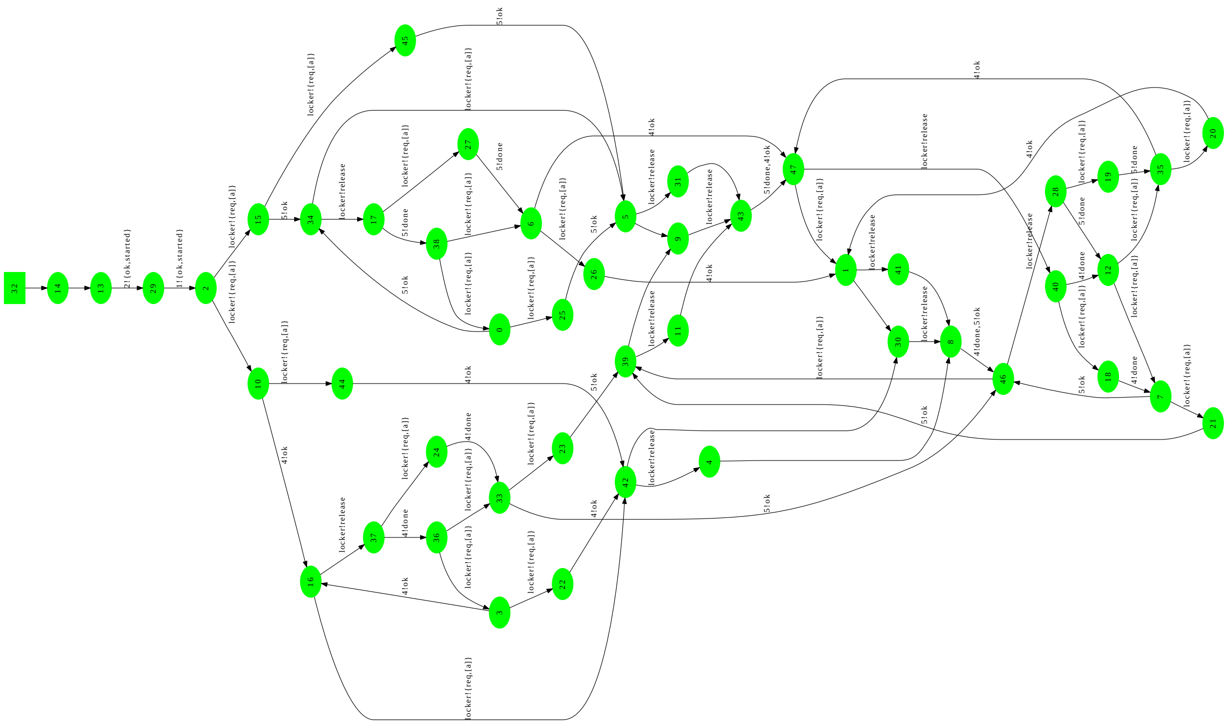
The McErlang Debugger

- There is a rudimentary debugger for examining counter examples
- After a failed model checking run, start the debugger on the counterexample using:

  ```
  mce_erl_debugger:start(mce:result()).
  ```

Things that can go wrong

- McErlang runs out of memory – too many states



- McErlang takes too long

- Why? Program uses timers, counters, random values, ... or is simply too complex

What can be done

Partial verification – explore part of the state space

What can be done

Partial verification – explore part of the state space

- Use a (lossy) bounded size state table:

```
#mce_opts
  {...,table={mce_table_bitHash,Size}, ...}
```

- Use a bounded stack

```
#mce_opts
  {...,stack={mce_stack_bounded,Size}, ...}
```

- Try a more random state space exploration algorithm (`mce_alg_safety_rnd`)
- Put a bound on the verification time
- Check smaller examples (a set of test cases)

Integration of QuickCheck and McErlang

- Permits to check QuickCheck properties using McErlang to execute the Erlang code
- For normal properties, `eqc_statem:commands` or `eqc_statem:parallel_commands`

Integration of QuickCheck and McErlang

- Permits to check QuickCheck properties using McErlang to execute the Erlang code
- For normal properties, `eqc_statem:commands` or `eqc_statem:parallel_commands`
- QuickCheck Choices:
  - Use the normal, pretty deterministic, Erlang scheduler to execute programs

