Recent Improvements to the McErlang Model Checker

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

Work supported by the EU ProTest Project
and the PROMESAS project funded by the Madrid Regional Government

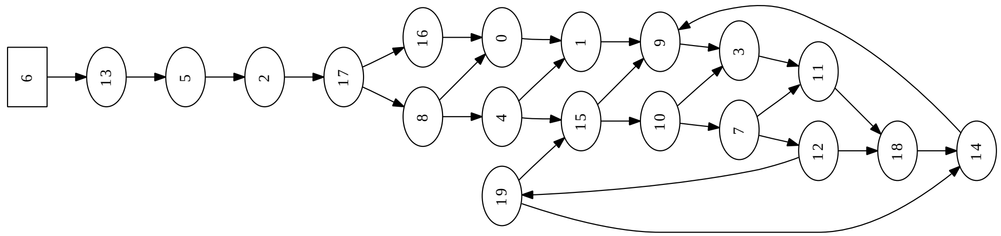
Presentation Outline

- What is model checking & a brief comparison with testing
- McErlang: a model checker for concurrent Erlang programs
- Recent improvements to McErlang

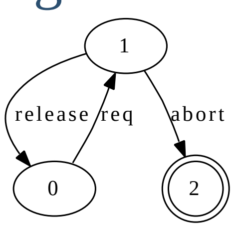
More information and download:
https://babel.ls.fi.upm.es/trac/McErlang/

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):

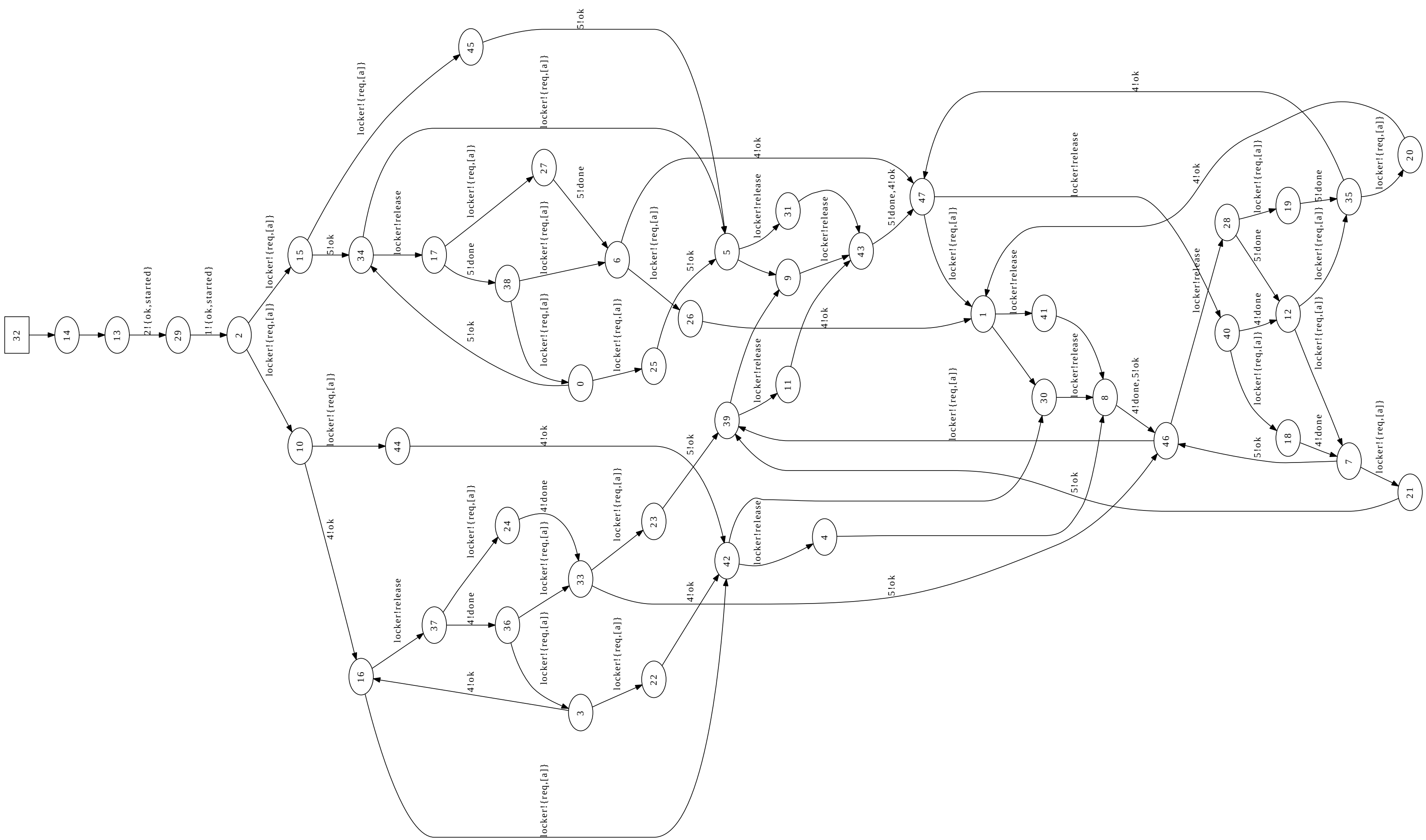


- A node represents a **program state** which records the state of all Erlang processes, all nodes, messages in transit. . .
- **Graph edges** represent computation steps from one program state to another
- **Correctness Properties** are automata that run in lock-step with the program; they inspect each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



