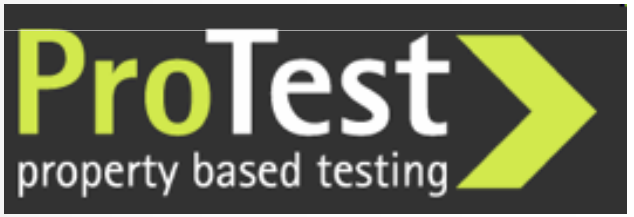
Erlang Training and Consulting Ltd

Erlang Programming for Multi-core

QCON London, March 12th, 2009



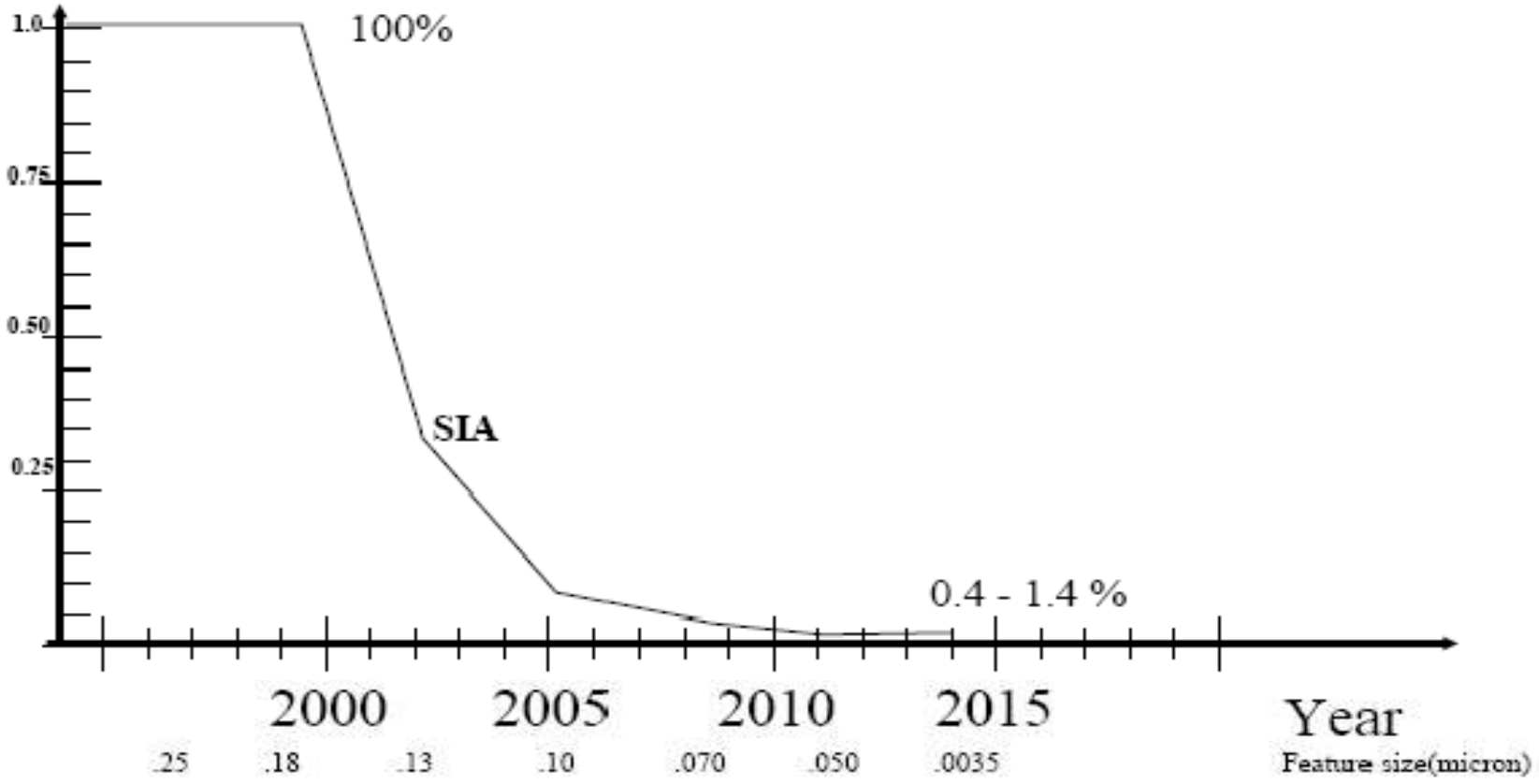
http://www.protest-project.eu/

Ulf Wiger

ulf.wiger@erlang-consulting.com

The age of multicore

Fraction of chip reachable within one clock cycle



[source] Erik Hagersten (http://www.sics.se/files/projects/multicore/day2007/ErikHintro.pdf)

Industry's dilemma

- The shift to multicore is inevitable.
- Parallelizing legacy C (and Java) code is very hard.
- Debugging parallelized C (and Java) is even harder.
- "...but what choice do we have?"

