Finding Concurrency Errors using **Concuerror**

Kostis Sagonas

RELEASE

Outline

- Context of this work & Motivation
- **Concuerror**: Systematic testing tool for Erlang
  - High-level description
  - Demo
  - Implementation technology
  - Blocking avoidance & Preemption bounding
  - More demos
  - Evaluation & Experience
- Related testing tools
- Concluding remarks

Erlang

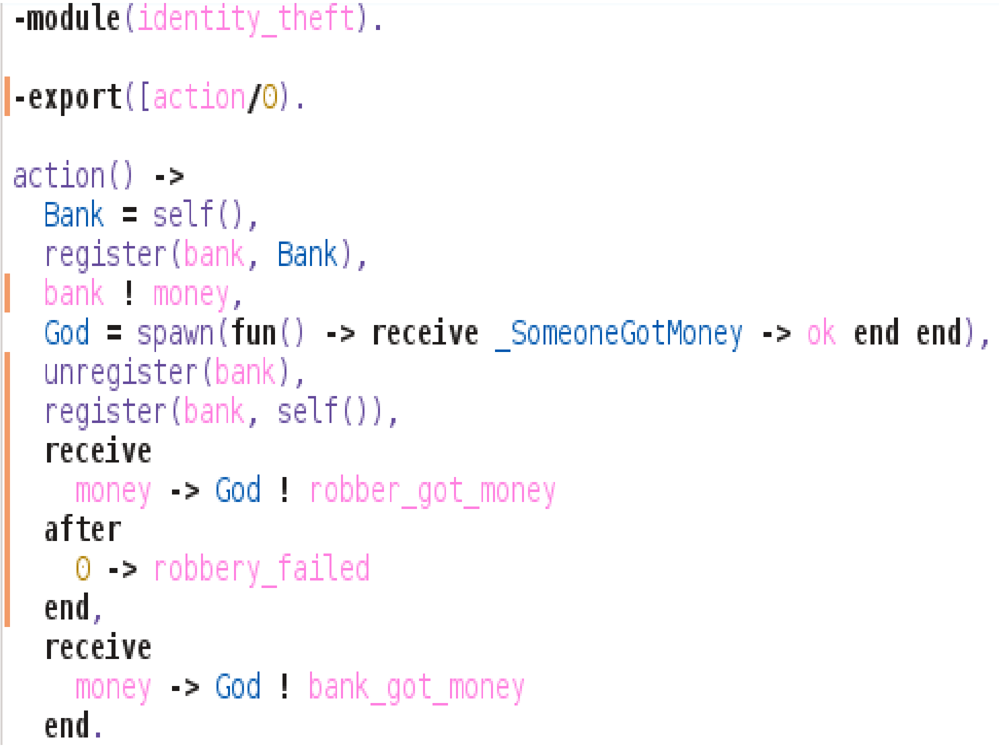
- Concurrent functional programming language
- Implements the actor model of concurrency
  - lightweight processes (“green threads”)
  - communicating via asynchronous message passing
  - selective receive
  - conceptually no shared memory between processes

- Erlang’s implementation
  - built-ins that manipulate shared memory

    e.g. process registry, ETS tables, etc.

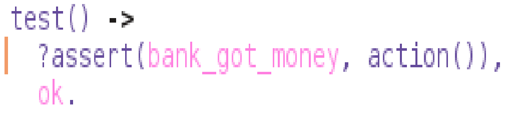
Motivation

Program



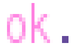
```
-module(identity_theft).

-export([action/0]).

action() ->
  Bank = self(),
  register(bank, Bank),
  bank ! money,
  God = spawn(fun() -> receive _SomeoneGotMoney -> ok end end),
  unregister(bank),
  register(bank, self()),
  receive
    money -> God ! robber_got_money
  after
    0 -> robbery_failed
  end,
  receive
    money -> God ! bank_got_money
  end.
```