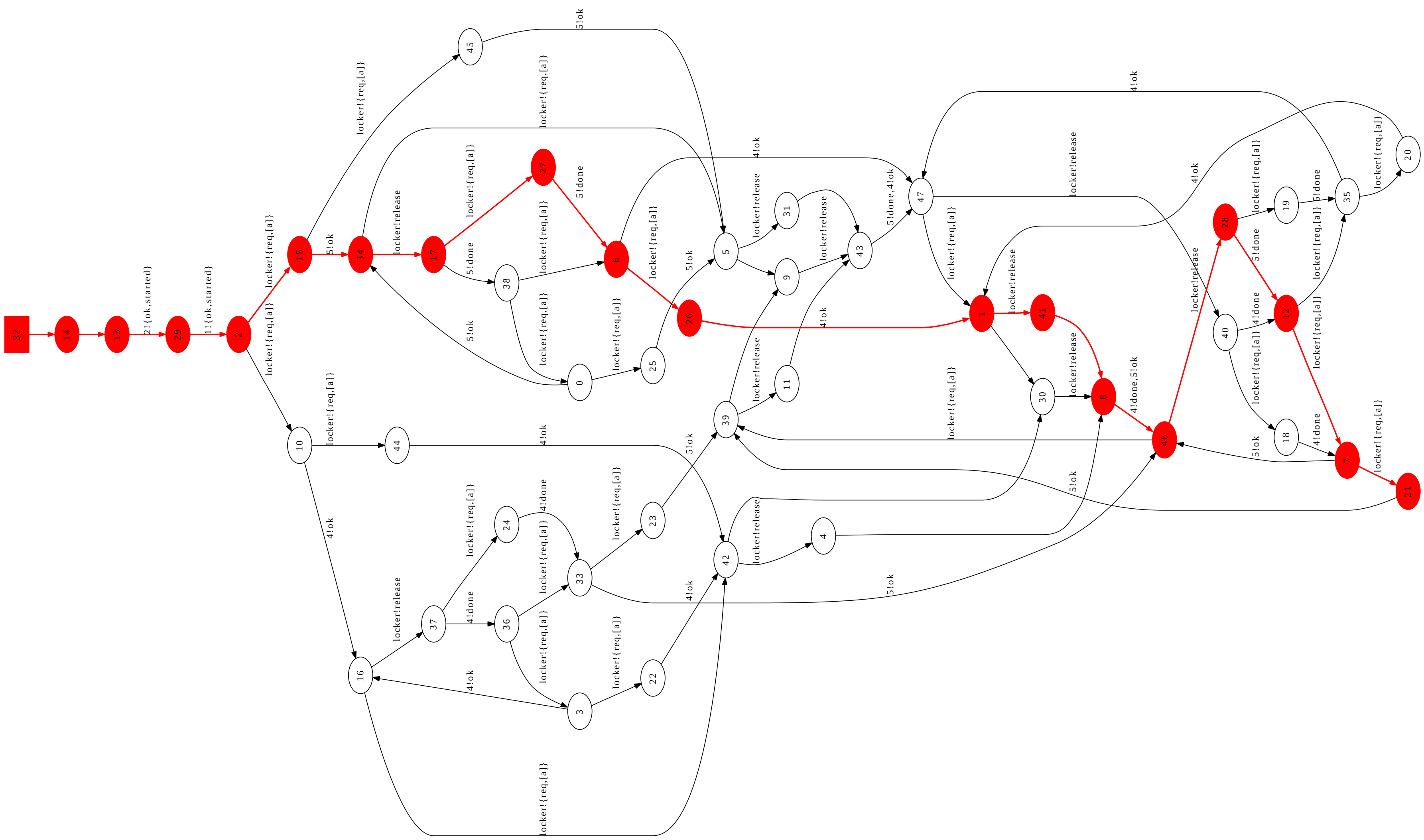
Comparison with Random Testing

The State Space of a small program:



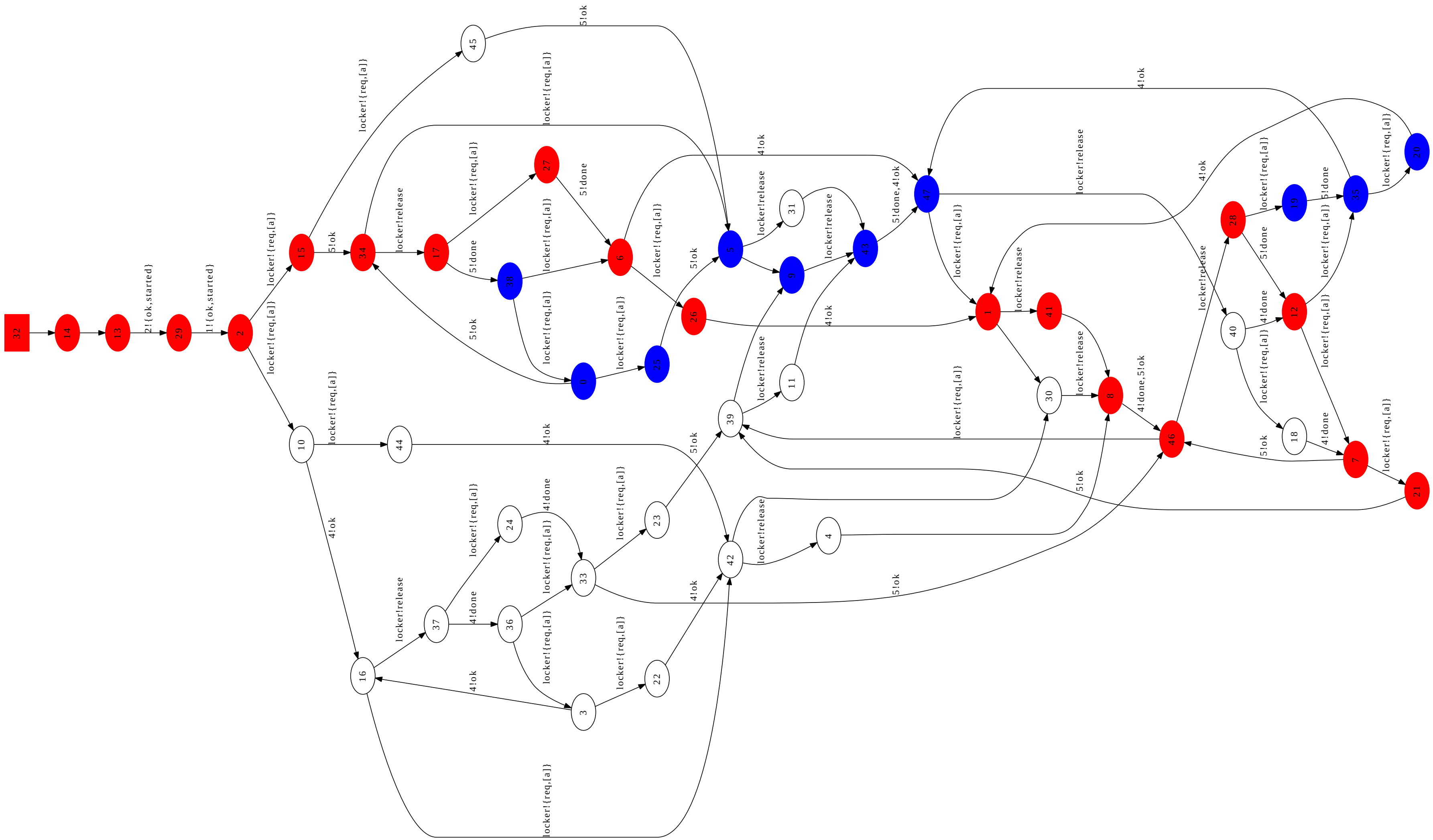
Testing, run 1:

Random testing explores one path through the program:



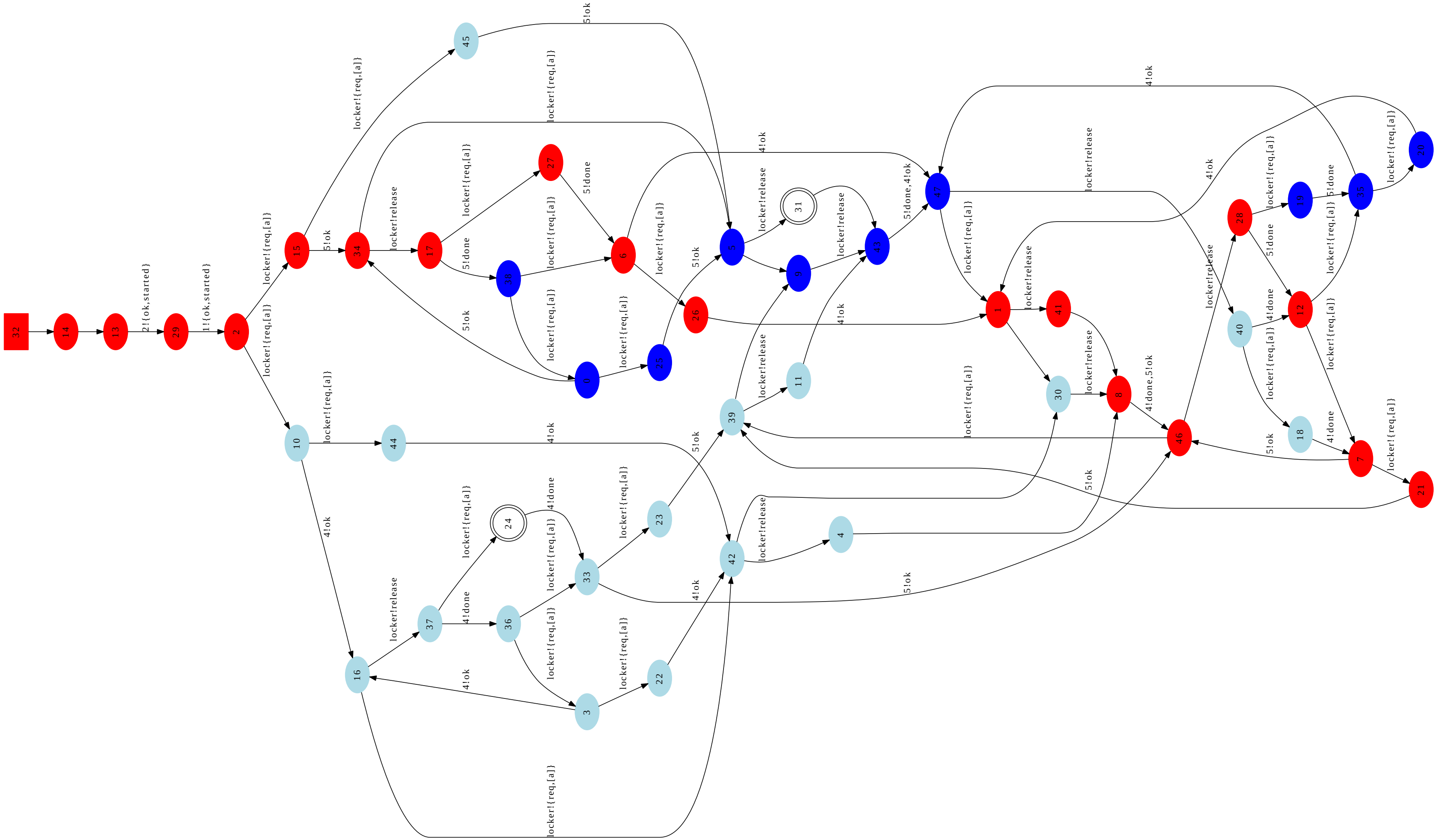
Testing, run 2:

With repeated tests the coverage improves:



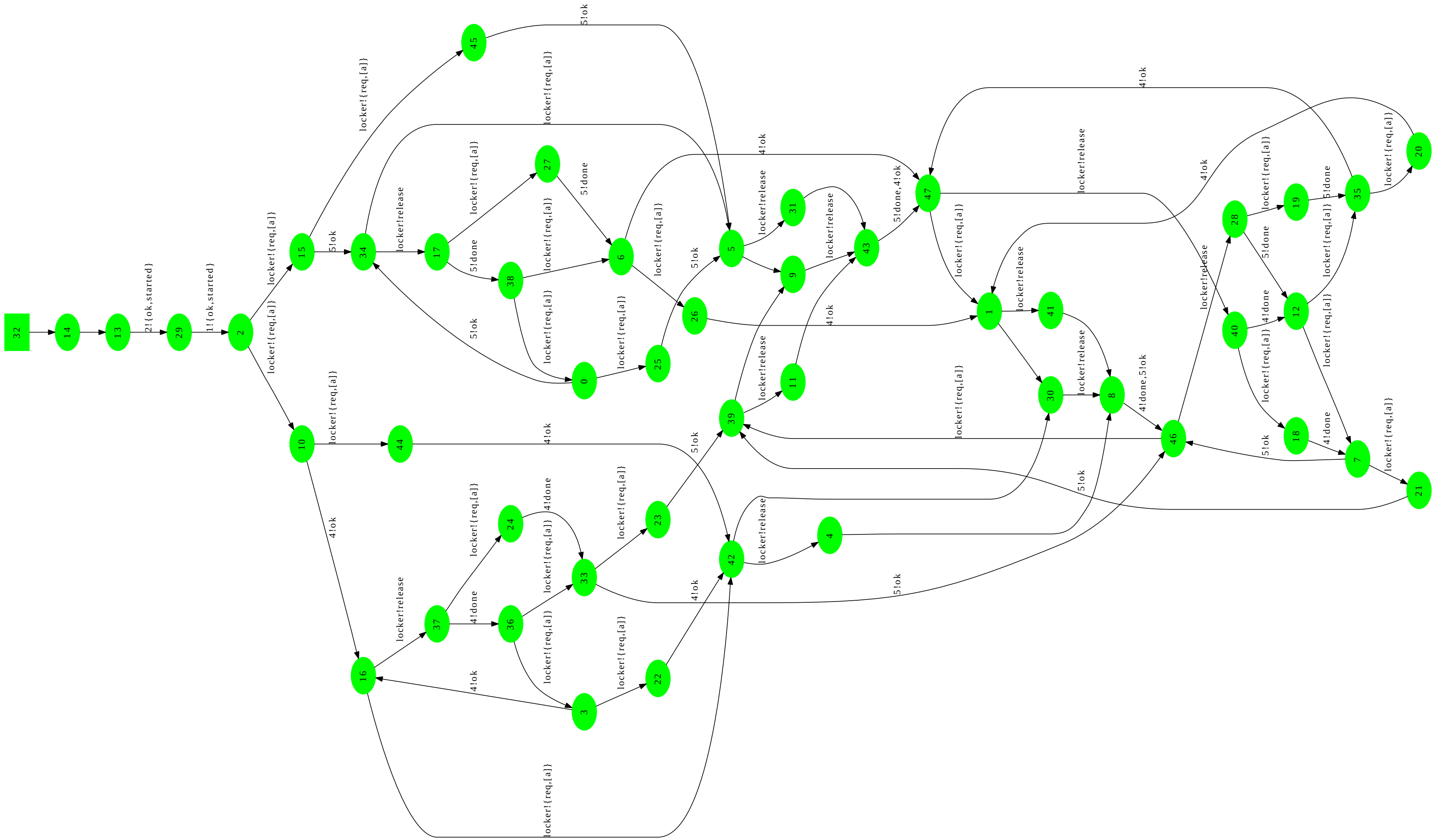
Testing, run n:

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



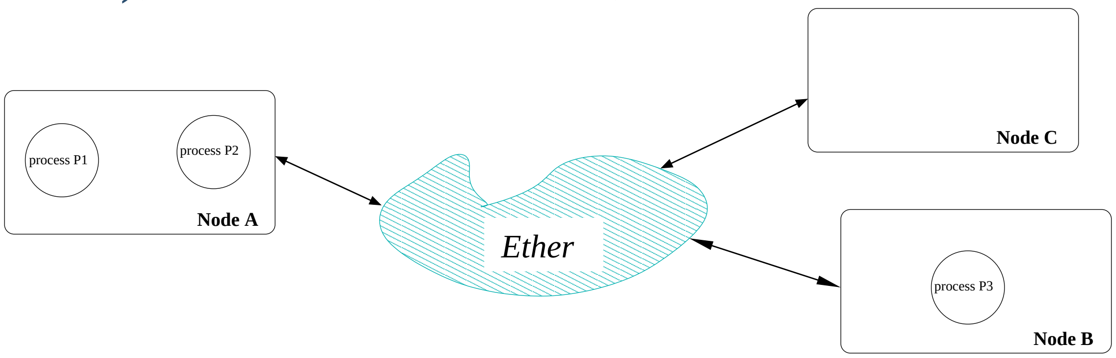
Model checking: 100% coverage

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



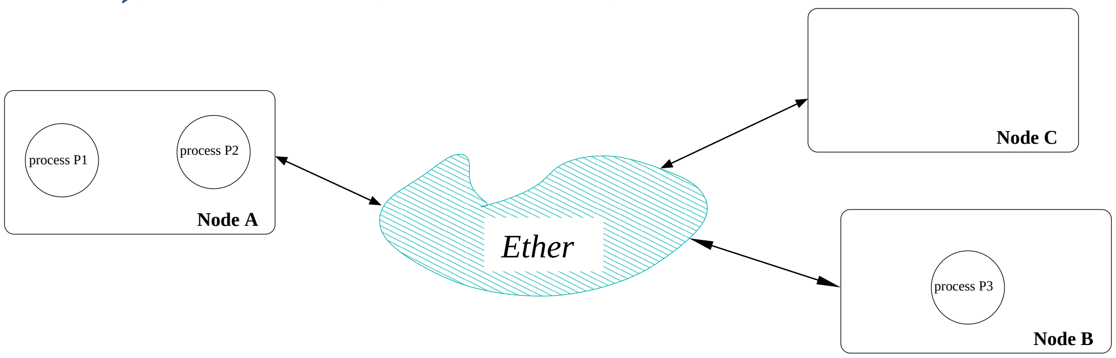
What is the trick? How can we achieve 100% coverage

- 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

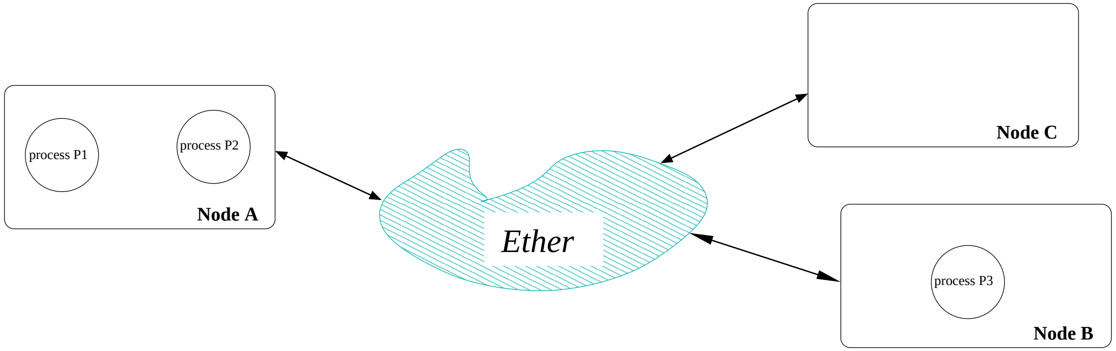
- Needed: the capability to take a **snapshot** of the Erlang system
  - A **program state** is: the contents of all process mailboxes, the state of all running processes, messages in transit (the ether), all nodes, monitors, . . .



