

---

# McErlang: Verifying Distributed Systems Written in Erlang

Lars-Åke Fredlund, Clara Benac Earle

Facultad de Informática, Universidad Politécnica de Madrid

Hans Svensson

IT University Gothenburg

# Context: The ProTest project

- ProTest: property based testing (for Erlang)
- European FP7 STREP project (May 2008–April 2011)
- Partners:
  - ◆ Sheffield (John Derrick, Qiang Guo)
  - ◆ Chalmers (John Hughes, Koen Claessen)
  - ◆ Gothenburg Univ. (Thomas Arts, Hans Svensson)
  - ◆ Kent University (Simon Thompson, Huiqing Li)
  - ◆ Quviq, Ericsson, ETC, LambdaStream
  - ◆ UPM

# ProTest: property based testing

Slogan: write properties and derive tests from those

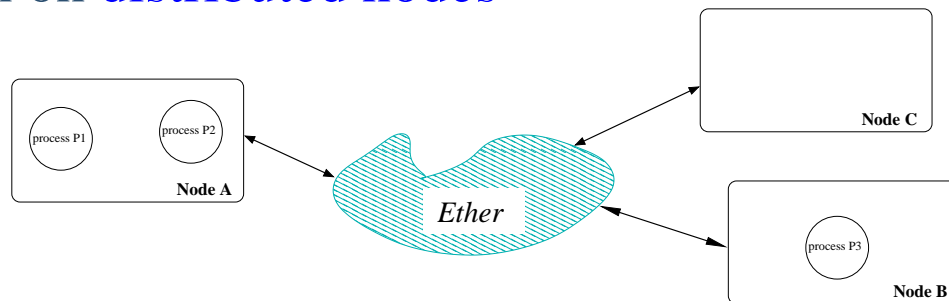
- WP1: QuickCheck/Erlang (random testing)
- WP2: Refactoring of Erlang programs
- WP3: Log analysis, runtime monitoring
- WP4: Concurrent systems (QuickCheck/model checking)

# 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, ...)

# Erlang/OTP language features

- Basis: a general purpose **functional programming language**
- **Lightweight processes** communicating using **message passing**
- The OTP library provides software components: server–client pattern, publish–subscribe pattern, ...
- Processes run on **distributed nodes**



# 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

# Erlang as a specification language

- Decent data types
- Higher-order functions for writing concise specifications
- Concurrency+Distribution
- Has software components for lifting specification abstraction level: server–client pattern, publish–subscribe pattern, ...
- Can treat program code as data (nice meta level)
- Missing non-determinacy: easy to add

```
do  
    e1; ...; en  
od
```

# Erlang for specifying distributed algorithms

Matches common assumptions in the distributed algorithm/web service community:

- **Isolated** processes that communicate using message-passing
- **Fault-tolerance** by using unreliable failure detectors
- Process **fairness** built into the language
- **Locality** is important: stronger communication guarantees for intra-node communication than inter-node communication

# Model Checking Programs

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

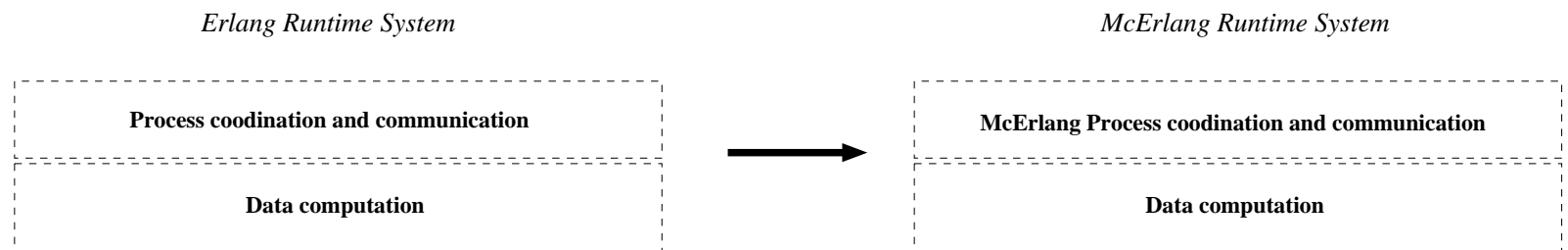
- Compute transitions of a state  $s$ :  
 $s \xrightarrow{\alpha} s'$  for all states  $s'$  and actions  $\alpha$
- Compare program states for equality ( $s \equiv s'$ ), to detect recurring states
- Inspect states or actions to determine whether they violate the correctness property being checked