Integration of QuickCheck and McErlang

- Permits to check QuickCheck properties using McErlang to execute the Erlang code
- For normal properties, `eqc_statem:commands` or `eqc_statem:parallel_commands`
- QuickCheck Choices:
  - Use the normal, pretty deterministic, Erlang scheduler to execute programs
  - Use Pulse to check program under a more random scheduler

Integration of QuickCheck and McErlang

- Permits to check QuickCheck properties using McErlang to execute the Erlang code
- For normal properties, `eqc_statem:commands` or `eqc_statem:parallel_commands`
- QuickCheck Choices:
  - Use the normal, pretty deterministic, Erlang scheduler to execute programs
  - Use Pulse to check program under a more random scheduler
  - Use McErlang to check program under potentially **all** schedulings

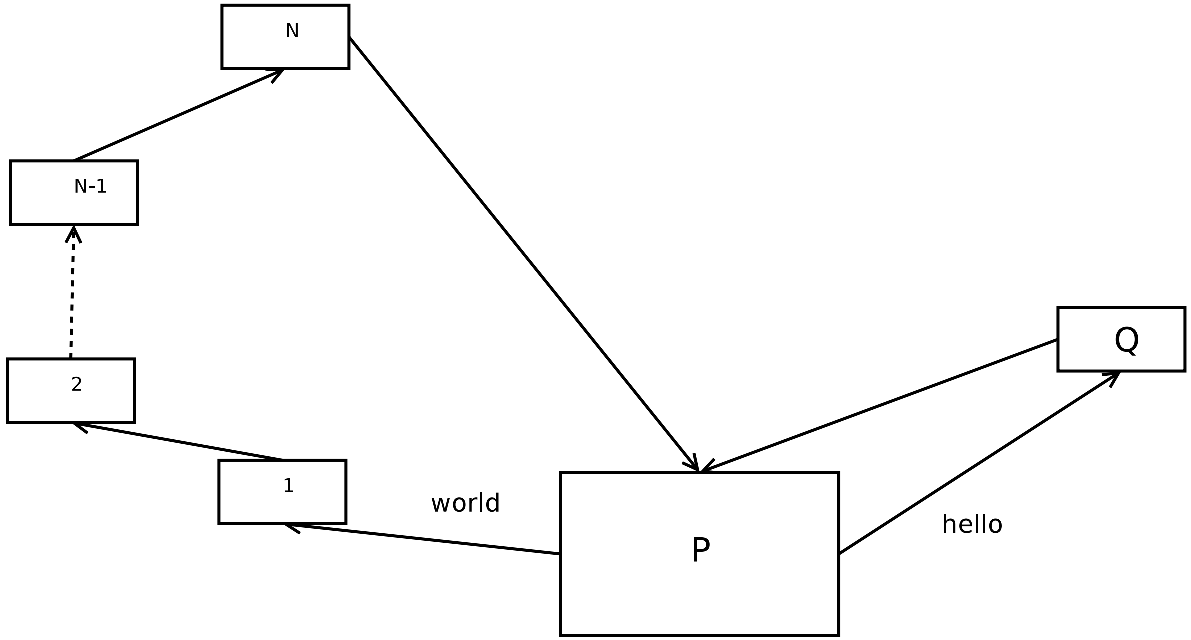
Integration of QuickCheck and McErlang

- Permits to check QuickCheck properties using McErlang to execute the Erlang code
- For normal properties, `eqc_statem:commands` or `eqc_statem:parallel_commands`
- QuickCheck Choices:
  - Use the normal, pretty deterministic, Erlang scheduler to execute programs
  - Use Pulse to check program under a more random scheduler
  - Use McErlang to check program under potentially **all** schedulings
- McErlang interface currently distributed with QuickCheck