Test



```
test() ->
  ?assert(bank_got_money, action()),
  ok.
```

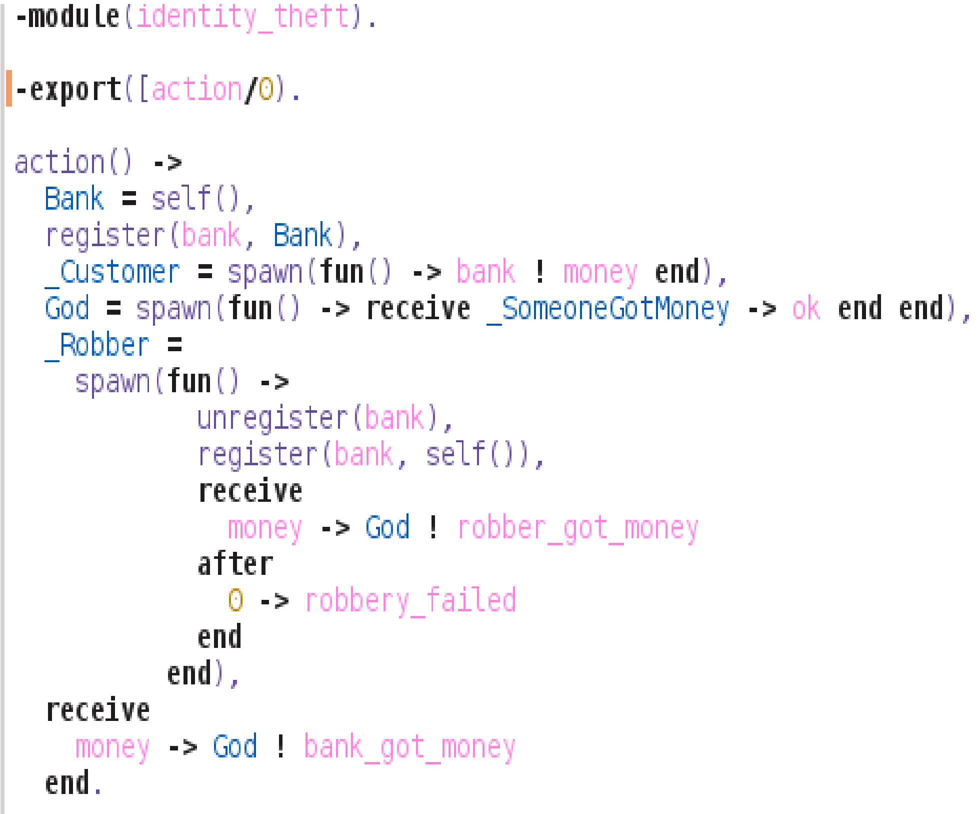
Test Result



```
ok.
```

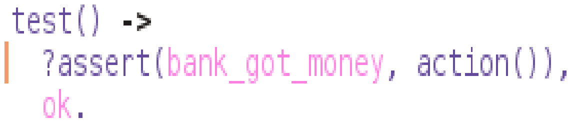
Motivation

Concurrent Program



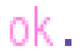
```
-module(identity_theft).

-export([action/0]).

action() ->
  Bank = self(),
  register(bank, Bank),
  _Customer = spawn(fun() -> bank ! money end),
  God = spawn(fun() -> receive _SomeoneGotMoney -> ok end end),
  _Robber =
    spawn(fun() ->
            unregister(bank),
            register(bank, self()),
            receive
              money -> God ! robber_got_money
            after
              0 -> robbery_failed
            end
          end),
  receive
    money -> God ! bank_got_money
  end.
```

Test



```
test() ->
  ?assert(bank_got_money, action()),
  ok.
```

Test Result?



```
ok.
```

Concurrent programming is **HARD**

- Concurrent execution is difficult to reason about and get right (even for experts!)
- Rare process interleaving results in bugs that are
  - hard to anticipate
  - difficult to find, reproduce, and debug (“Heisenbugs”)
  - hard to be sure whether they are really fixed
- Big productivity problem: it can waste significant developers’ time and resources
- This work focuses on *systematic testing*
  - aka *stateless model checking*

Comparison of approaches

	Model Checking	Static Analysis	Systematic Testing
Scalability	+	++	++
Precision	+	+	++
Coverage	++	++	+
Generality	++	+	++

[Taken from CHESS tutorial]

Erlang program and its unit test

```
-module(ping_pong).
-export([pong/0]).

pong() ->
  Self = self(),
  Pid = spawn(fun() -> ping(Self) end),
  register(?MODULE, Pid),
  receive ping -> ok end.

ping(P) ->
  P ! ping.
```

```
-module(ping_pong_test).
-export([test/0]).

test() ->
  ok = ping_pong:pong().
```