Erlang

Erlang SMP ”Credo”

- SMP should be transparent to the programmer in much the same way as distribution is,
  - I.e. you shouldn’t have to think about it
  - ...but sometimes you must
- Use SMP mainly for stuff that you’d make concurrent anyway
- Erlang uses concurrency as a structuring principle
  - Model for the natural concurrency in your problem

Erlang

Programming Examples

Traditional implementation of lists:map/2:

```
map(F, L) ->
    [F(X) || X <- L].
```

(Simple) SMP implementation: pmap/2:

```
pmap(F, L) ->
    Parent = self(),
    Pids = [spawn(fun() ->
                      Parent ! {self(),F(X)}
                  end) || X <- L],
    [receive {Pid,Res} -> Res end || Pid <- Pids].
```

Not quite the same semantics...

Order: preserved
Exceptions: hangs

Erlang

Trying to fix pmap...

Catches errors and includes them in the map result:

```
pmap1(F, L) ->
    await1(spawn_jobs1(F,L)).  % trivial refactoring

spawn_jobs1(F, L) ->
    Parent = self(),
    [spawn(fun() ->
                Parent ! {self(), catch F(X)}
           end) || X <- L].

await1(Pids) ->
    [receive {Pid,Res} -> Res end || Pid <- Pids].
```

catch Expr :: Value | {'EXIT', Reason}

...but we're now returning exceptions as 'normal' values to the user!

Erlang

Still trying to fix pmap...

Catches first error, terminates rest and raises exception:

```
pmap2(F, L) -> await2(spawn_jobs2(F, L)).

spawn_jobs(F, L) ->
    Parent = self(),
    [spawn(fun(X) -> Parent ! {self(),catch {ok,F(X)}}
     || X <- L].

await2([H|T]) ->
    receive
       {H, {ok, Res}} -> [Res | await2(T)];
       {H, {'EXIT',_} = Err} ->
           [exit(Pid,kill) || Pid <- T],
           [receive {P,_} -> d_ after 0 -> i_ end || P <- T],
           erlang:error(Err)
    end;
await2([]) -> [].
```