QuickCheck integration: example

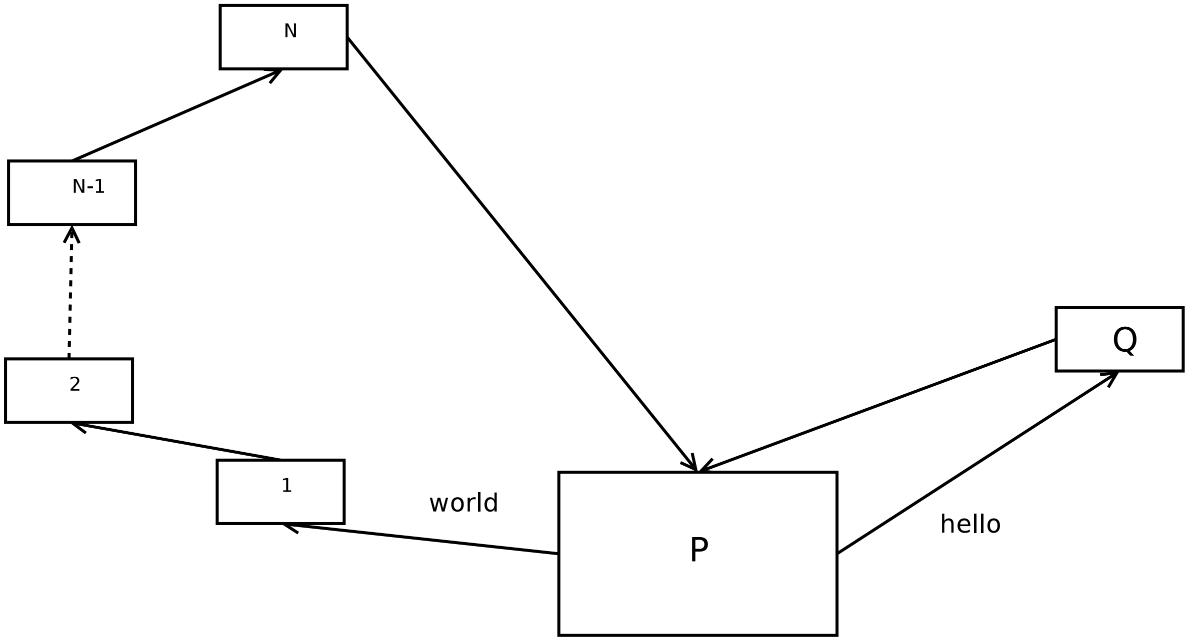
```
proxy(Pid) ->
  spawn(fun() ->
    receive  Msg ->  Pid ! Msg end
        end).

proxy(0, Pid) -> Pid;
proxy(N, Pid) ->
  Proxy = proxy(N-1, Pid),
  proxy(Proxy).

world_hello(N) ->
  C = self(),
  B1 = proxy(1, C),
  B2 = proxy(N, C),
  _A = spawn(fun() -> B1!hello, B2!world end),
  Msg1 = receive Msg1_ ->  Msg1_  end,
  Msg2 = receive Msg2_ ->  Msg2_  end,
  {Msg1, Msg2}.
```

As a graph



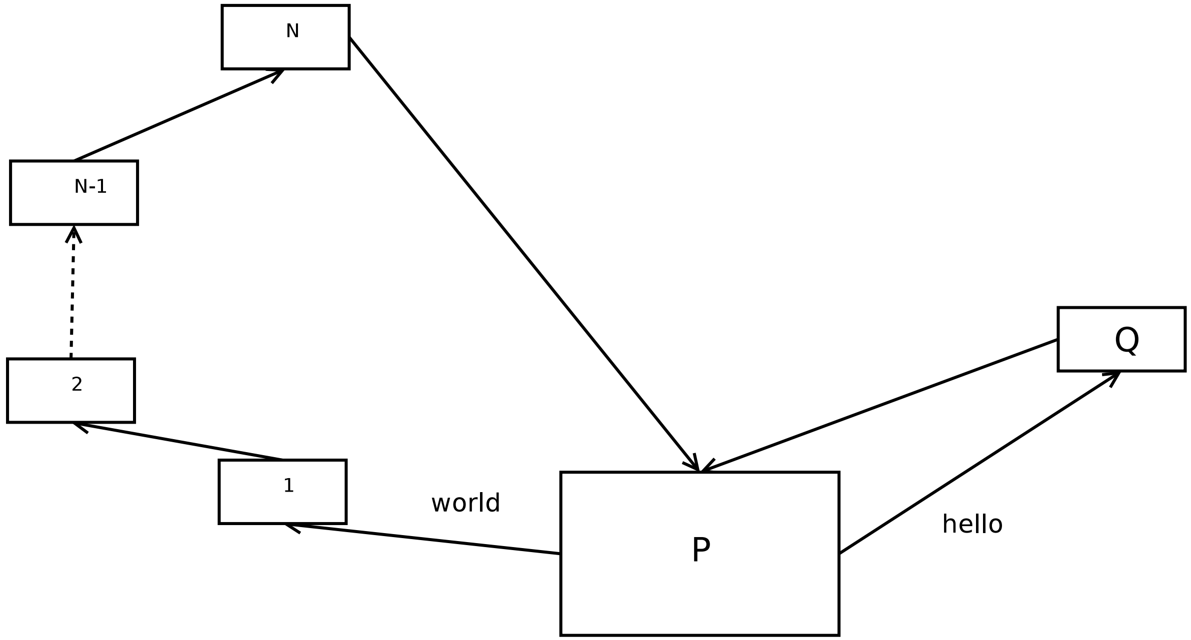
- P sends `world` to 1, who forwards to 2, . . . , to N-1, to N, which eventually sends it back to P
- And P sends `hello` to Q, which directly sends it back to P