Error discovered by Concuerror

```
Checked 5 interleaving(s). 1 error found.

Error type : Exception
Details    : {badarg,[{erlang,register,[ping_pong,<...>],[]},
                      ...
  Process P1 spawns process P1.1
  Process P1.1 sends message `ping` to process P1
  Process P1.1 exits (normal)
  Process P1 registers process P1.1 (dead) as `ping_pong`
  Process P1 exits ("Exception")
```

Concuerror in a nutshell

- A tool for systematic testing (aka stateless model checking) of concurrent Erlang programs

- Given a program and its test suite Concuerror systematically explores process interleaving and presents detailed interleaving information about any errors that occur during the execution of these tests

Concuerror in a nutshell

- Takes control of the scheduler and runs a function (usually a test) to detect whether its execution results in the following errors
  - Process crashes and abnormal termination
  - Assertion violations
  - “Deadlocks”: lack of progress for processes
- Totally automatic
  - Explores all “interesting” interleaving sequences ...
  - … possibly up to a preemption bound ...
  - … and by employing some very clever algorithms

Concuerror's properties

- Easy to use
- Scalable
  - Applicable to “real-world” programs
- Precise
  - Any error found is possible to occur
  - Does not introduce new behaviors
- Sound
  - All concurrency errors (for a test) can be found
  - Aims to capture all scheduling non-determinism
  - Exhaustively explores this non-determinism

Scheduling non-determinism

[Taken from CHESS tutorial]

Thread 1

```
x = 1;
y = 1;
```

Thread 2

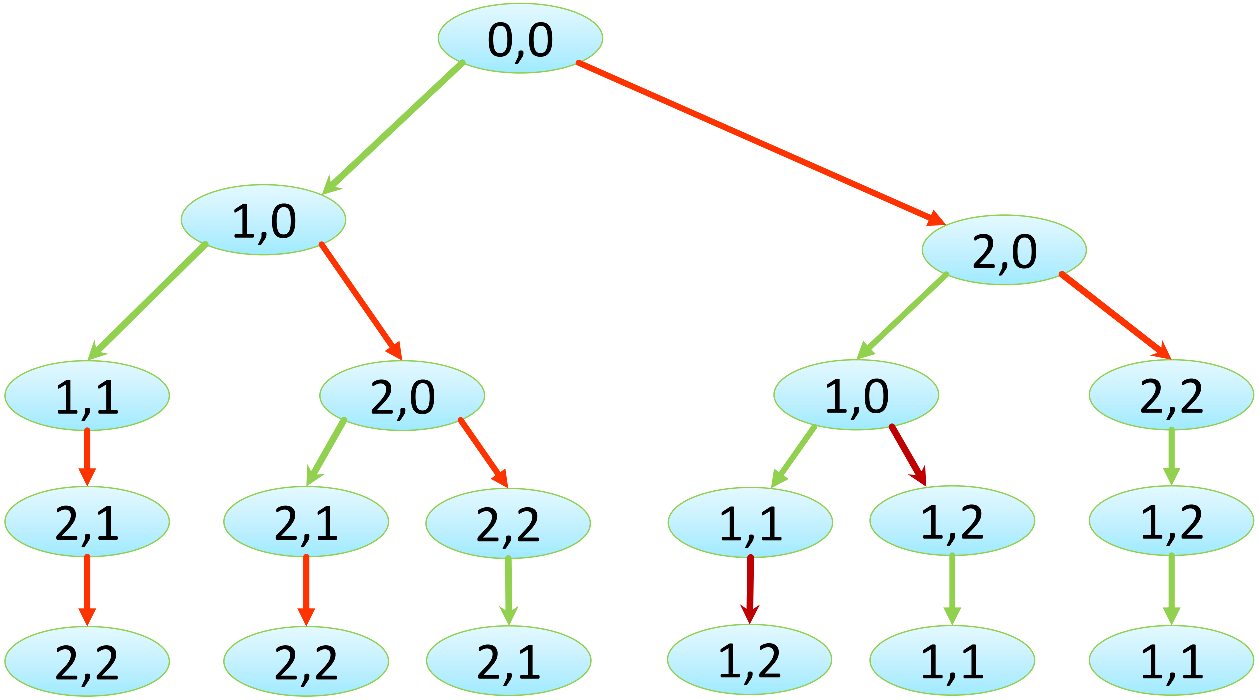
```
x = 2;
y = 2;
```

x = 1;

y = 1;

x = 2;

y = 2;