Erlang

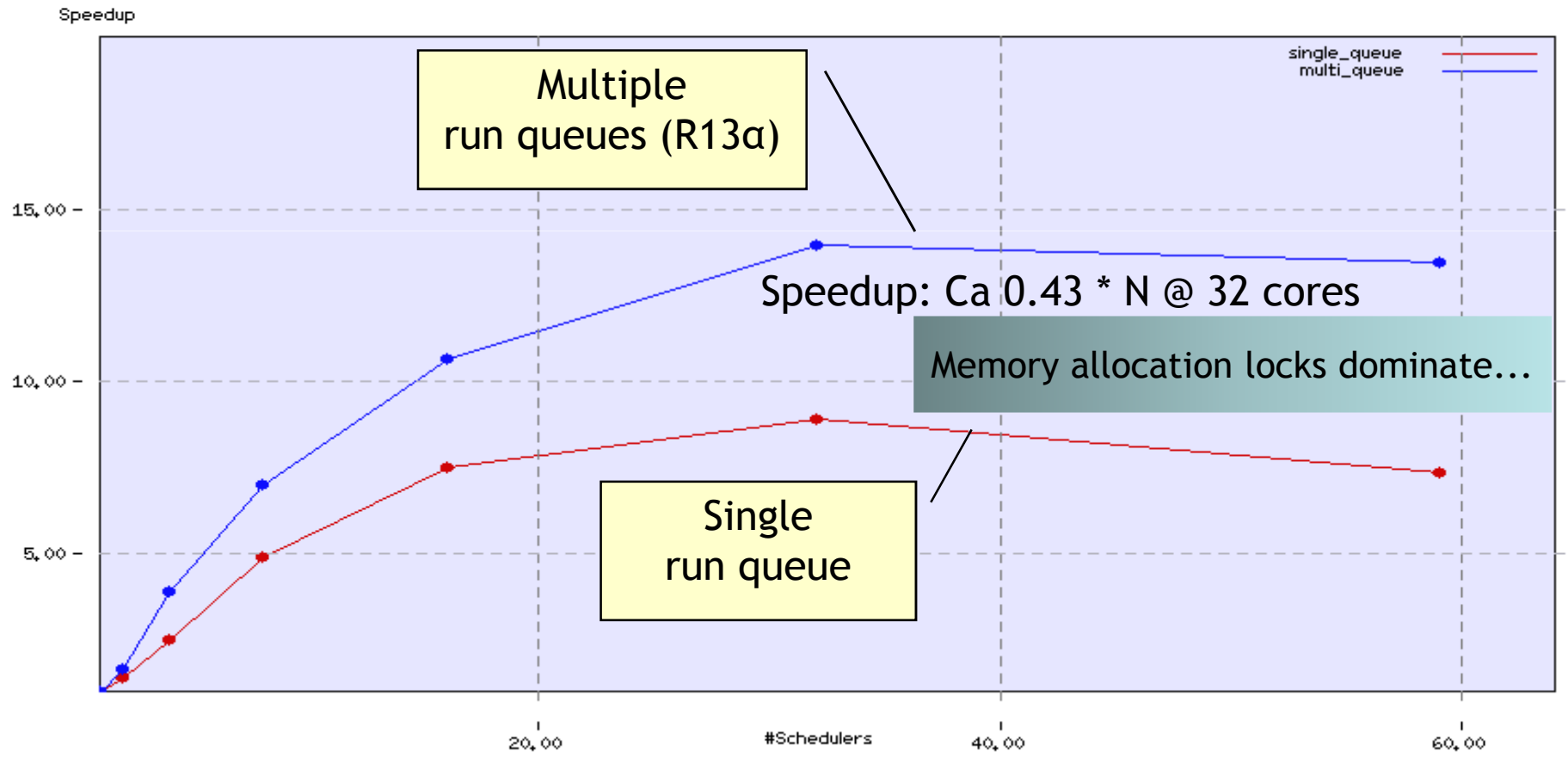
Pmap discussion

- Extra overhead for each function application
  - (spawn + msg send + msg receive + process exit)
- Erlang processes are reasonably lightweight
  - But not primarily intended for data parallelism
  - Measure to see if granularity is suitable
- Exception handling semantics force tradeoffs
- Preserving order is easy, but has a cost
- ...but map is inherently sequential!
- Use SMP mainly for naturally concurrent patterns