As a graph



- P sends `world` to 1, who forwards to 2, . . . , to N-1, to N, which eventually sends it back to P
- And P sends `hello` to Q, which directly sends it back to P
- In what order is `world` and `hello` received at P?

As a graph



- P sends `world` to 1, who forwards to 2, . . . , to N-1, to N, which eventually sends it back to P
- And P sends `hello` to Q, which directly sends it back to P
- In what order is `world` and `hello` received at P?
- If N is sufficiently large, almost always `hello` is received first at P, and then `world`

Checking “correct reception” in QuickCheck

```
-module(proxy_eqc).

...

prop_world_hello() ->
   ?FORALL(
      1,
      1,
      world_hello(100) == {hello,world}
     ).
```

Checking “correct reception” in QuickCheck

```
-module(proxy_eqc).

...

prop_world_hello() ->
   ?FORALL(
      1,
      1,
      world_hello(100) == {hello,world}
     ).
```

Checking:

```
> erl
Erlang R14B03 ...

1> c(proxy_eqc).
{ok,proxy_eqc}
2> eqc:quickcheck(proxy_eqc:prop_world_hello()).
```

Checking "correct reception" in QuickCheck and McErlang

```
-module(proxy_mce).

-include_lib("eqc_mcerlang/include/eqc_mcerlang.hrl").
...

prop_world_hello_mce() ->
  mce_app:set_verification_algorithm(mce_alg_safety_rnd),
  ?FORALL(
     1,
     1,
     ?MCERLANG(
        [?MODULE],
        Res,
        world_hello(100),
        Res == {hello,world}
       )).
```

Checking using QuickCheck+McErlang

```
> erl
Erlang R14B03 ...

1> mce_app:start().
ok
2> mce_erl_compile:file("proxy_mce.erl",[{outdir,"."}]).
{ok,[proxy_mce]}
3> eqc:quickcheck(proxy_mce:prop_world_hello_mce()).
```

Integrating with QuickCheck testing: options

When programs are too complex to fully verify, model checking becomes a form of controlled testing:

- The amount of memory and time available to verify a program can be controlled (a verification attempt can be *inconclusive*)
- Randomized (wrt. state space exploration order) verification algorithms are available (thus repeating a verification run can explore new parts of the state space)
- Randomized state storage data structures are available (Holzmann's bitspace algorithms)

McErlang in Practise: downloading

- Web page:
  https://babel.ls.fi.upm.es/trac/McErlang/

- Use subversion to check out the McErlang sources:

  ```
  svn checkout \
  https://babel.ls.fi.upm.es/svn/McErlang/trunk \
  McErlang
  ```

- Precompiled versions are avaible too

Installing and Documentation

- We use Ubuntu – McErlang doesn't work well under Windows
- Compile McErlang:

```
cd McErlang; ./configure; make release
```

- Installing McErlang among normal Erlang libraries:

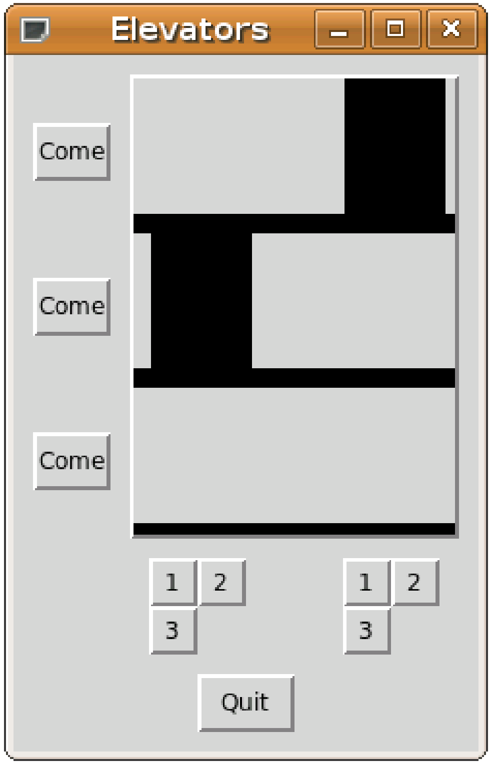
```
> cd release/McErl*
> erl
Erlang R14B03 ...

1> mcerlang_install:install().
```

- Read the manuals:

```
acroread doc/tutorial/tutorial.pdf
acroread doc/userManual/userManual.pdf
```

McErlang in practise: The Elevator Example

- We study the control software for a set of elevators

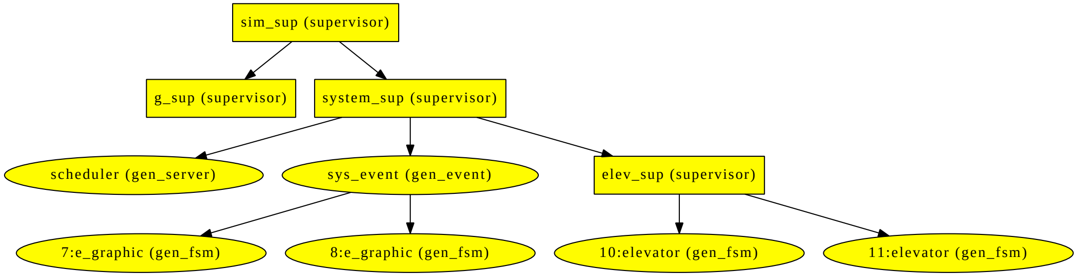


- Used to be part of an Erlang/OTP training course from Ericsson

The Elevator Example

Example complexity:

- Static complexity: around 1670 lines of code
- Dynamic complexity: around 10 processes (for two elevators)
- Uses quite a few libraries: `lists`, `gen_event`, `gen_fsm`, `supervisor`, `timer`, `gs`, `application`



Running the elevator under McErlang

- First we just try to run it under the McErlang runtime system (forgetting about model checking for a while)
- This will test the system under a less deterministic scheduler than the normal Erlang scheduler

Running the elevator under McErlang

- First we just try to run it under the McErlang runtime system (forgetting about model checking for a while)
- This will test the system under a less deterministic scheduler than the normal Erlang scheduler
- Executing:

```
mce:start
  (#mce_opts
   {program={sim_sup,start_link,[1,3,2]},
    sim_external_world=true,
    algorithm=mce_alg_simulation}).
```

Running the elevator under McErlang

- First we just try to run it under the McErlang runtime system (forgetting about model checking for a while)
- This will test the system under a less deterministic scheduler than the normal Erlang scheduler
- Executing:

```
mce:start
  (#mce_opts
   {program={sim_sup,start_link,[1,3,2]},
    sim_external_world=true,
    algorithm=mce_alg_simulation}).
```

- Seems to work...