Sources of non-determinism

- Scheduling non-determinism
  - Interleaving non-determinism
    - Processes can race to access shared resources
    - Processes can be preempted at arbitrary points
  - Timing non-determinism
    - Sleeping processes can wake up at any point
    - Timers can fire in arbitrary points/orders
- Input non-determinism
  - Programs can be used in a variety of ways
  - Non-deterministic system calls (e.g. **`random()`**)
- Memory model effects

Concuerror's anatomy

- GUI
- Instrumenter
- Scheduler
- “Replaying” machinery

Concuerror's instrumentation (vsn 0.9)

```
pause() ->
  receive scheduler_prompt -> ok end.
```

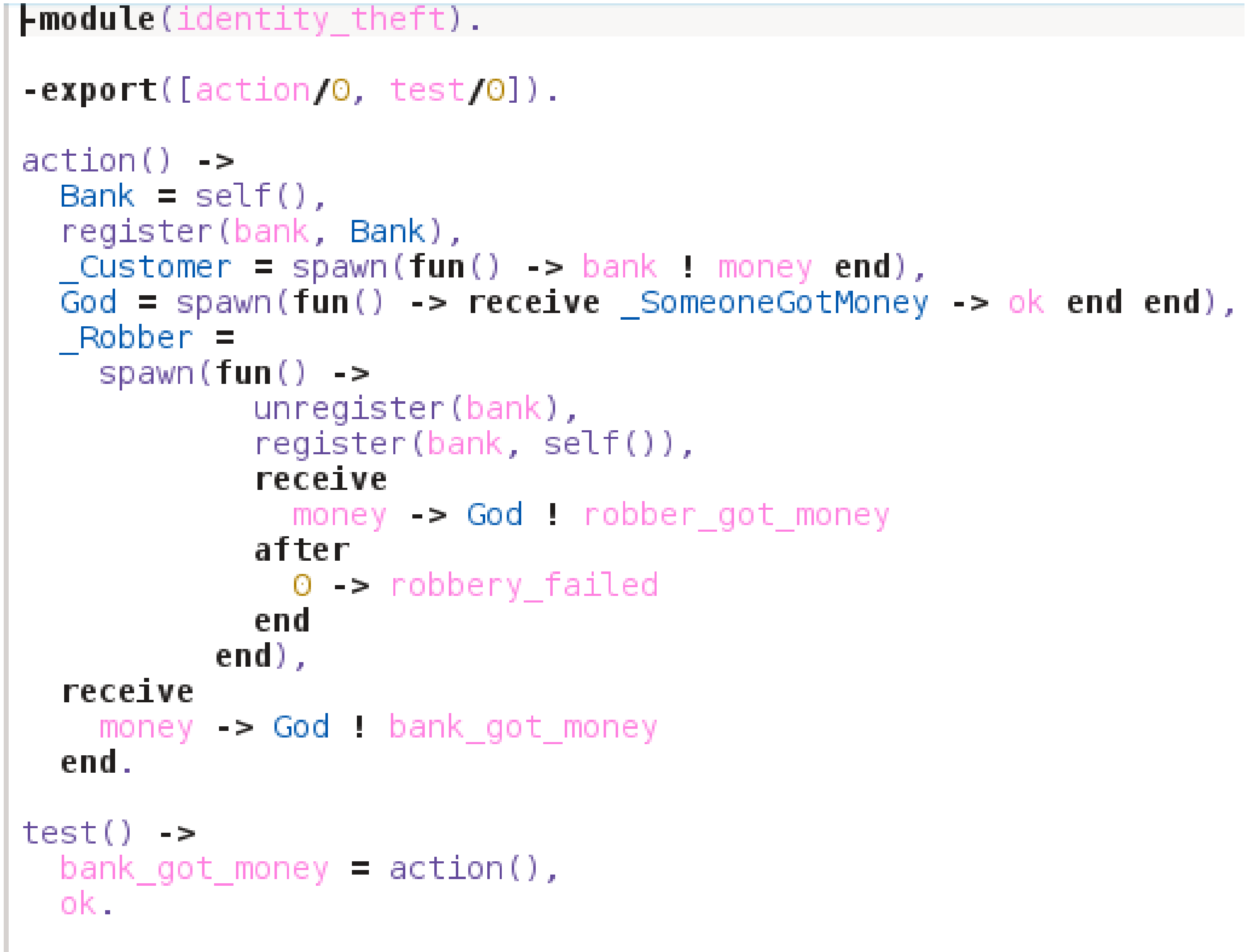
```
spawn_wrapper(F) ->
  Fun = fun() -> pause(), F() end,
  Pid = spawn(Fun),
  notify_scheduler(spawn, Pid),
  Pid.
```

```
send_wrapper(Dest, Msg) ->
  Dest ! ?INSTR_MSG(Msg),
  notify_scheduler(send, {Dest,Msg}),
  pause(),
  Msg.
```

