McErlang: a model checker for concurrent Erlang programs

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

Hans Svensson
IT University Gothenburg

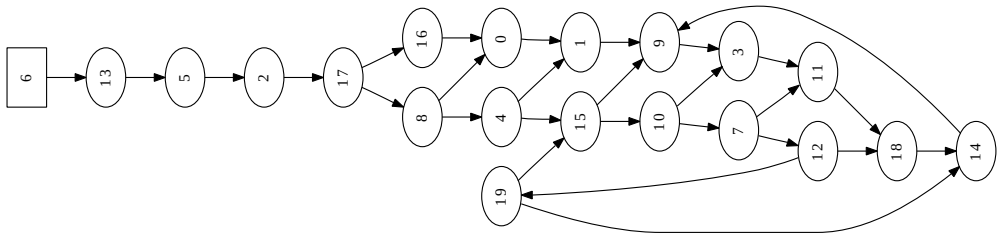
Presentation Outline

- What is model checking & a brief comparison with testing
- McErlang: a model checker for concurrent Erlang programs
- Experiences with using McErlang

More information and download:
`https://babel.ls.fi.upm.es/trac/McErlang`

What is Model Checking

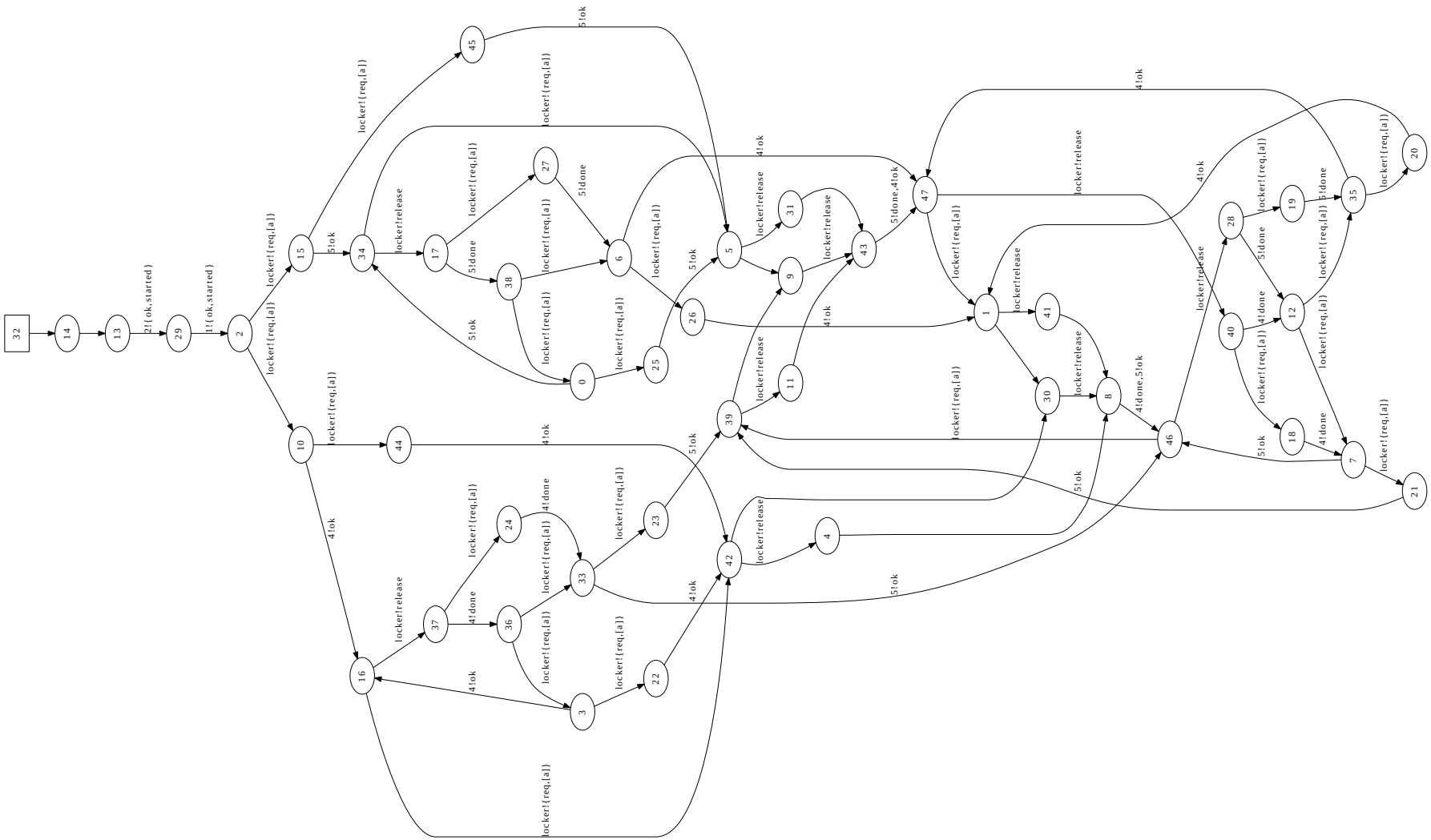
- Construct an abstract **model** of the behaviour of the program, usually a finite state transition graph:



- A node represents a **program state** which records the state of all Erlang processes, all nodes, messages in transit. . .
- **Graph edges** represent computation steps from one program state to another
- **Checking** = explore **systematically** all program states (100% guarantee that all program states seen)
- Establish whether all executions of the program have some **good/bad properties**

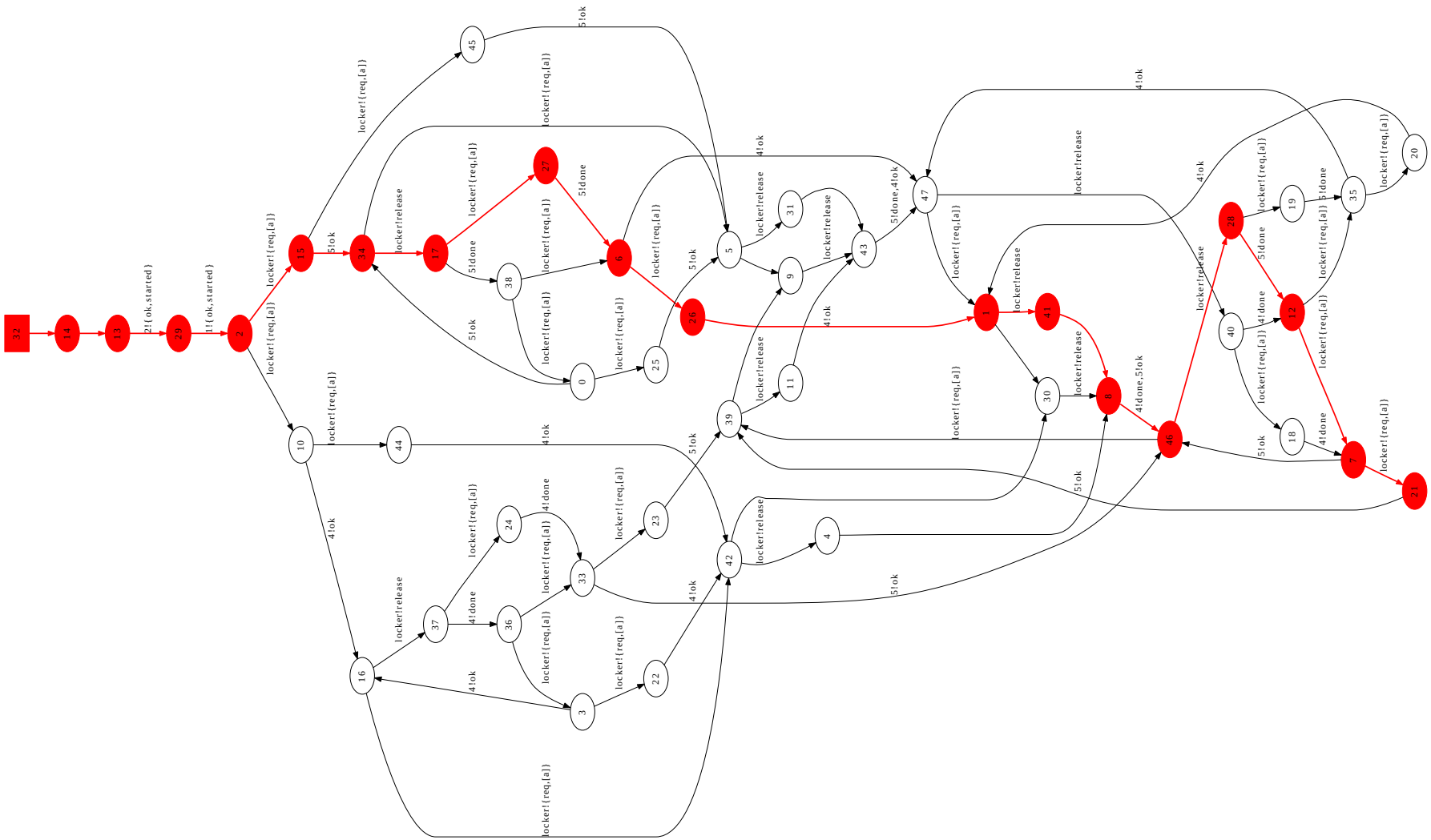
Comparison with Testing

The State Space of a small program:



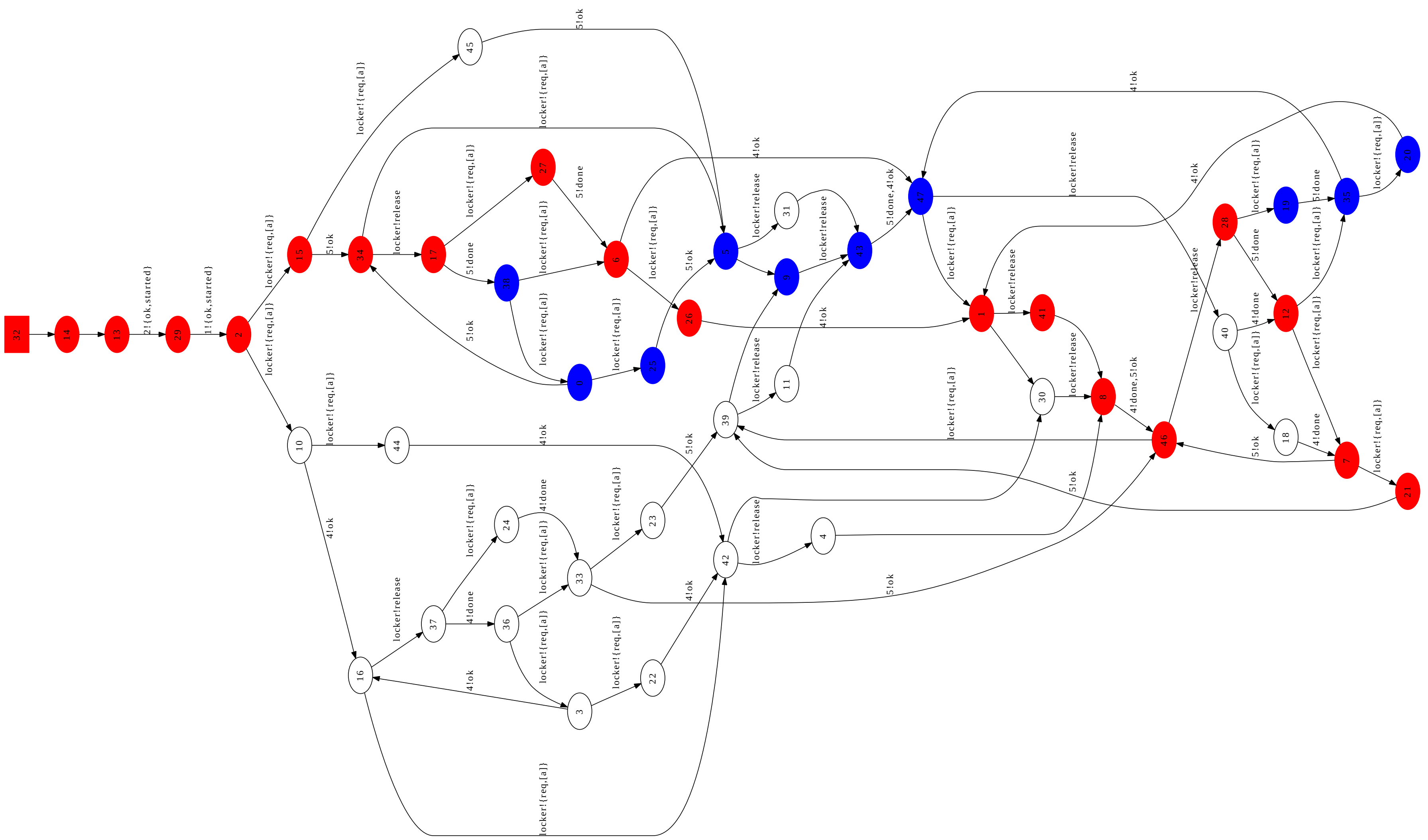
Testing, run 1:

Testing explores one path through the program:



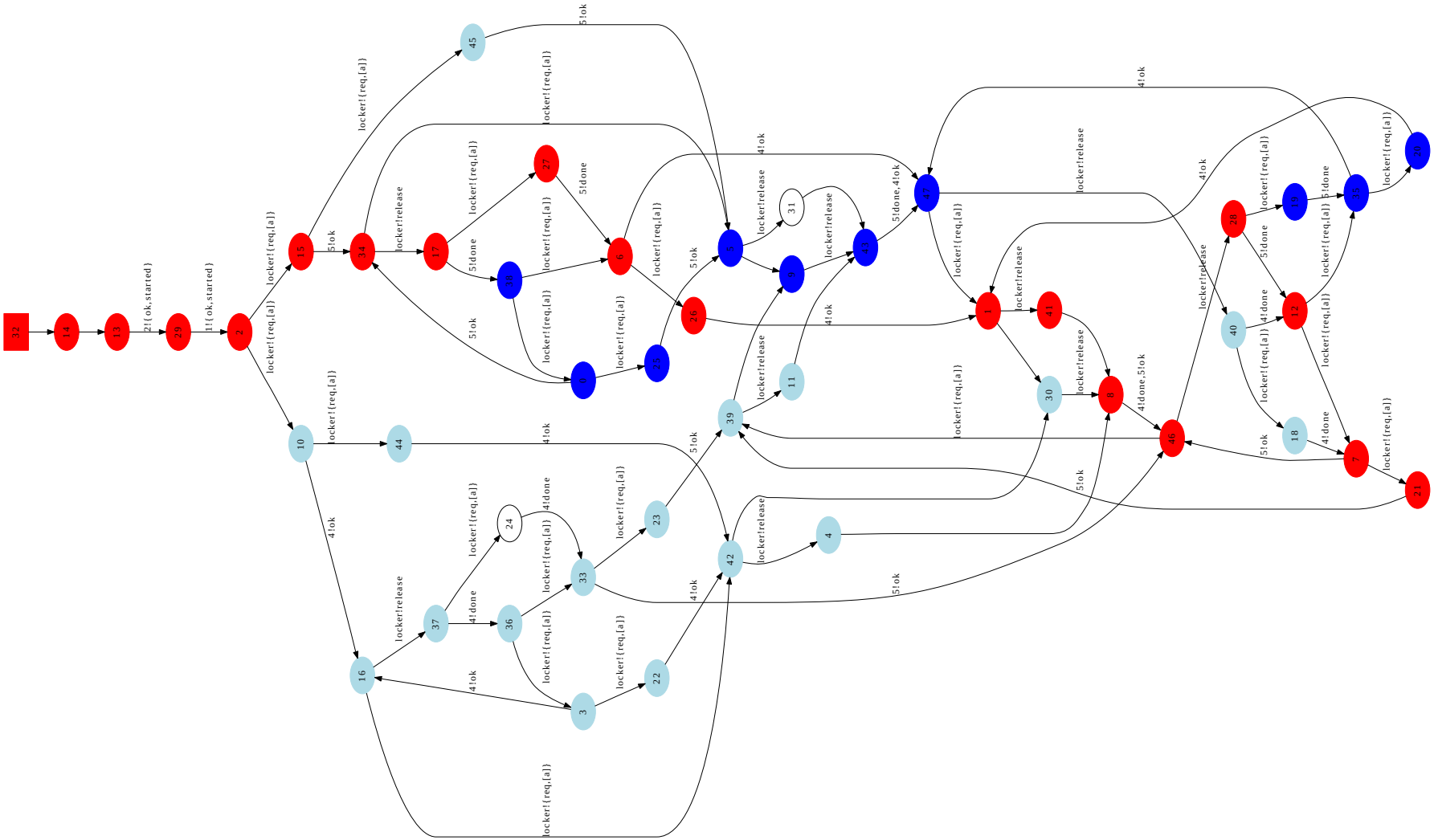
Testing, run 2:

With repeated tests the coverage improves:



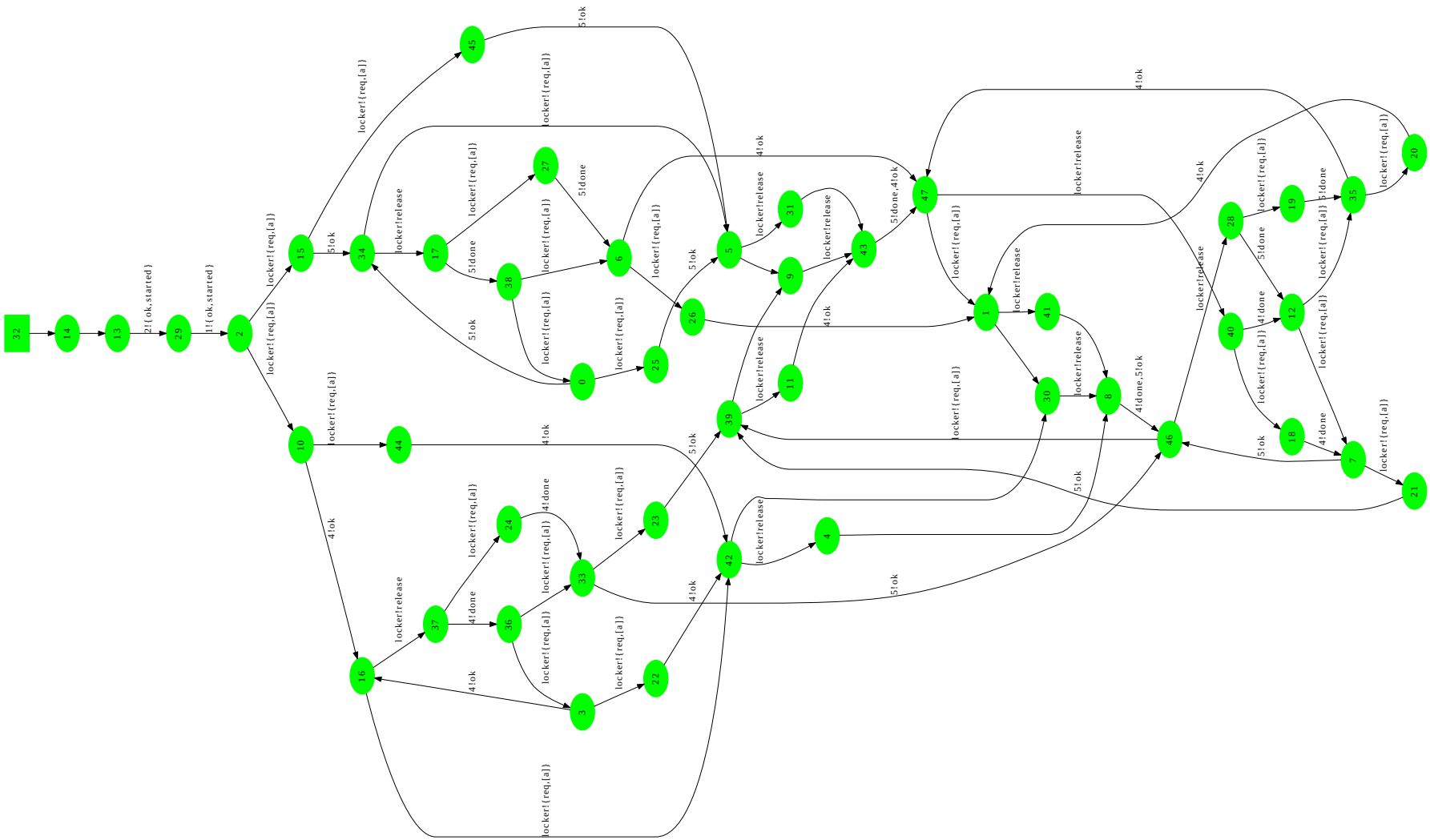
Testing, run n:

But even after a lot of testing some program states may not have been visited:



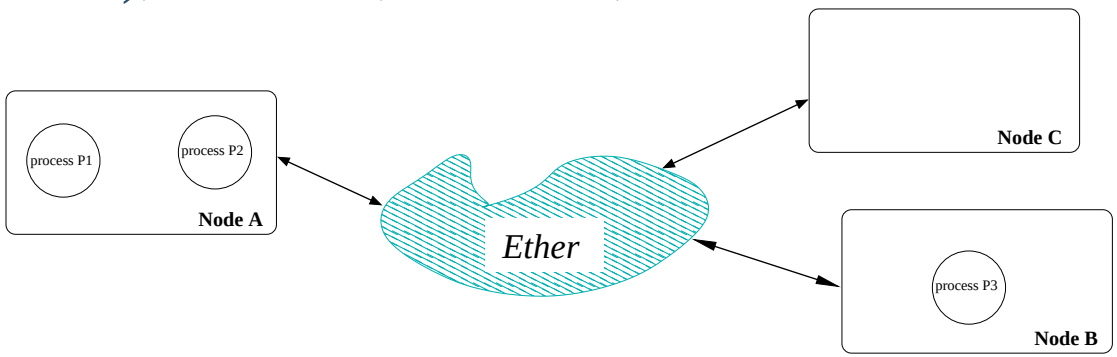
Model checking: 100% coverage

Model checking can guarantee that all states are visited



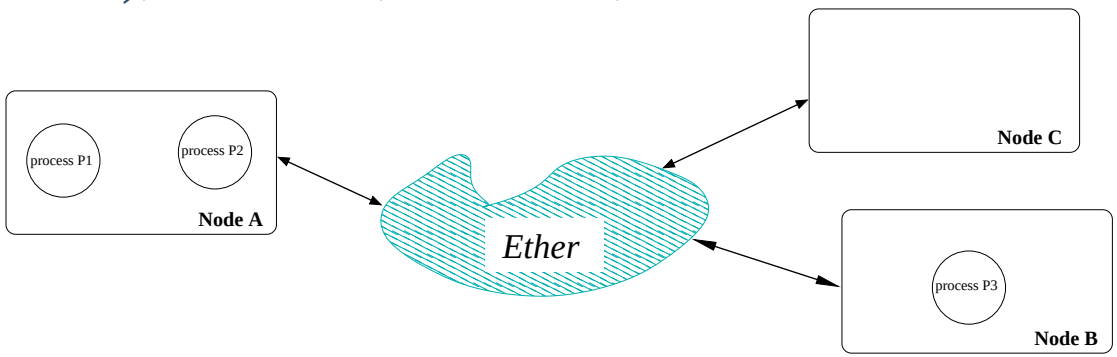
What is the trick? How can we achieve 100% coverage

- Needed: the capability to take a **snapshot** of the Erlang system
  - A **program state** is: the contents of all process mailboxes, the state of all running processes, messages in transit (the ether), all nodes, monitors, …



What is the trick? How can we achieve 100% coverage

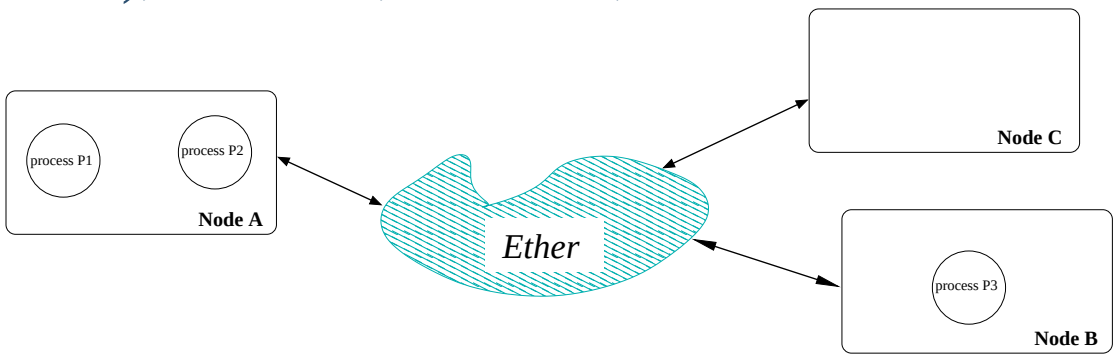
- Needed: the capability to take a **snapshot** of the Erlang system
  - A **program state** is: the contents of all process mailboxes, the state of all running processes, messages in transit (the ether), all nodes, monitors, . . .



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

What is the trick? How can we achieve 100% coverage

- Needed: the capability to take a **snapshot** of the Erlang system
  - A **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
- Difficulties:
  - too many states (not enough memory to save snapshots)
  - we have to save state outside of Erlang (disk writes,. . .)

The McErlang model checker: Design Goals

- Reduce the gap between program and verifiable model (the program *is* the model)
- Write correctness properties in Erlang
- Implement verification methods that permit partial checking when state spaces are too big – Holzmann's bitspace algorithms
- Implement the model checker in a parametric fashion (easy to plug-in new algorithms, new abstractions, . . .)

Relevancy for non Erlang programmers

The model checker has implications for non-Erlang programmers:

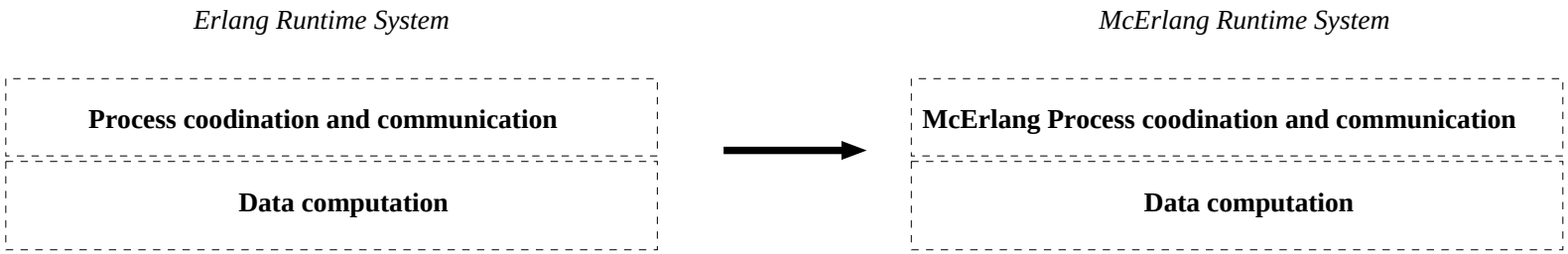
- Erlang is a good *specification* language
- Erlang is a good language for specifying distributed algorithms