Model checking the elevator under McErlang

Model checking is a bit more complicated:

Model checking the elevator under McErlang

Model checking is a bit more complicated:

- The `gs` graphics will not make sense when model checking $\Rightarrow$
  We shut it off in model checking mode

Model checking the elevator under McErlang

Model checking is a bit more complicated:

- The `gs` graphics will not make sense when model checking $\Rightarrow$

  We shut it off in model checking mode

- The example is very geared to smooth graphical display $\Rightarrow$

  We modify the program to only have three (3) intermediate points between elevator floors (normally 20)

Model checking the elevator under McErlang

Model checking is a bit more complicated:

- The `gs` graphics will not make sense when model checking $\Rightarrow$

  We shut it off in model checking mode

- The example is very geared to smooth graphical display $\Rightarrow$

  We modify the program to only have three (3) intermediate points between elevator floors (normally 20)

- The program contain timers (for moving the elevator) $\Rightarrow$

  We assume that the program is *infinitely fast* compared to the timers: timer only release when no program action is possible

Model checking the elevator under McErlang

Model checking is a bit more complicated:

- The `gs` graphics will not make sense when model checking ⇒

  We shut it off in model checking mode

- The example is very geared to smooth graphical display ⇒

  We modify the program to only have three (3) intermediate points between elevator floors (normally 20)

- The program contain timers (for moving the elevator) ⇒

  We assume that the program is *infinitely fast* compared to the timers: timer only release when no program action is possible

- In total, about 15 lines of code had to be changed to enable model checking – **not too bad!**

Scenarios

- Instead of specifying one big scenario with a really big state space, we specify a number of smaller scenarios
- Paremeters:

  Number of elevators,

  Number of floors,

  Commands:

```
[{scheduler,f_button_pressed,[1]},
 {scheduler,e_button_pressed,[2,1]},
 {scheduler,f_button_pressed,[1]}]
```

- QuickCheck can be used to generate a set of scenarios

Correctness Properties for the Elevator System

What are good correctness properties for the Elevator system?

Correctness Properties for the Elevator System

What are good correctness properties for the Elevator system?

- *No runtime exceptions*

Correctness Properties for the Elevator System

What are good correctness properties for the Elevator system?

- *No runtime exceptions*
- *An elevator only stops at a floor after receiving an order to go to that floor*

Correctness Properties for the Elevator System

What are good correctness properties for the Elevator system?

- *No runtime exceptions*
- *An elevator only stops at a floor after receiving an order to go to that floor*
- *If there is a request to go to some floor, eventually some elevator will stop there*
- ...

Checking absence of exceptions

Checking absence of exceptions

- *No runtime exceptions*

Checking absence of exceptions

- *No runtime exceptions*
- Checking:

```
> mce:start(#mce_opts
   {program={run_scenario,run_scenario,
             [2,2,[{scheduler,f_button_pressed,[1]}]]},
    algorithm={mce_alg_safety,void}}).
```

Checking absence of exceptions

- *No runtime exceptions*
- Checking:

```
> mce:start(#mce_opts
   {program={run_scenario,run_scenario,
             [2,2,[{scheduler,f_button_pressed,[1]}]]},
    algorithm={mce_alg_safety,void}}).
```

- Result:

```
*** User code generated error:
exception error due to reason {badmatch,[]}
Stack trace:
  scheduler:add_to_a_stoplist near line 344/3
  scheduler:handle_cast/2
  ...
```

Checking absence of exceptions

- *No runtime exceptions*
- Checking:

```
> mce:start(#mce_opts
   {program={run_scenario,run_scenario,
             [2,2,[{scheduler,f_button_pressed,[1]}]]},
    algorithm={mce_alg_safety,void}}).
```

- Result:

```
*** User code generated error:
exception error due to reason {badmatch,[]}
Stack trace:
  scheduler:add_to_a_stoplist near line 344/3
  scheduler:handle_cast/2
  ...
```

- **Bug** - the system received the "press button"-command before it had been initialised

"Hiding the bug"

- Instead of fixing the bug we hide it by only sending commands when the system has started by enabling the option `is_infinitely_fast=true`

- Checking:

```
> mce:start(#mce_opts
   {program={run_scenario,run_scenario,
              [2,2,[{scheduler,f_button_pressed,[1]}]]},
    is_infinitely_fast=true,
    algorithm={mce_alg_safety,void}}).
```

Checking Safety Properties

Checking Safety Properties

- *An elevator only stops at a floor after receiving an order to go to that floor*

- We use a *safety monitor* to implement the property. Remember:
    - A safety monitor runs in parallel (lock-step) with the program

Checking Safety Properties

- *An elevator only stops at a floor after receiving an order to go to that floor*

- We use a *safety monitor* to implement the property. Remember:
  - A safety monitor runs in parallel (lock-step) with the program
  - A monitor has an internal state, which can be updated when the program does a *significant* action (or something happens – *a button press*)

Checking Safety Properties

- *An elevator only stops at a floor after receiving an order to go to that floor*

- We use a *safety monitor* to implement the property. Remember:
  - A safety monitor runs in parallel (lock-step) with the program
  - A monitor has an internal state, which can be updated when the program does a *significant* action (or something happens – *a button press*)
  - The monitor should signal an error if an action happens in an incorrect state

Significant Events

Which elevator events do the monitor need to react to?