Process scheduling

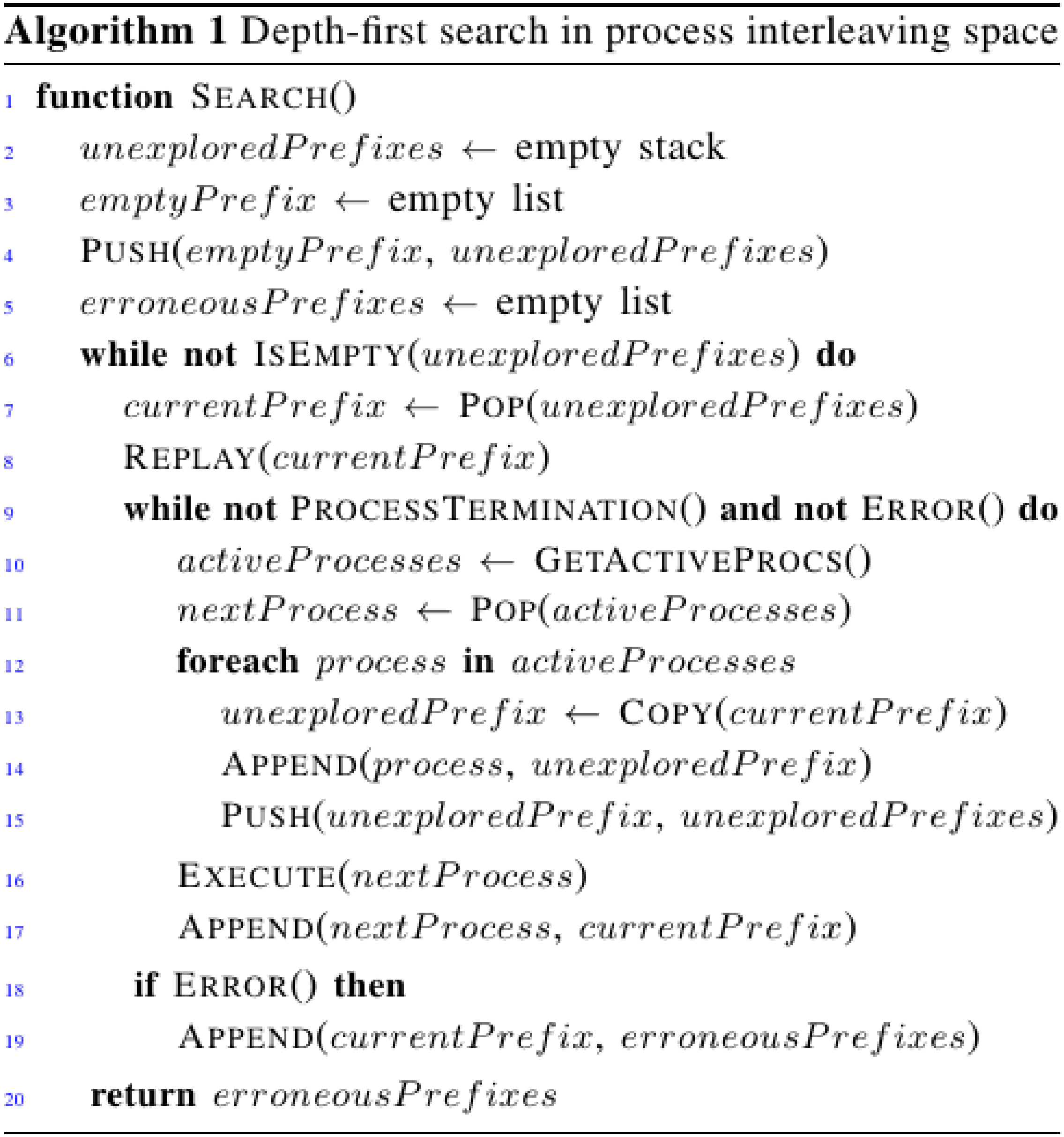
- Each process is assigned a logical identifier (LID)
  - that uniquely identifies the process
- Interleaving sequences are
  - represented as sequences of LIDs
  - explored using depth-first search
- For n processes with k preemption points each, the number of interleaving sequences is exponential in both n & k
- Space complexity is $O(n^2k)$