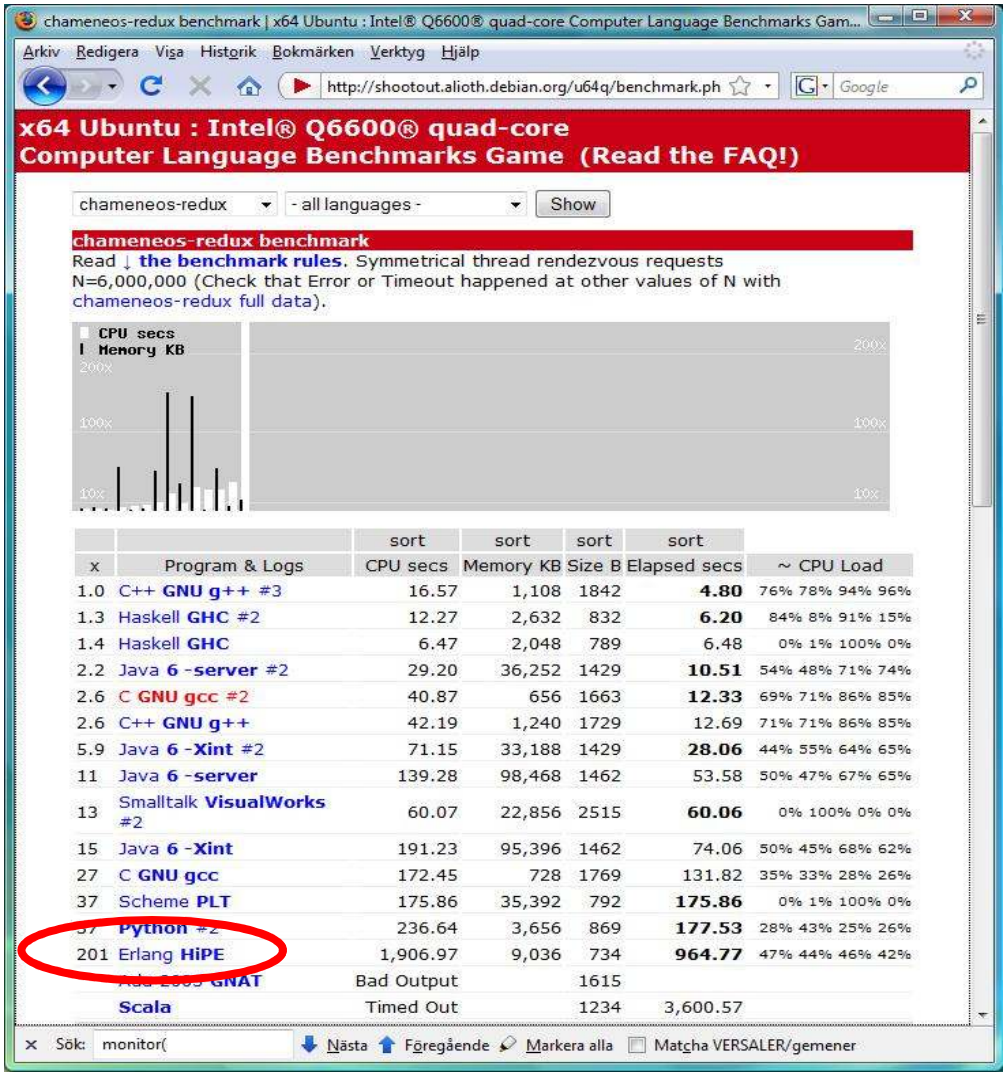
Erlang

Some benchmarks

- Speedup of ”Big Bang” on a Tilera Tile64 chip (R13α)
  - 1000 processes, all talking to each other



A really bad benchmark result



Chameneos Redux
in The Shootout

Gets worse, the more cores are added...

Essentially a rendezvous benchmark

Successful entries:

- C++ (pthread_spinlock)
- Haskell (Mvar – spinlocks Haskell-style)
- Java (synchronized + busy loop wait)

Now used as an ”evil reference” when optimizing the Erlang VM

The €2,500,000* Question

How do you test and debug your program on multicore?

* The total budget of the ProTest EU project

Current situation

- No shared memory → (mostly) Goodness
- SMP a bit similar to distributed programs → (mostly) Goodness
- Selective message reception → simpler state machines
- Great tracing facilities. This helps in SMP also

- But this is not nearly good enough!