# 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

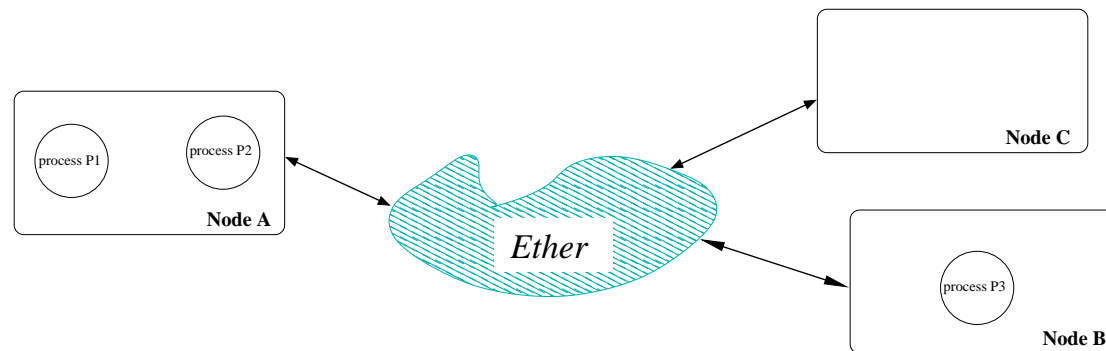
- The lazy solution: just execute the Erlang program to verify in the normal interpreter
- And extract the system state (processes, queues, function contexts) from the Erlang runtime system
- Too messy! We have developed a **new runtime system** for the process part, and still use the old runtime system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



# Erlang System State in New Runtime System

Much of the complexity of the model checker lies in accurately programming the new runtime system:

- A *process* executes some expression, has a pid, and a queue (mailbox) of incoming messages
- A *node* is a collection of running processes, a register that gives symbolic names to process identifiers (pids), and a set of links between processes (to handle fault tolerance)
- A *program state* is a collection of nodes and messages in transit between nodes (ether)



# Adapting code for the new runtime environment

Erlang code must be “recompiled” in order to run under the new runtime system:

- API changes: **spawn**  $\Rightarrow$  **evOS:spawn**
- Calls to functions which executes **receive** (for retrieving a message from the process mailbox) will block

```
receive
```

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

```
end
```

Instead of executing **receive** a recompiled function returns a special Erlang value describing the receive request:

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

# Receive Example

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

becomes

```
f(Pid) -> {recv, {fun f_0/2, [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}.
```

# Operational Semantics

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

- Transitions occur between *stable states* by choosing a ready process to run
- In a stable runtime state all processes are *stable*, i.e.:
  - ◆ waiting in a receive statement
  - ◆ or it having just been spawned
- The *initial state* is given by a single process executing a given function
- *Actions* are a sequence of side effects (e.g., message passing) between stable states
- A big-step interleaving semantics

# Consequences of Transition Semantics

- **Side effect free functions** are executed by normal Erlang interpreter (fast, but a dedicated model checker is faster)

# Consequences of Transition Semantics

- **Side effect free functions** are executed by normal Erlang interpreter (fast, but a dedicated model checker is faster)
- **Side effect functions** (message passing) are executed by new API functions (destructive state updates are slow in Erlang)

# Consequences of Transition Semantics

- **Side effect free functions** are executed by normal Erlang interpreter (fast, but a dedicated model checker is faster)
- **Side effect functions** (message passing) are executed by new API functions (destructive state updates are slow in Erlang)
- We can easily inspect the global system state

# Consequences of Transition Semantics

- **Side effect free functions** are executed by normal Erlang interpreter (fast, but a dedicated model checker is faster)
- **Side effect functions** (message passing) are executed by new API functions (destructive state updates are slow in Erlang)
- We can easily inspect the global system state
- Checking for state equality is easy (Erlang equality “==”)

# Consequences of Transition Semantics

- **Side effect free functions** are executed by normal Erlang interpreter (fast, but a dedicated model checker is faster)
- **Side effect functions** (message passing) are executed by new API functions (destructive state updates are slow in Erlang)
- We can easily inspect the global system state
- Checking for state equality is easy (Erlang equality “==”)
- We have great power over the execution of a system: we can kill processes randomly, we can break communication links, bring down nodes, restart nodes...

# 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!

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 (using an on-the-fly algorithm)
- The monitor can inspect the current state, and the actions leading to the current state
- The monitor either returns a new monitor state (success), or signals an error (full LTL checking also available)

# Modularity of McErlang

Parametric wrt. a number of modules:

- Source language (Erlang, WS-CDL)
- Algorithm – (safety, liveness, simulation, smp variants)
- Abstraction
- State table implementation
- Stack implementation
- Scheduler

# Abstractions

Maps a concrete program state to a reduced one (before checking for recurrent states)

- For reducing the checked state space
- General abstractions:  $programstate \rightarrow integer$   
(for use in bitspace algorithms)
- Specialized ones: abstracting queue contents...
- Not guaranteed safe

## Performance example: early data

A resource locker+clients example

(x=client asking exclusive access, s=shared)

configuration	states	time (1 core)	time (4 cores)
XXXXX	52197	13s	7s
XXXXS	50805	12s	5s
XXXSS	56313	13s	6s
XXSSS	75801	18s	8s
XSSSS	130101	31s	13s
SSSSS	284277	70s	33s

# 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!
- For the future:
  - ◆ Use a less coarse transition semantics; break the execution for every side effect (e.g., sends, spawns)
  - ◆ Experiment with partial-order reductions
- Downloading:  
<http://babel.ls.fi.upm.es/~fred/McErlang>