A Short Course on McErlang – model checking for Erlang

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

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 normal Erlang runtime system for process handling and communicating has been replaced with a new runtime system written in Erlang
- `erlang:send`, `erlang:spawn`, `erlang:monitor`, … have been reimplemented

McErlang 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.

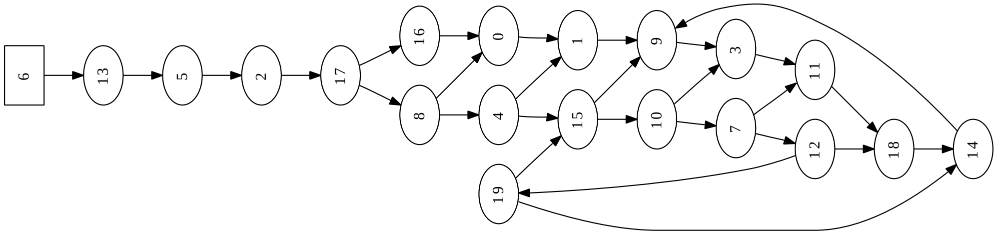
Presentation Outline

- What is model checking & a brief comparison with testing
- McErlang: installing and usage
- Hands-on with McErlang:

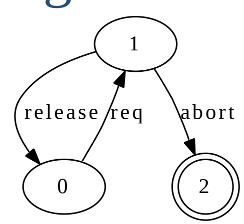
  A prepared example (a lift control system)
  or work with your own examples

What is Model Checking

- Run the program in a controlled manner so that all program states are visited (visualized as a finite state transition graph):

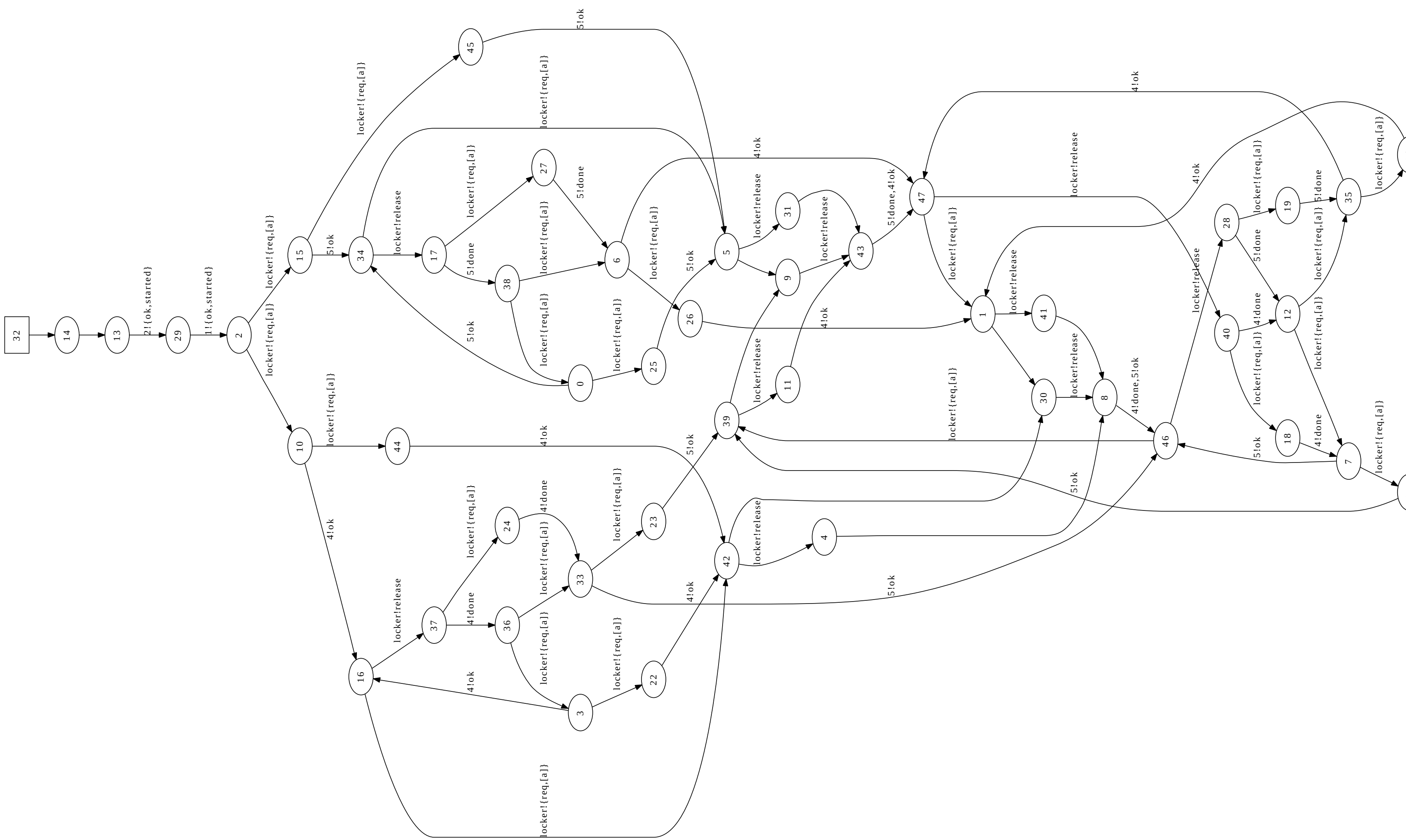


- A node represents a **program state** which records the state of all Erlang processes, all nodes, messages in transit. . .
- **Graph edges** represent computation steps from one program state to another
- **Correctness Properties** are automata that run in lock-step with the program; they inspect each program state to determine whether the state is ok or not