Significant Events

Which elevator events do the monitor need to react to?

- Button presses in the elevator

Significant Events

Which elevator events do the monitor need to react to?

- Button presses in the elevator
- Button presses at each floor

Significant Events

Which elevator events do the monitor need to react to?

- Button presses in the elevator
- Button presses at each floor
- The arrival of the elevator at a floor

State and Correctness Check

- What is the state of the monitor?

State and Correctness Check

- What is the state of the monitor?

  A data structure that remembers orders to go to a certain floor

State and Correctness Check

- What is the state of the monitor?

  A data structure that remembers orders to go to a certain floor

- What is the correctness check?

State and Correctness Check

- What is the state of the monitor?

  A data structure that remembers orders to go to a certain floor

- What is the correctness check?

  When the elevator arrives at a floor, the order to do so is in the monitor state

A Monitor Implementing the Floor Request Property

```
-module(stop_after_order).
-behaviour(mce_behav_monitor).

%% The monitor state is a set of floor requests
init(_) -> ordsets:new().

%% Called when the program changes state
stateChange(_,FloorReqs,Action) ->
  case interpret_action(Action) of
    {f_button,Floor} ->
      ordsets:add_element(Floor,FloorReqs);
    {e_button,Elevator,Floor} ->
      ordsets:add_element(Floor,FloorReqs);
    {stopped_at,Elevator,Floor} ->
      case ordsets:is_element(Floor,FloorReqs) of
        true -> FloorReqs;
        false -> throw({bad_stop,Elevator,Floor})
      end;
    _ -> FloorReqs
  end
```

Checking the first correctness property

- Checking:

```
> mce:start(#mce_opts
   {program={run_scenario,run_scenario,
             [3,2,[{scheduler,f_button_pressed,[3]}]]},
    is_infinitely_fast=true,
    algorithm={mce_alg_safety,void},
    monitor={stop_after_order,void}}).
```

- Fails…
- We display the counterexample (a program trace) using a custom pretty printer:

```
Floor button 3 pressed
Elevator 1 is moving up
Elevator 1 is approaching floor 2
Elevator 1 is stopping
Elevator 1 stopped at floor 2
```

More Correctness Properties

- Refining the floor correctness property:

  An elevator only stops at a floor after receiving an order to go to that floor, if no other elevator has met the request

  (implemented as a monitor that keeps a set of floor requests; visited floors are removed from the set)

More Correctness Properties

- Refining the floor correctness property:

  An elevator only stops at a floor after receiving an order to go to that floor, if no other elevator has met the request

  (implemented as a monitor that keeps a set of floor requests; visited floors are removed from the set)

- A *Liveness* property:

  If there is a request to go to some floor, eventually some elevator will stop there

Checking Liveness Properties

- For expressing that *something good eventually happens*
- Linear Temporal Logic (always, eventually, until, next, . . .) is used to express liveness properties
- State predicates are Erlang functions
- Example:

```
always(fun liftprop:go_to_floor/3 =>
       eventually fun liftprop:stopped_at_floor/3)
```

- State predicate:

```
go_to_floor(_ProgramState,Action,_PrivateData) ->
  case interpret_action(Action) of
    {f_button,Floor}   -> {true,Floor};
    {e_button,_,Floor} -> {true,Floor};
    _                  -> false
  end.
```

McErlang Status and Conclusions

- Supports a large language subset (full support for distribution and fault-tolerance and many higher-level components)
- Everything written in Erlang
(programs, correctness properties, . . .)
- An alternative implementation of Erlang for testing
(using a much less deterministic scheduler)
- Using McErlang and testing tools like QuickCheck can be complementary activities:
  - Use QuickCheck to generate a set of test scenarios
  - Run scenarios in McErlang