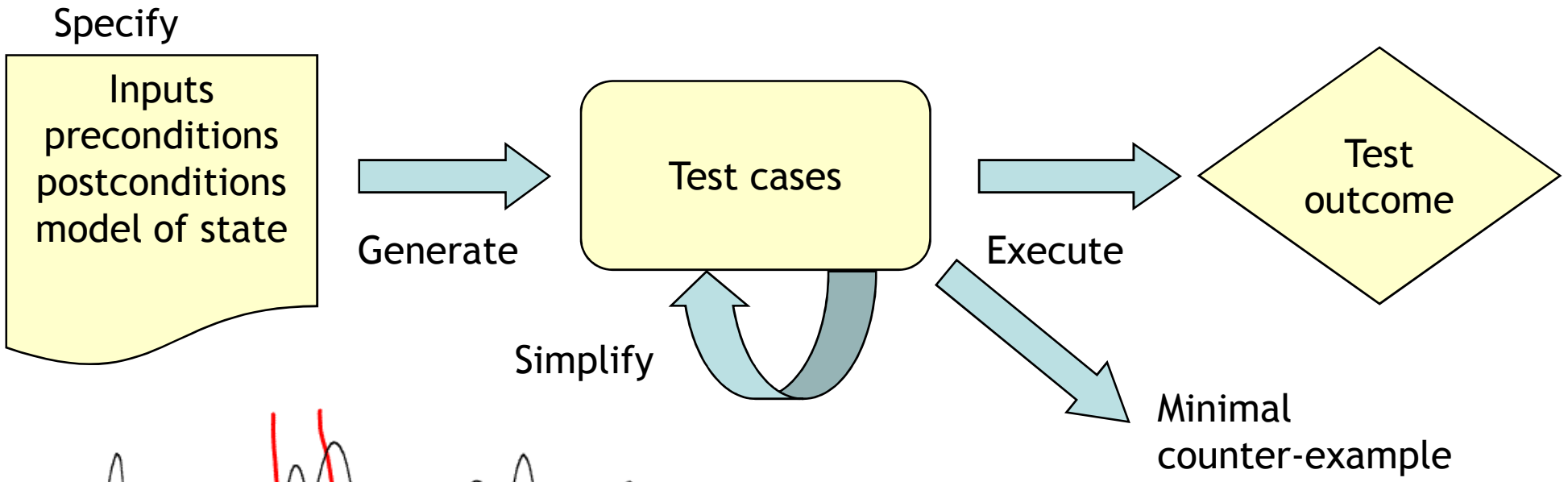
Erlang

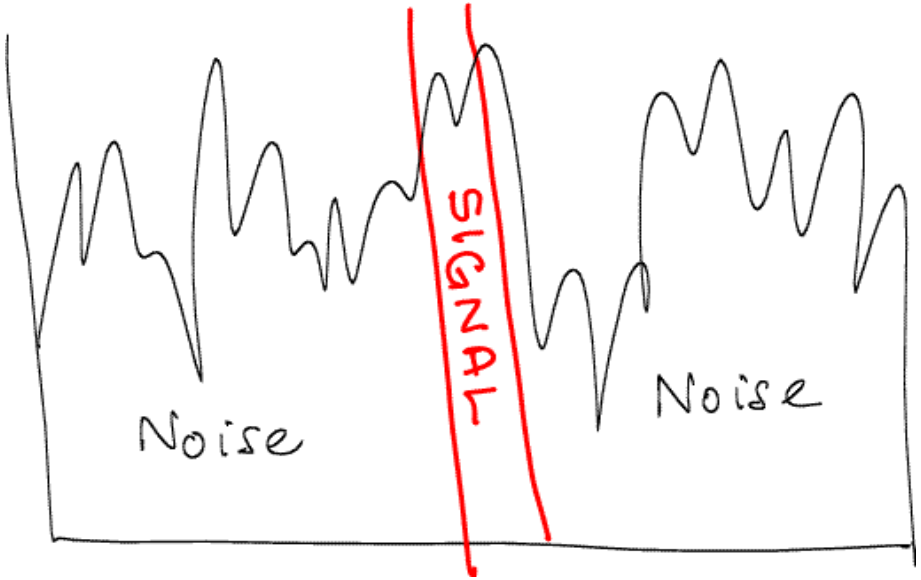
Hypothesis

- Timing-related bugs are often triggered only when running large systems
- But this is mainly because timing is different in large systems, compared to unit test
- If we can control timing aspects in unit test, many concurrency bugs can be found by running small examples

Erlang

Introducing QuickCheck





Erlang

QuickCheck example

A simple property

```
prop_lists_delete() ->
  ?FORALL(I, int(),
    ?FORALL(List, list(int()),
          not lists:member(
                I, lists:delete(I,List)))).
```

Test run

```
1> eqc:quickcheck(example:prop_lists_delete()).
.......................................
Failed! after 42 tests
-8
[5,-13,-8,-8,-9]
Shrinking..............(16 times)
-8
[-8,-8]
false
```


Erlang

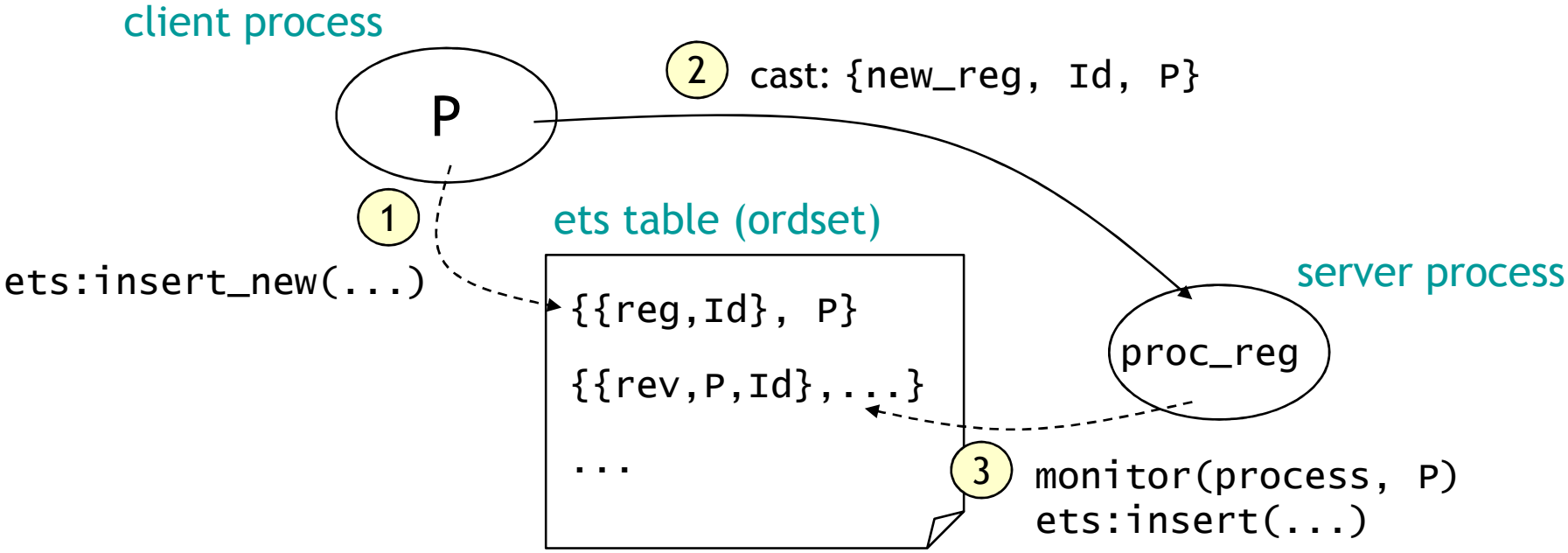
Case study: proc_reg

- Extended process registry for Erlang*
- Early (2005 at Ericsson) prototype using 'clever' optimizations
- An experiment with QuickCheck triggered a bug, but we couldn't diagnose it
- So I discarded that code. The gen_server-based replacement has now been used in the field for 3 years.
- We tried again with a new experimental QuickCheck (Part of the ProTest EU project)