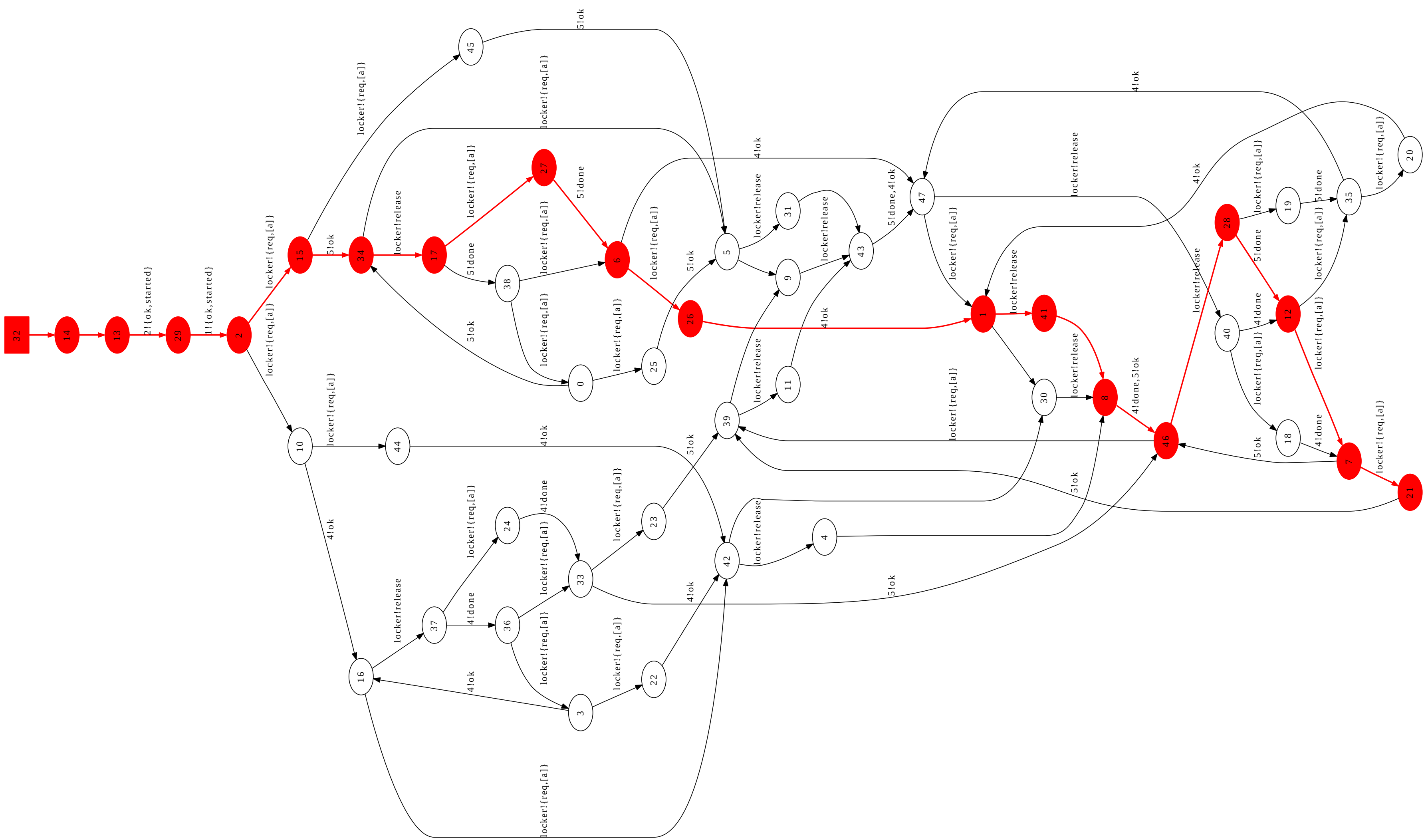
Comparison with Random Testing

The State Space of a small program:



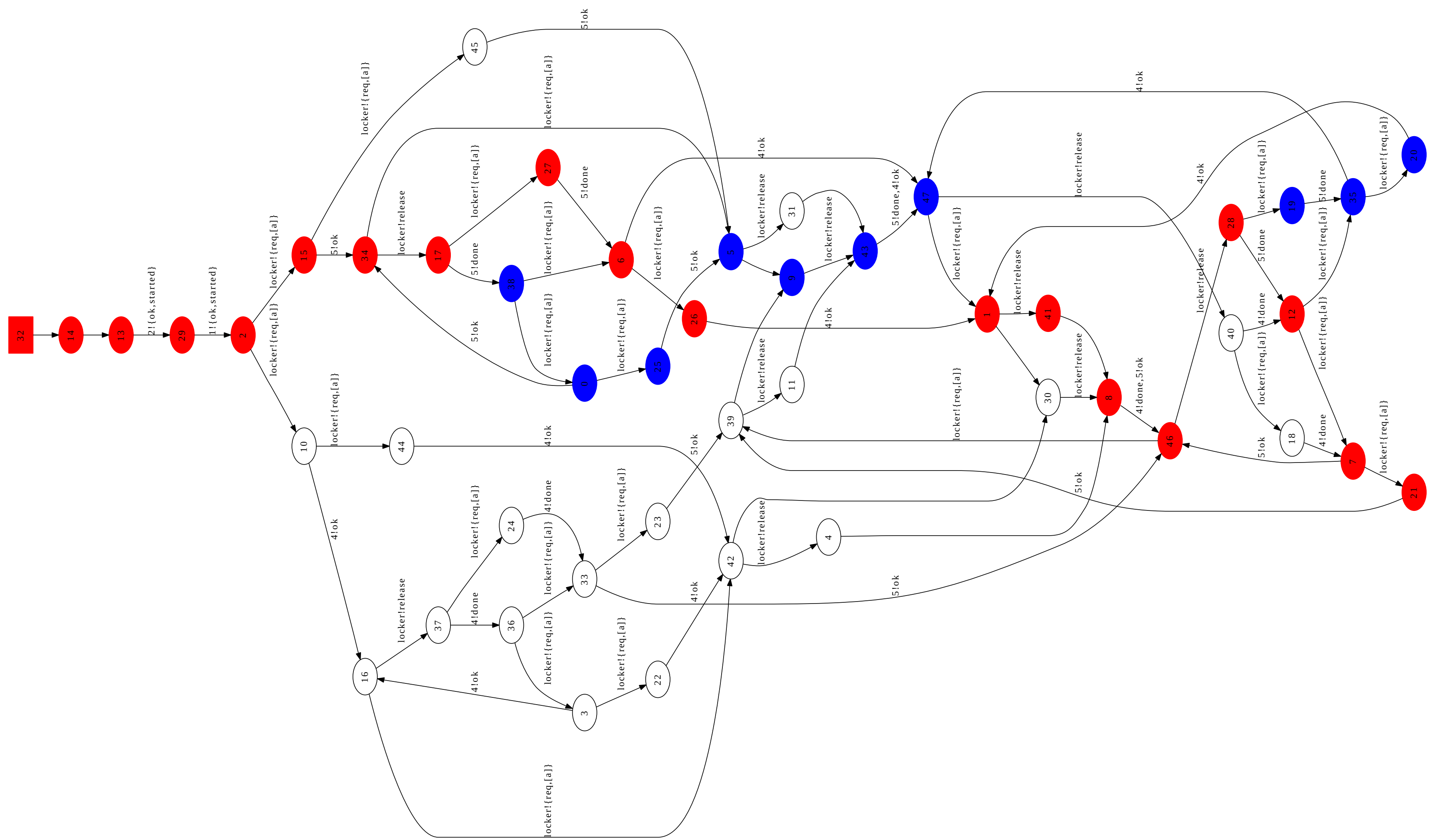
Testing, run 1:

Random testing explores one path through the program:



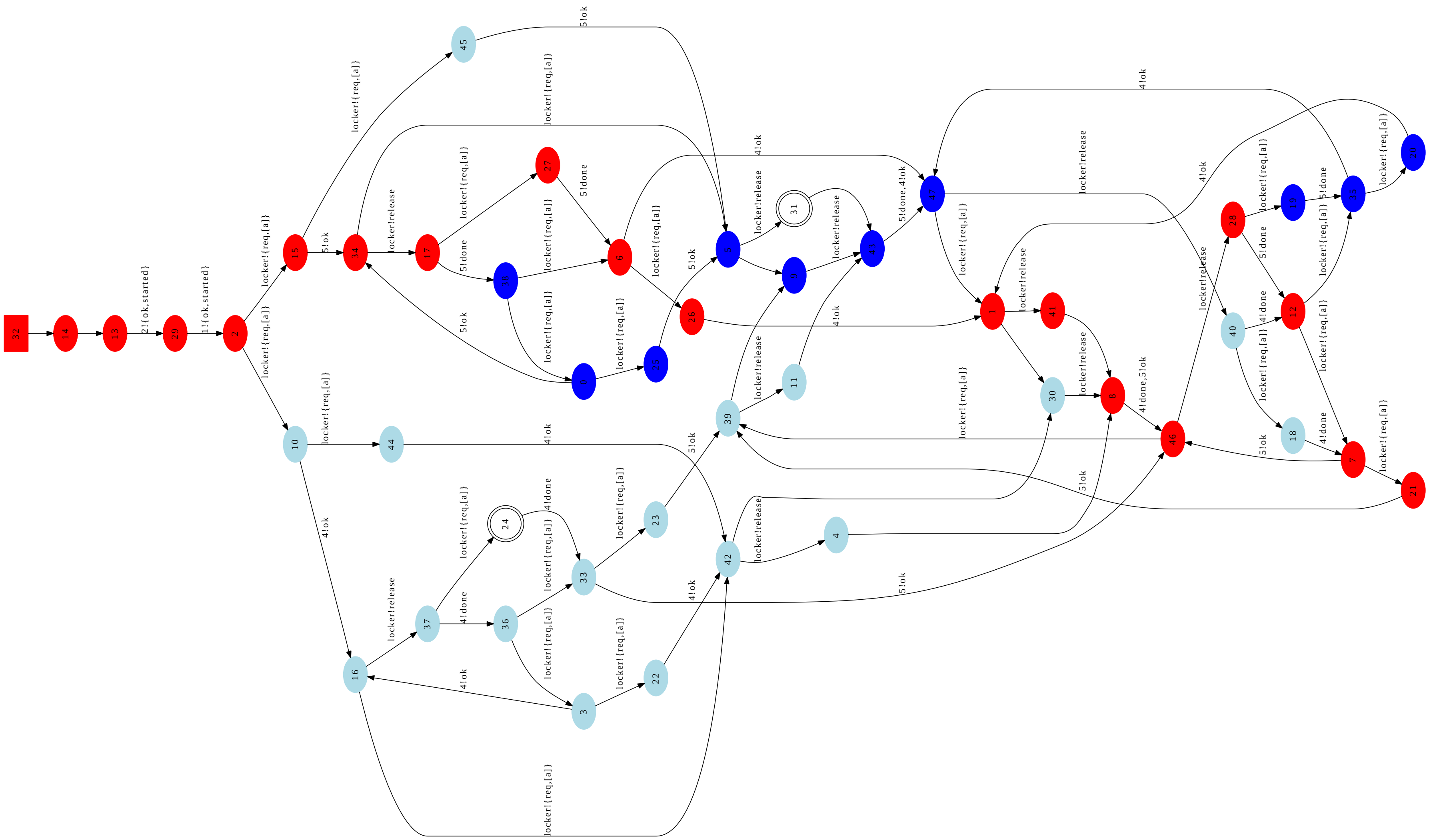
Testing, run 2:

With repeated tests the coverage improves:



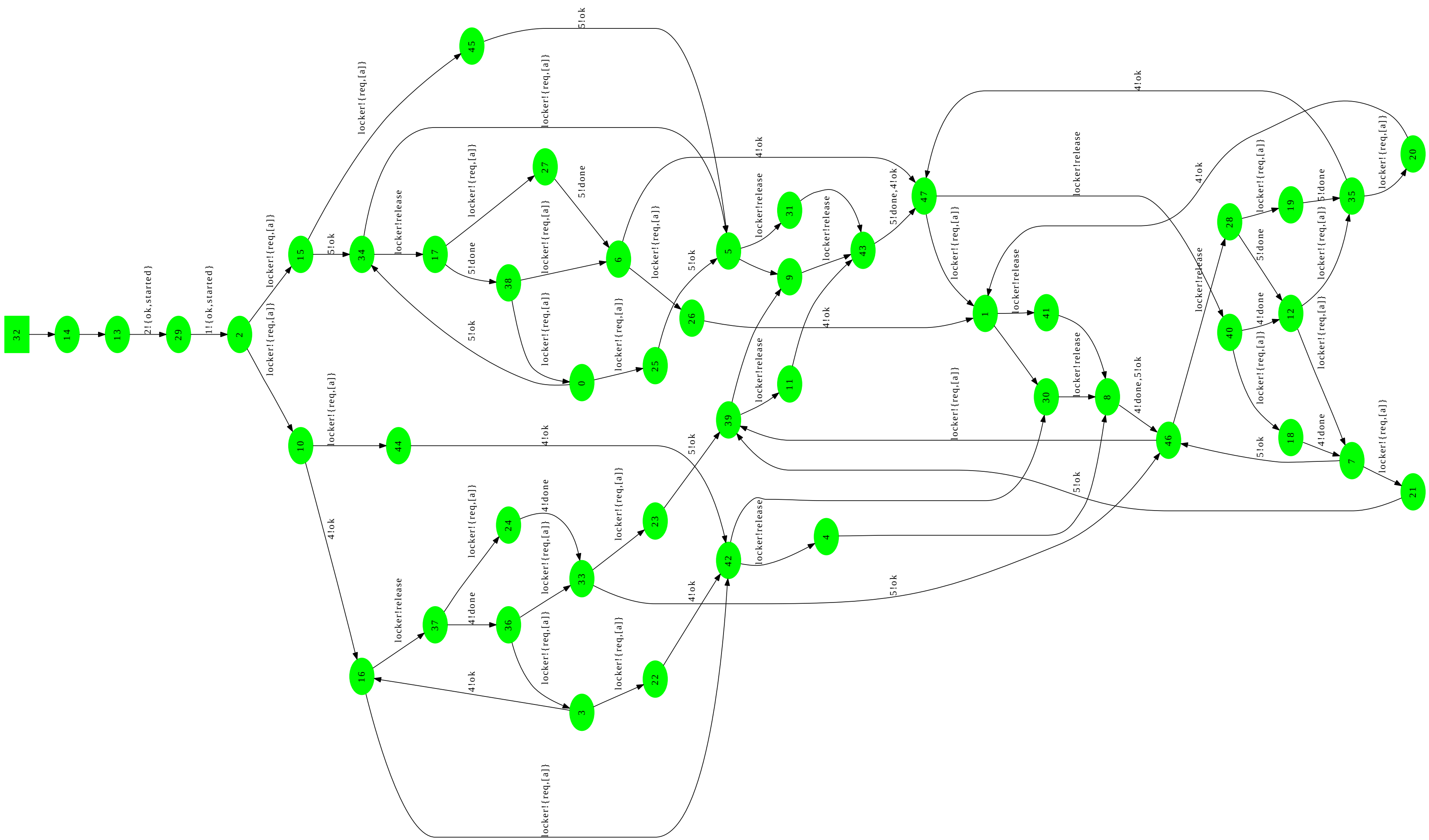
Testing, run n:

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



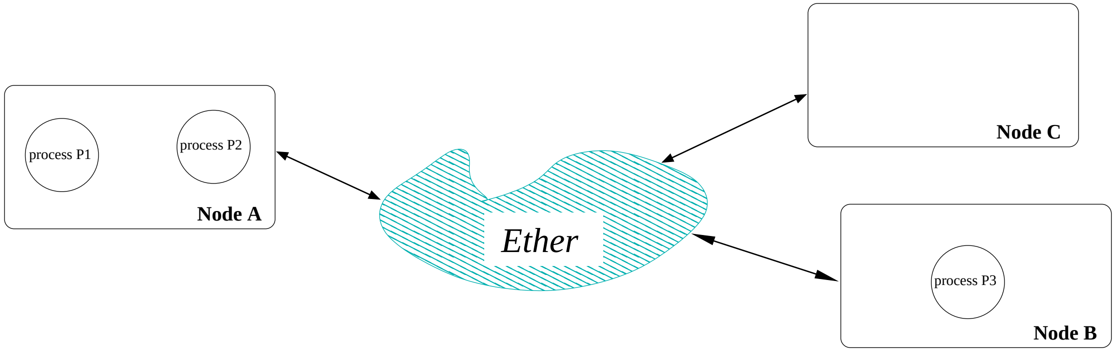
Model checking: 100% coverage

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



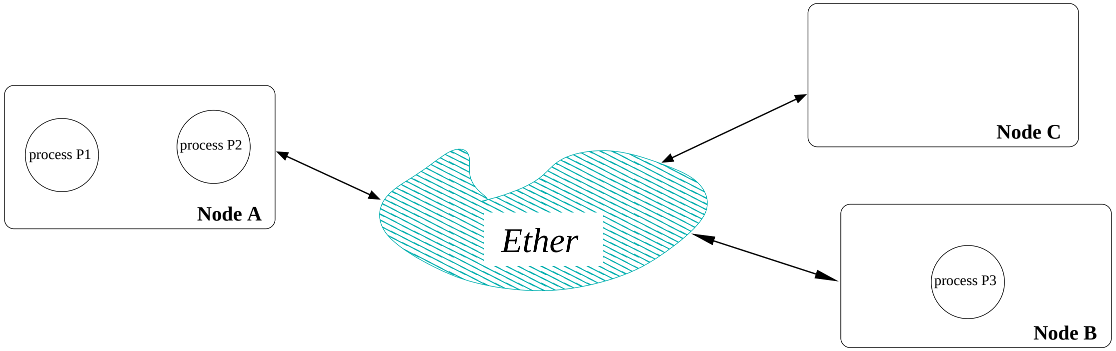
What is the trick? How can we achieve 100% coverage

- 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

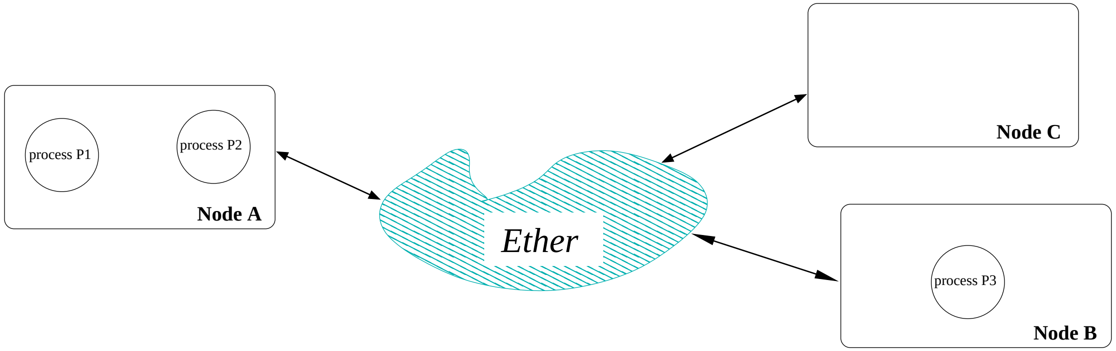
- Needed: the capability to take a **snapshot** of the Erlang system
  - A **program state** is: the contents of all process mailboxes, the state of all running processes, messages in transit (the ether), all nodes, monitors, . . .



- 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

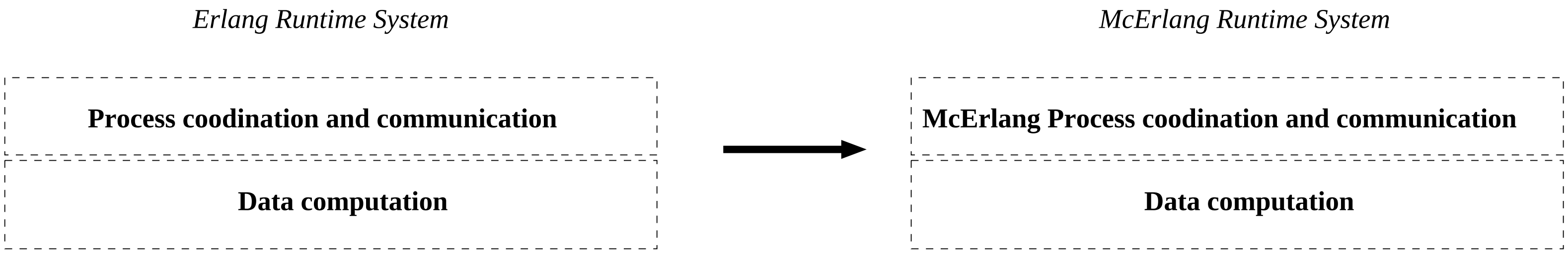
- Reduce the gap between program and verifiable model (the program *is* the model)
- Write correctness properties in Erlang
- Implement verification methods that permit partial checking when state spaces are too big – Holzmann's bitspace algorithms
- Implement the model checker in a parametric fashion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

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 changes: call `mcerlang:spawn` instead of `erlang:spawn`
- Instead of executing (which would block)