- 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, . . .)

Relevancy for non Erlang programmers

The model checker has implications for non-Erlang programmers:

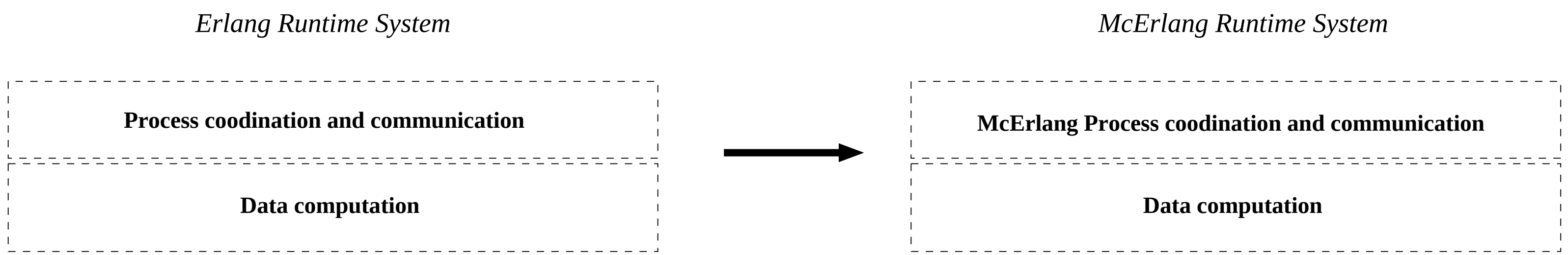
- Erlang is a good *specification* language
- Erlang is a good language for specifying distributed algorithms

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

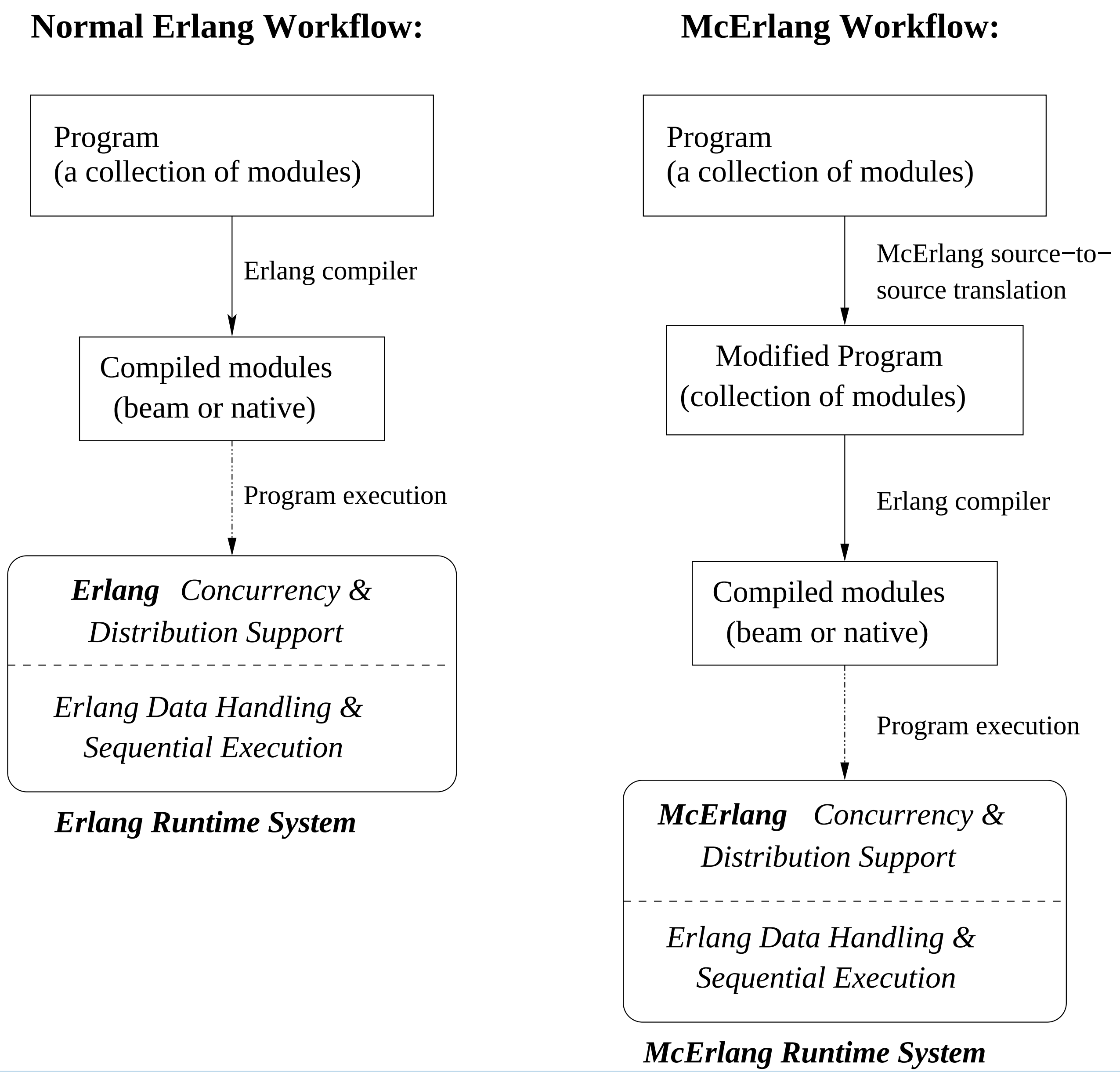
- The lazy solution: just execute the Erlang program to verify in the normal interpreter
- And extract the system state (processes, queues, function contexts) from the Erlang runtime system
- Too messy! We have developed a **new runtime system** for the process part, written in Erlang, and still use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