* Wiger: "Extended Process Registry for Erlang" ACM SIGPLAN Erlang Workshop 2007

Erlang

Short intro to proc_reg



- Key API:

```
reg(Id, Pid) -> true  | fail()
unreg(Id)    -> true  | fail()
where(Id)    -> pid() | undefined
```


Erlang

Proc_reg and QuickCheck

```
%% Command generator, S is the state. Format is {call, Module, Function, Args}.
command(S) ->
    oneof([{call, ?MODULE, spawn, []}] ++
        [{call, ?MODULE, kill,  [elements(S#st.pids)]} || S#st.pids/=[]] ++
        [{call, ?MODULE, reg,   [name(),elements(S#st.pids)]} || S#st.pids/=[]] ++
        [{call, ?MODULE, unreg, [name()]}] ++
        [{call,proc_reg, where, [name()]}]
        ).
```

```
prop_proc_reg() ->
    ?FORALL(Cmds, commands(?MODULE),
            ?TRAPEXIT(
            begin
                {ok,Environment} = setup_proc_reg(),
                {H,S,Res} = run_commands(?MODULE,Cmds),
                cleanup_proc_reg(Environment),
                ?WHENFAIL(
                   io:format("History: ~p\nState: ~p\nRes: ~p\n",[H,S,Res]),
                   Res == ok)
            end)).
```

This property works all the time.

Erlang

QuickCheck post-conditions

```
postcondition(S, {call,_,where, [Name]}, Res) ->
    Res == proplists:get_value(Name, S#state.regs);
postcondition(S, {call,_,unreg, [Name]}, Res) ->
    case Res of
        true -> unregister_ok(S, Name);
        {'EXIT',_} ->
            not unregister_ok(S, Name)
    end;
postcondition(S, {call,_,reg, [Name, Pid]}, Res) ->
    case Res of
        true -> register_ok(S, Name, Pid);
        {'EXIT',_} ->
            not register_ok(S, Name, Pid)
    end;
postcondition(_S, {call,_,_,_}, _Res) ->
    true.

unregister_ok(S, Name) ->
    lists:keymember(Name, 1, S#state.regs).

register_ok(S, Name, Pid) ->
    not lists:keymember(Name, 1, S#state.regs).
```

Surprisingly few

This is the heart of the specification

Erlang

Parallelizing the property

```
prop_parallel() ->
    ?FORALL(PCmds = {_,{_ACmds, _BCmds}}, eqc_par_statem:pcommands(?MODULE),
            ?ALWAYS(5,
            begin
                {ok,Environment} = setup_proc_reg(),
                {H, AB, Res} = eqc_par_statem:run_pcommands(?MODULE, PCmds),
                cleanup_proc_reg(Environment),
                ?WHENFAIL(
                   io:format("Sequential: ~p\nParallel: ~p\nRes: ~p\n",[H,AB,Res]),
                   Res == ok)
            end)).
```

- QuickCheck tries different ways to parallelize some commands without violating preconditions
- The ALWAYS operator re-runs each test to check ”timing stability”
- The property fails if there is no possible interleaving of the parallel commands that satisfy the postconditions
- Shrinking is not deterministic, but surprisingly effective...

Erlang

Using a custom scheduler

```
prop_scheduled(Verbose) ->
    ?FORALL(PCmds = {_,{_ACmds,_BCmds}}, eqc_par_statem:pcommands(?MODULE),
    ?ALWAYS(5,
    ?FORALL(Seed, seed(),
       begin
           L = scheduler:start([{seed, Seed}, {verbose, Verbose}],
                 fun() ->
                         {ok,Environment} = setup_proc_reg(),
                         {H,AB,Res} = eqc_par_statem:run_pcommands(?MODULE, PCmds),
                         cleanup_proc_reg(Environment),
                         {H,AB,Res}
                 end),
           delete_tables(),
           {H, AB = {A,B}, Res} = proplists:get_value(result,L),
           ?WHENFAIL(
             ..., Res == ok)
       end))).
```

- The code under test must first be instrumented
- The scheduler controls important scheduling events