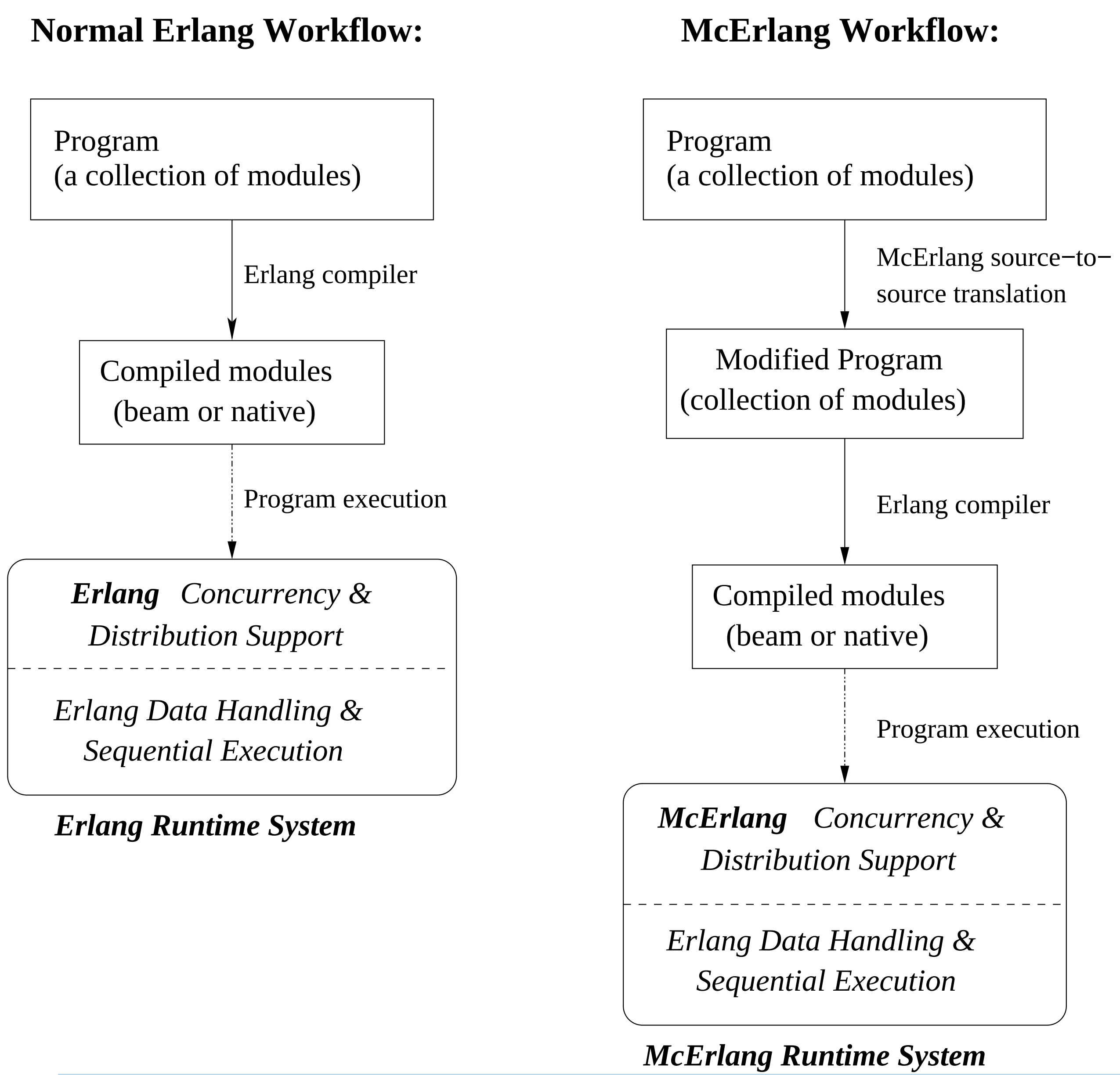
  ```
  receive
    {request, ClientId} -> ...
  end
  ```

  a compiled function returns a special Erlang value describing the receive request with a new anonymous function implementing the clauses of the **`receive`**:

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

- McErlang translator works on the HiPE Core Erlang code

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** implementation yet

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

behaves the same as

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

Extensions to Erlang in McErlang

- Nondeterminacy:

```
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 doesn't already exist

McErlang in Practise: downloading

- Read https://babel.ls.fi.upm.es/trac/McErlang/
- Use subversion to check out the McErlang sources:

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

- Get bugfixes and improvements using subversion:

```
svn update
```

Installing

- We use Ubuntu – Fedora, probably works too
  McErlang doesn’t work well under Windows
- Compile McErlang:

  ```
  cd McErlang; make
  ```

- Put `scripts` directory on the command path (in Bash):