The McErlang approach to model checking

- The lazy solution: just execute the Erlang program to verify in the normal 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 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 changes: call `mcerlang:spawn` instead of `erlang:spawn`
- Instead of executing (which would block)

```
receive
  {request, ClientId} -> ...
end
```

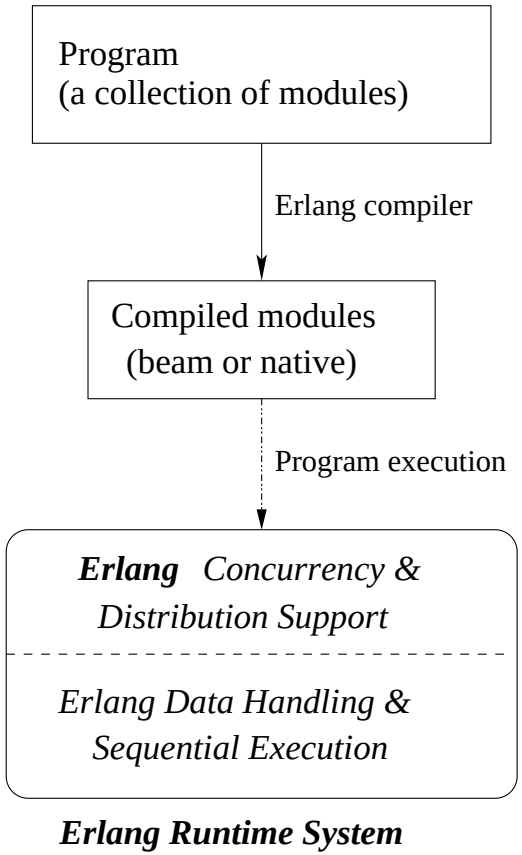
  a compiled function returns a special Erlang value describing the receive request:

```
{'_recv_', {Fun, VarList}}
```

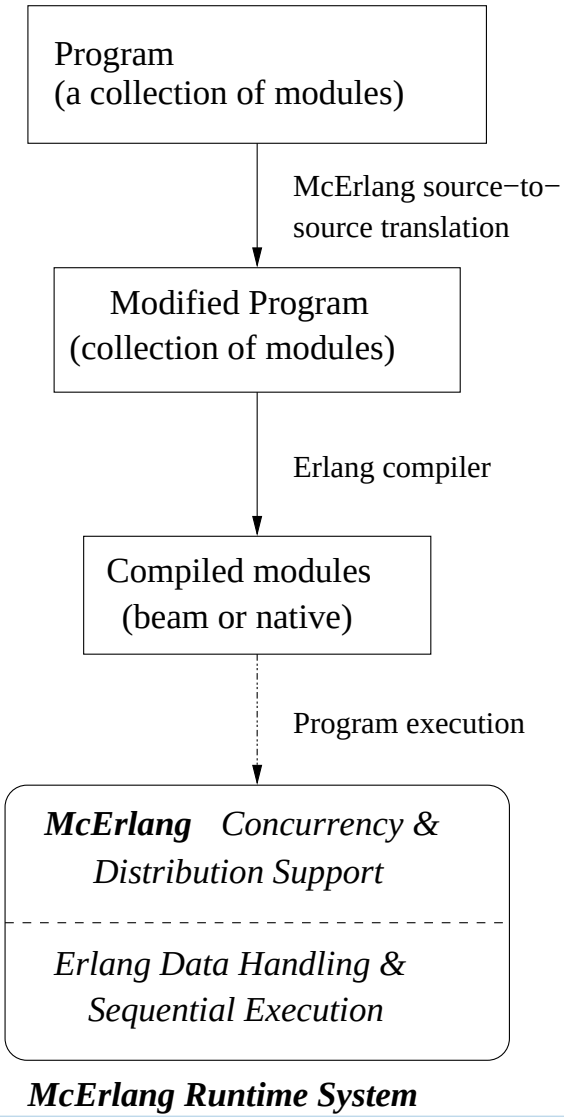
- Compiler works on the HiPE Core Erlang code format

McErlang Workflow

Normal Erlang Workflow:



McErlang Workflow:



Full Erlang Supported?