Recent Improvements to McErlang

- Using a Core Erlang transformation for adapting code to run under McErlang
- Using Hans Svensson's LTL to Büchi automaton translator
- Support for mixing simulation and model checking to reduce state space usage
- SMP model checking algorithms

McErlang Workflow



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}}
```

Adapting code for the new runtime environment

- Translation requires a *global* analysis over all source files in a project, so no `parse_transform`

- Transformation steered by a configuration file:

```
[
 {gen_server,
   [{translated_to,mce_erl_gen_server}]},
 {supervisor,
   [{translated_to,mce_erl_supervisor}]},
 {erlang,
   [{rcv,false}]},
 {{erlang,spawn,4},
   [rcv,{translated_to,{mcerlang,spawn}}]},
 {{erlang,open_port,2},
   [blacklisted]},
 ...
]
```

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

Core Erlang Experiences

- More regular than normal Erlang (saner variable binding) – we use our own “normalized” Core Erlang subset
- **Problem**: not all Core Erlang features handled by compiler
- **Problem**: clause guards – calls to the guard `is_pid` must be replaced to handle McErlang pids `{pid,atom(),int()}`:

```
is_tuple(Pid),
size(Pid)=:=3,
element(1,Pid)=:=pid,
is_integer(element(3,Pid))
```

- McErlang problem:
  detecting calls from non McErlang code to McErlang code, and McErlang code calling non McErlang code
- Overall a positive experience

Correctness Properties

- Correctness properties are expressed in Linear Temporal Logic
- LTL properties are translated automatically to Büchi automata using Hans Svensson's LTL2Buchi translator

Correctness Properties

- Correctness properties are expressed in Linear Temporal Logic
- LTL properties are translated automatically to Büchi automata using Hans Svensson's LTL2Buchi translator

LTL Operators check properties of *program runs*:

- *Always* ϕ

  ϕ holds in all future states of the run

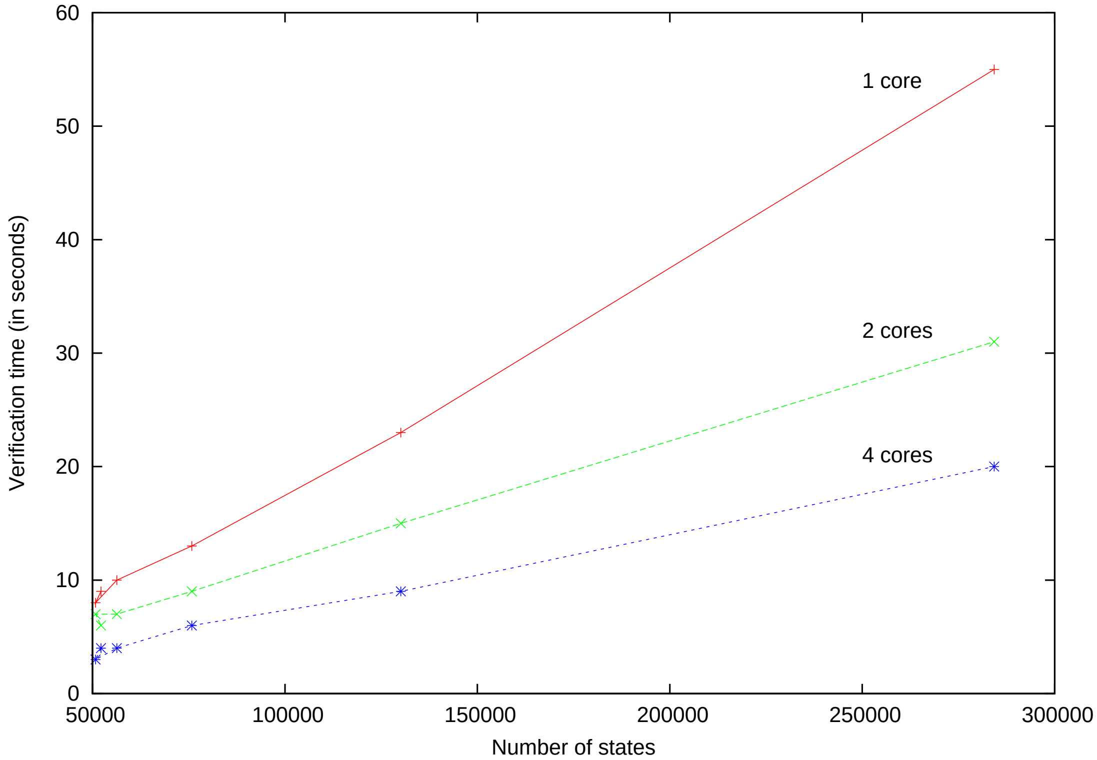
- *Eventually* ϕ, ϕ_1 *Until* ϕ_2, ...
- Predicates on actions or Erlang states:

```
queueOverflow(State,Actions,_) ->
 lists:any
 (fun (P) -> length(P#process.queue) > 5 end,
  mce_erl:allProcesses(State)).
```

  Checks whether any process has a queue of size greater than 5

SMP Algorithms for Model Checking

- **Idea**: make use of the Erlang smp implementation to do model checking on multiple processors and cores
- **Implementation**: use an `ets` table to store the state hash table; use one Erlang process per core to generate new states
- **Experiences** (on Erlang/OTP R12B-5):



Combining Simulation and Model Checking

- **Observation**: most applications use the supervisor behaviour, but trust it

  But when model checking such an app we check the supervisor start up phase too $\Rightarrow$ big state spaces $\Rightarrow$ slow model checking

- **Idea**: Simulate the system until an interesting start state is seen (when the supervisor have "booted" the app processes) – then switch to model checking

- **Experiences**: an app (the "simple messenger") which uses the `supervisor` behaviour:

# users	states (modelcheck)	states (simulation+modelcheck)
2	3324	912
3	179422	56938

Case Study Experience

Simple Messenger: explained in the “Getting started with Erlang” document in the Erlang/OTP documentation

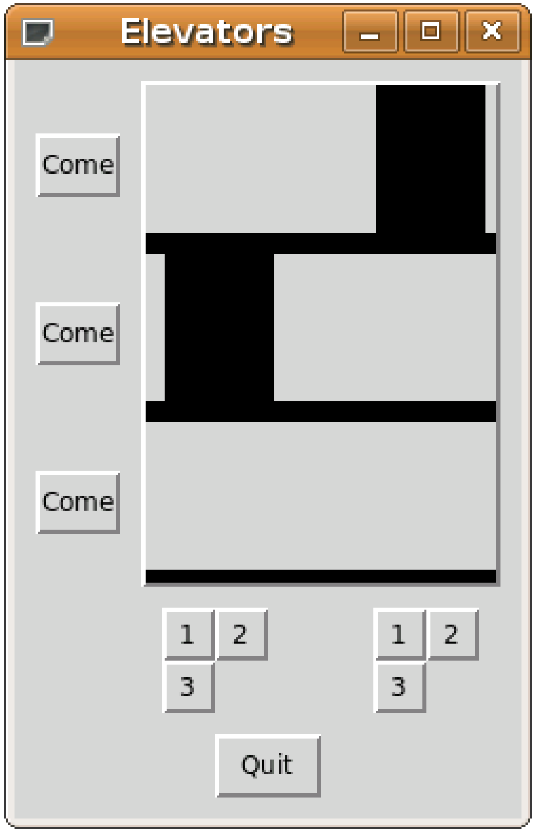
- Lines of code: 176
- API usage: `erlang`, `lists`, `io`
- Lines of code to adapt for modelchecking: 0
- Added code: adding “probe actions” and “probe states” to make program state visible to correctness properties:

```
client(Server_Node, Name) ->
  {messenger, Server_Node} ! {self(), logon, Name}
  await_result(),
  mce_erl:probe(logon,Name), %% Logon probe
  client(Server_Node).
```

Case Study Experience 2

The control software for a set of elevators – used to be part of an Erlang/OTP training course from Ericsson

- API usage: `lists`, `gen_event`, `gen_fsm`, `supervisor`, `timer`, `gs`, `application`
- Static complexity: 1670 loc
- Dynamic complexity: 10 processes (for two elevators)
- Lines of code changed: 15 (decouple graphics from control)



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
  - Run these scenarios in McErlang
  - Evaluate results in QuickCheck
- More info:
  `https://babel.ls.fi.upm.es/trac/McErlang/`