Erlang

Bug # 1

```
{[{set,{var,6},{call,proc_reg_eqc,spawn,[]}},
  {set,{var,8},{call,proc_reg_eqc,kill,[{var,6}]}}],
 {[{set,{var,11},{call,proc_reg_eqc,reg,[a,{var,6}]}}],
  [{set,{var,12},{call,proc_reg_eqc,reg,[a,{var,6}]}}]}}
{2829189918,7603131136,617056688}
Sequential: [{{state,[],[],[]},<0.10470.0>},
             {{state,[<0.10470.0>],[],[]},ok},
             {{state,[<0.10470.0>],[],[<0.10470.0>]},ok}]
Parallel: {[{{call,proc_reg_eqc,reg,[a,<0.10470.0>]},
             {'EXIT',{badarg,[{proc_reg,reg,2},
                              {proc_reg_eqc,reg,2},
                              {parallel2,run,2},
                              {parallel2,'-run_pcommands/3-fun-0-',3}]}}}],
           [{{call,proc_reg_eqc,reg,[a,<0.10470.0>]},true}]}
Res: no_possible_interleaving
```

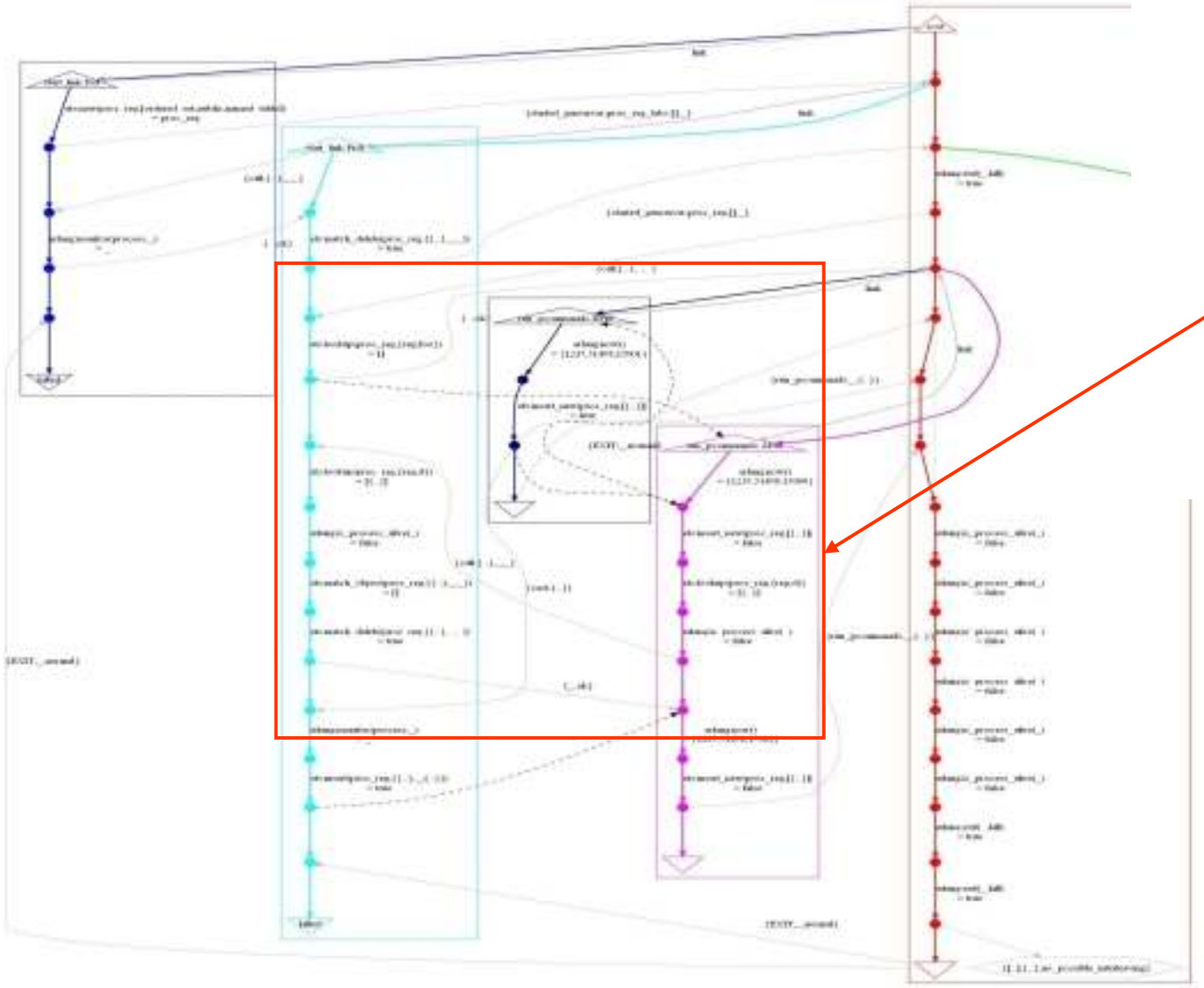
"Sequential prefix"

Parallel component

- This violates the specification
  - Registering a dead process should always return 'true'
- Absolutely minimal counter-example, but hard to understand
- QuickCheck can also output a graph (as a .dot file)