  ```
  export PATH=~/McErlang/scripts:$PATH
  ```

- Read the manuals:

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

McErlang Directory Organisation

- `scripts` – `mcerl_compile` and `mcerl`
- `configuration/funinfo.txt` – controls translation
- `doc` – usermanual and tutorial
- `examples`
- `lib/erlang/src` – re-implementation of some OTP behaviours
- `algorithms` – execution mode (simulation/model checking)
- `monitors` – standard correctness properties
- `scheduler` – Erlang scheduler
- `stacks`, `tables`, `abstractions` (tool parameters)

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 -output_dir ebin`
- The translation is controlled by the `funinfo.txt` file (an application specific configuration file can be given)
- 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 is a “result value” which can be inspected using the functions in the `mce_result` module. The result value is normally stored in the process dictionary under the key `result`.

McErlang runtime options

More `#mce_opts{}` record options:

- `sim_external_world = true() | false()`
  McErlang does I/O with external world? (false)
- `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)
- `terminate = true() | false()`
  Let the runtime system randomly terminate processes (false)
- `is_infinitely_fast = true() | false()`
  Prohibits (non-zero) timeouts (caused by **`after`** clauses in **`receive`** statements) from occurring if non-timeout transitions are enabled. This corresponds to the assumption that the system is infinitely fast (false)

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, which *must* be of type `safety`, on *all* program states of the program
  (either suceeds or returns a *counterexample*, an execution path leading to a state failing the monitor)

- `mce_alg_combine`
  This algorithm provides a method to combine two other algorithms (e.g., simulation and model checking)

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!

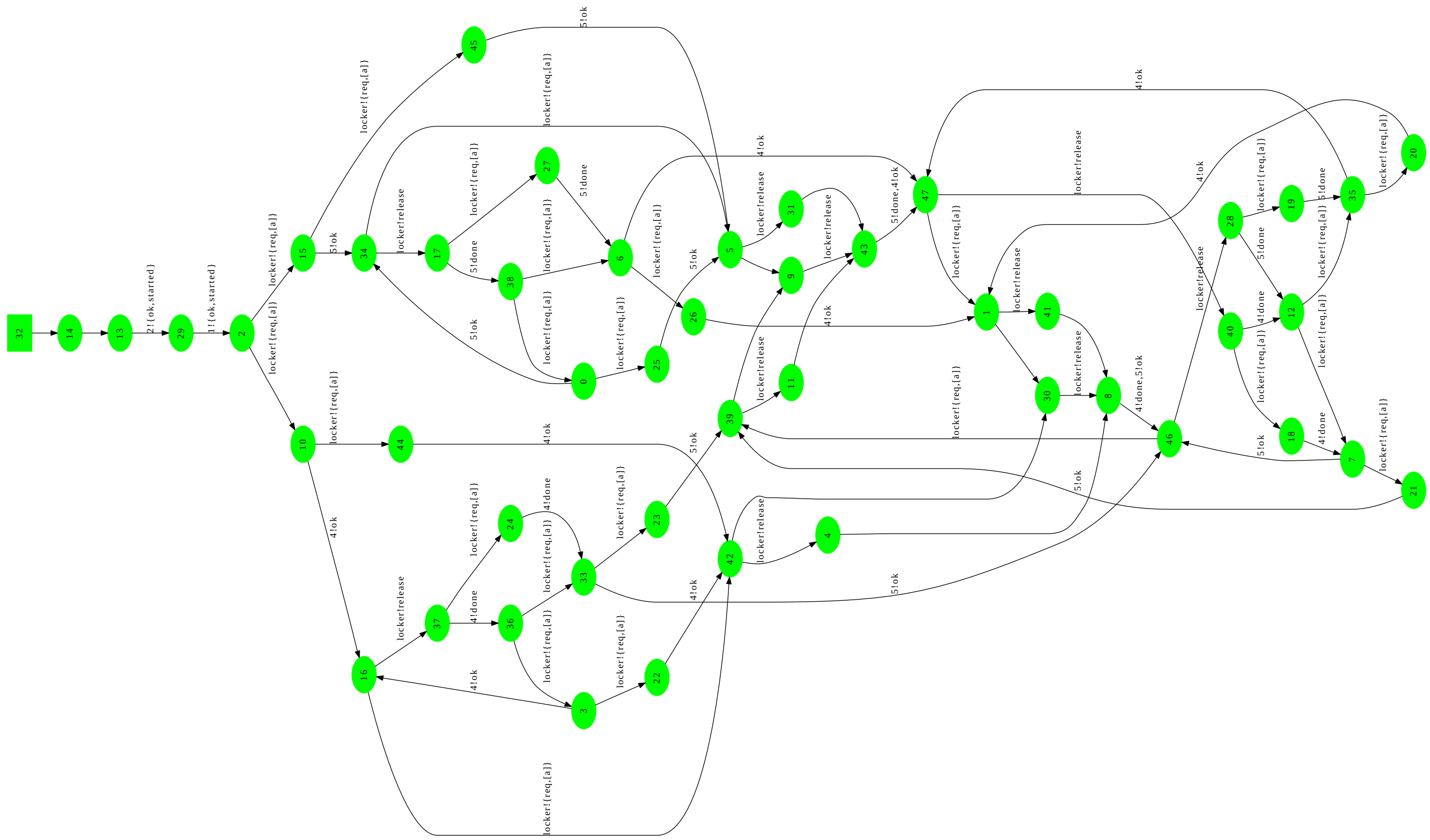
  A *safety monitor* is an user function with three arguments:

  ```
  stateChange(State, MonitorState, 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