Another example



```
-module(identity_theft).

-export([action/0, test/0]).

action() ->
  Bank = self(),
  register(bank, Bank),
  _Customer = spawn(fun() -> bank ! money end),
  God = spawn(fun() -> receive _SomeoneGotMoney -> ok end end),
  _Robber =
    spawn(fun() ->
                unregister(bank),
                register(bank, self()),
                receive
                  money -> God ! robber_got_money
                after
                  0 -> robbery_failed
                end
            end),
  receive
    money -> God ! bank_got_money
  end.

test() ->
  bank_got_money = action(),
  ok.
```

Concuerror's search strategy



Algorithm 1 Depth-first search in process interleaving space

```
function SEARCH()
    unexploredPrefixes ← empty stack
    emptyPrefix ← empty list
    PUSH(emptyPrefix, unexploredPrefixes)
    erroneousPrefixes ← empty list
    while not ISEMPTY(unexploredPrefixes) do
        currentPrefix ← POP(unexploredPrefixes)
        REPLAY(currentPrefix)
        while not PROCESSTERMINATION() and not ERROR() do
            activeProcesses ← GETACTIVEPROCS()
            nextProcess ← POP(activeProcesses)
            foreach process in activeProcesses
                unexploredPrefix ← COPY(currentPrefix)
                APPEND(process, unexploredPrefix)
                PUSH(unexploredPrefix, unexploredPrefixes)
            EXECUTE(nextProcess)
            APPEND(nextProcess, currentPrefix)
        if ERROR() then
            APPEND(currentPrefix, erroneousPrefixes)
    return erroneousPrefixes
```

Efficiency improvements

1. Blocking avoidance
2. Preemption bounding

Blocking avoidance

- A process executing a **receive** statement with no matching messages in its mailbox blocks
- Becomes active again only when a matching message arrives
- Although checking a process mailbox interacts with the shared state, it does not update it
- Interleaving sequences that will result in process blocks are redundant and can be soundly ignored

We call this optimization *blocking avoidance*

Preemption bounding

- Idea similar to iterative context bounding [Musuvathi & Qadeer 2007]
- Builds on the hypothesis that most concurrency errors involve a small number of context switches
- Eliminates exponential dependence on k

Preemption bounding

- Context bounding adapted to message passing
- Takes into account
  - process blocks in receives
  - process exits

Exploration with preemption bound = 1

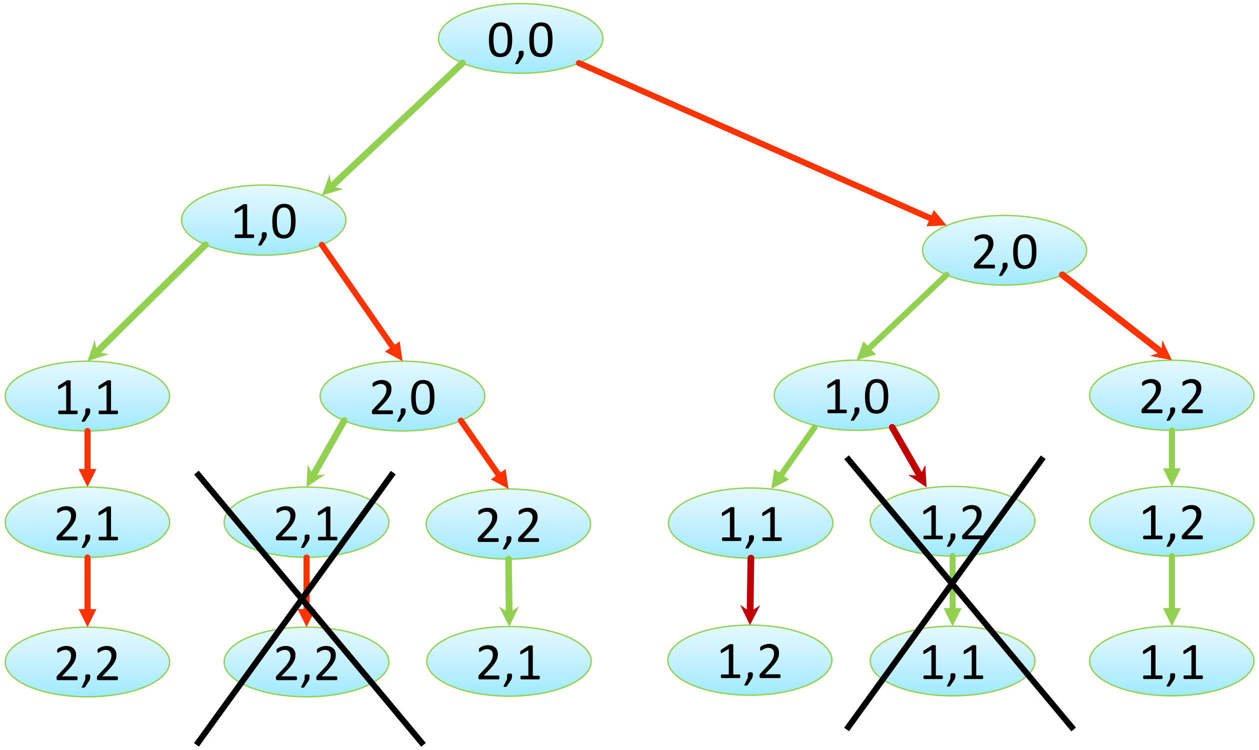
[Adapted from CHESS tutorial]