- Processes, nodes, links, full datatypes supported in McErlang
- Higher-order functions
- Many libraries at least partly supported: supervisor, gen_server, gen_fsm, gen_event, ets, . . .
- No real-time or discrete-time model checking yet

Correctness Properties

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

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 *monitor* is an user function with three arguments:

```
stateChange(State, MonitorState, Actions) ->
  ...
  {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 (success), or signals an error

A monitor example

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

monitorType() -> safety.

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

stateChange(State,MonState,_) ->
  case is_deadlocked(State) of
    true -> deadlock;
    false -> {ok, MonState}
  end.

is_deadlocked(State) ->
  State#state.ether =:= [] andalso
  (not(lists:any
       (fun (P) -> P#process.status =/= blocked end,
        mce_erl:allProcesses(State)))).
```

Writing more complex properties

- Linear Temporal Logic is used
- LTL properties are translated automatically to monitors

Writing more complex properties

- Linear Temporal Logic is used
- LTL properties are translated automatically to monitors

LTL Operators check properties of *program runs*:

- *Always* ϕ

  ϕ holds in all future states of the run
- *Eventually* ϕ

  ϕ holds in some future state of the run
- ϕ_1 *Until* ϕ_2

  ϕ_1 holds in all states until ϕ_2 holds (but ϕ_2 may never hold)
- Standard predicates: negation $\neg\ \phi$, conjunction $\phi_1 \vee \phi_2$,...
- Predicates on actions or Erlang states: `Pid!{request,A}`
  (a request message is sent to some process)

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 (BEAM) emulator version 5.6.5 [source] [smp:2]

Eshell V5.6.5  (abort with ^G)
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:

```
> mcerl
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
...

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(get(result)).
Starting debugger with a stack trace; execution termi
  user program raised an uncaught exception.

stack(@2)> where().
2:

1: process <node0,3>:
 run #Fun<example.2.125>([])
 process <node0,3> died due to reason badarg

0: process <node0,1>:
 run function example:start([])
 spawn({#Fun<example.1.278>,[]},[]) --> <node0,2>
 spawn({#Fun<example.2.125>,[]},[]) --> <node0,3>
 process <node0,1> was terminated
 process <node0,1> died due to reason normal
```

Error Cause

- Apparently in one program run the second process spawned (the one calling the echo server) was run before the echo server itself.
- Then upon trying to send a message

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

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

The Elevator Example

- Let us attempt something a bit more difficult. Can we check the elevator example using McErlang?

The Elevator Example

- Let us attempt something a bit more difficult. Can we check the elevator example using McErlang?
- So what API:s does it use?

The Elevator Example

- Let us attempt something a bit more difficult. Can we check the elevator example using McErlang?
- So what API:s does it use?
- `gs`, `lists`, `gen_event`, `supervisor`, `application`, `gen_fsm`, `timer`, `erlang`

  Should be ok
- Around 2000 lines of code

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
- Seems to work...

Model checking the elevator under McErlang

Model checking is more complicated:

- The `gs` graphics will not make sense when model checking, so we shut it off

Model checking the elevator under McErlang

Model checking is more complicated:

- The `gs` graphics will not make sense when model checking, so we shut it off
- The program is very geared to smooth graphical display.

  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 more complicated:

- The `gs` graphics will not make sense when model checking, so we shut it off
- The program is very geared to smooth graphical display.

  We modify the program to only have three (3) intermediate points between elevator floors (normally 20).
- In total, about 10 lines of code had to be changed to enable model checking

Correctness Properties

- What is a good correctness property for the elevator system?
- We “steal” one from a QuickCheck presentation:

  the elevator only stops at a floor after receiving an order to go to that floor

A Monitor Implementing the Floor Request Property

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

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

Checking the Property

Starting a McErlang run:

```
mce:start
(#mce_opts
  {program={?MODULE,start_it,[Experiment]},
   algorithm={mce_alg_safety,void},
   shortest=true,
   monitor={stops_at_floors,void}}).
```

A small scenario that shows the error is the system of two elevators with two floors:

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

The minimal number of steps to generate a counterexample is 55 for this scenario.

Closer Analysis

An error trace (from the McErlang debugger):

```
{notify,{e_button,1,2}}
{notify,{move,1,up}}
{notify,{approaching,1,2}}
{notify,{stopping,1}}
{notify,{e_button,2,2}}
{notify,{move,2,up}}
{notify,{approaching,2,2}}
{notify,{stopping,2}}
{notify,{stopped_at,2,2}}
{notify,{open,2}}
{notify,{stopped_at,1,2}}
{notify,{open,1}}
```

Hmm...our property is wrong. The elevator buttons in elevator 1 and 2 are independent.

McErlang Status and Conclusions

- Lightweight “everything-in-Erlang” approach
- Supports a large language subset (full support for distribution and fault-tolerance and many higher-level components)
- 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
  - Check these scenarios in McErlang
- More info:
  `https://babel.ls.fi.upm.es/trac/McErlang`