Safety Monitors

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



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)))).
```

Facultad de Informática
Universidad Politécnica de Madrid

ProTest
property based testing

What can monitors observe?

- Program actions such as e.g. sending or receiving a message
- Program state such as e.g. contents of process mailboxes, name of registered processes
- Indirectly the values of some program variables (but are difficult to access)
- Programs can be instrumented with special “probe actions” that are easy to detect in monitors
- Programs can be instrumented too with special “probe states”, which are persistent (actions are transient)

Some Predefined Monitors

- `{mce_mon_deadlock, Any::any()}`
  Checks that there is at least one non-deadlocked process
- `{mce_mon_queue, MaxQueueSize::int()}`
  Checks that all queues contain at most `MaxQueueSize` elements.

Checking Liveness Properties (tomorrow)

- For expressing that “something good eventually happens”
- In McErlang Linear Temporal Logic is used to express liveness properties

Checking Liveness Properties (tomorrow)

- For expressing that "something good eventually happens"
- In McErlang Linear Temporal Logic is used to express liveness properties

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)

The McErlang Debugger

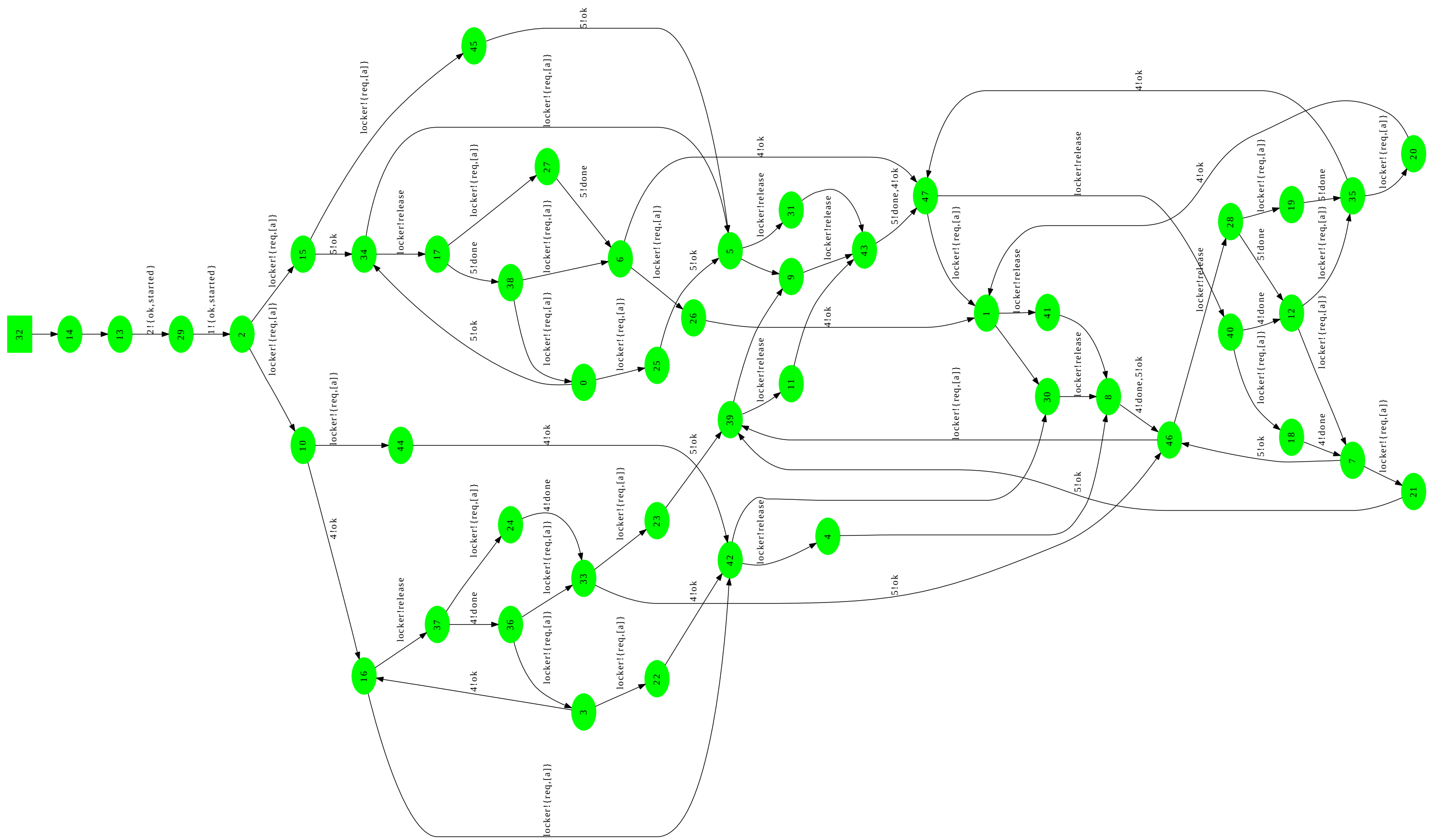
- There is a rudimentary debugger for examing model checking counter examples
- After a failed model checking run, start the debugger on the counterexample using:

  `mce_erl_debugger:start(get(result))`

- For further details see the user manual

Things that can go wrong

- McErlang runs out of memory – too many states/too long runtime stack



- Why? (program uses timers, counters, random values, ...)

Things that can go wrong

- McErlang runs out of memory – too many states/too long runtime stack
- Why? (program uses timers, counters, random values, ...)
- Possibly **fix** by using a fixed size state table implementation:

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

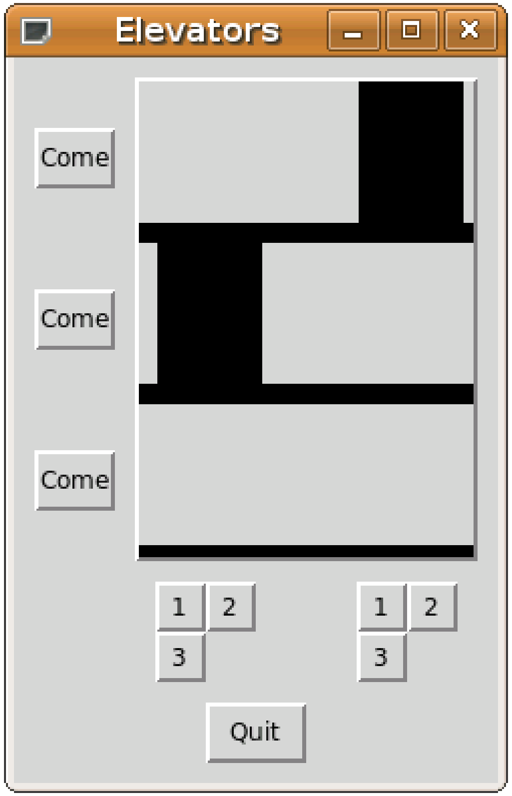
  implements a hash table where states are hashed into an integer value, and there is no collision handling (i.e., different states can be mapped to the same hash value without being detected)

- **and** we can also use a bounded stack

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

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:

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

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 $\Rightarrow$ We shut it off in model checking mode

Model checking the elevator under McErlang

Model checking is 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 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 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

Course Material

- Download the source code from the McErlang wiki page: `https://babel.ls.fi.upm.es/trac/McErlang/`
- Attachment: `elevator_code.tar.gz`
- The file `elevator_example/exercises.txt` contains instructions

Correctness Properties

Correctness Properties

- *No runtime exceptions*

Correctness Properties

- *No runtime exceptions*

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

  (implemented as a monitor that keeps a set of floor requests, and checks that visited floors are in the set)

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(_,FloorReqs,Actions) ->
  ...
  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
  ...
```

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

Scenarios

- Instead of specifying one big scenario with a really big state space, we specify a number of smaller scenarios
- QuickCheck can be used to generate them

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
  - Analyze results in QuickCheck
- https://babel.ls.fi.upm.es/trac/McErlang/