Thread 1

```
x = 1;
y = 1;
```

Thread 2

```
x = 2;
y = 2;
```



Evaluation & Experience

- Applied Concuerror to some large code bases
- One example: code of **Dialyzer**
  - Static analyzer for Erlang programs
  - About 28,000 LOC
  - Aggressively parallelized
- On a relatively simple test, Concuerror reported various interleaving sequences with a stuck server process, i.e. a resource leak

Evaluation & Experience

- Applied Concuerror to some large code bases

- Another example: code of **mochiweb**
  - Erlang library for building lightweight HTTP servers
  - About 12,000 LOC (including the test code)
  - Cleanly written code & (extensive?) test suite
- One (serious?) bug found
  - Using a **`cast`** instead of a **`call`** to stop the socket server (for **`mochiweb_socket_server:stop/0`**)
  - Confirmed by developer; fixed end of May 2013

Concuerror’s options

```
usage: concuerror [<args>]
Arguments:
 -t|--target module      Run eunit tests for this module
 -t|--target module function [args]
                         Specify the function to execute
 -f|--files  modules     Specify the files (modules) to instrument
 -o|--output file        Specify the output file (default results.txt)
 -p|--preb   number|inf  Set preemption bound (default is 2)
 -I  include_dir         Pass the include_dir to concuerror
 -D  name=value          Define a macro
 --noprogress            Disable progress bar
 -q|--quiet              Disable logging (implies --noprogress)
 -v                      Verbose [use twice to be more verbose]
 --fail-uninstrumented   Fail if there are uninstrumented modules
 --ignore   modules      It’s OK for these modules to be uninstrumented
 --show-output           Allow program under test to print to stdout
 --wait-messages         Wait for uninstrumented messages to arrive
 --app-controller        Start an (instrumented) application controller
 -T|--ignore-timeout bound
                         Treat big after Timeouts as infinity timeouts
 --gui                   Run concuerror with a graphical interface
 --dpor                  Runs the experimental optimal DPOR version
 --help                  Show this help message
```

Related testing tools

- **CHESS** from Microsoft Research [Musuvathi et al.]
  - Similarities:
    - systematic testing tool for finding concurrency errors
    - iterative context bounding
  - Difference: uses platform-dependent wrappers
- **VeriSoft** [Godefroid]
- **Erlang QuickCheck/PULSE** [Claessen et al.]
- **McErlang** [Fredlund and Svensson]

Future work

- Parallelize Concuerror's exploration engine
- Investigate the interaction between PropEr (a property-based testing tool) and Concuerror
- Test suite minimization

Concluding remarks

- Conventional testing, e.g. unit testing, is not able to expose concurrency errors
- Using Concuerror not only allows us to see that our tests pass, but also *guarantees* that the programs are robust and correct w.r.t. these tests
- In practice, a small preemption bound is enough to reveal most concurrency-related defects
  - Start with a small preemption bound and gradually increase
- Exponential increase with number of processes
  - Write tests for small # of processes and generalize
- Concuerror provides detailed explanation about errors

Thanks to the Concuerror developers

Alkis Gotovos

Maria Christakis

Stavros Aronis

Ilias Tsitsimpis