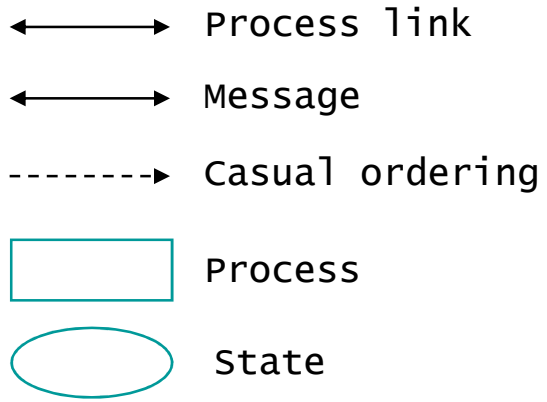
Erlang

Generated state chart



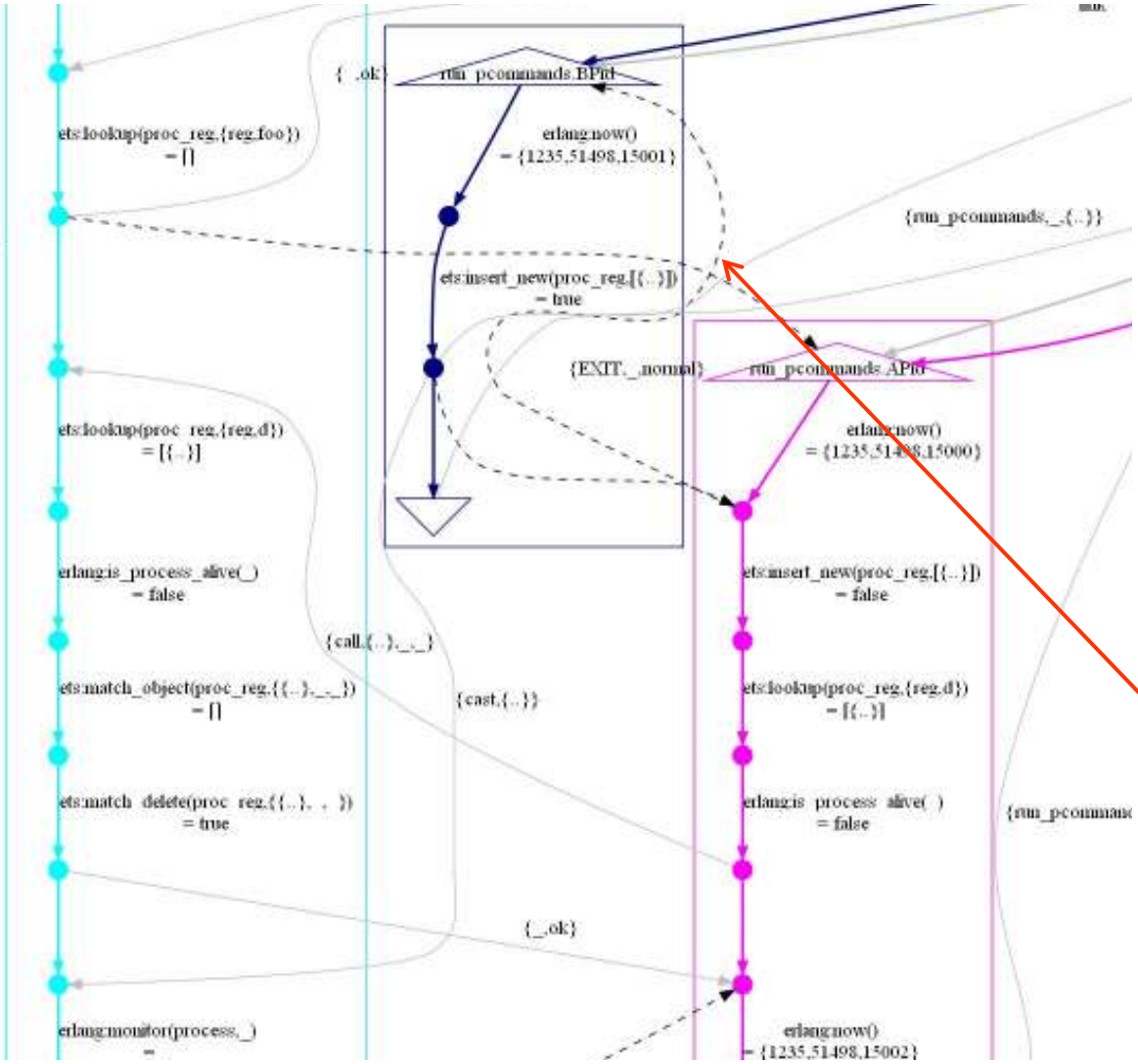
QuickCheck-generated .dot file, visualized with GraphViz

We'll zoom in on the interesting part



Erlang

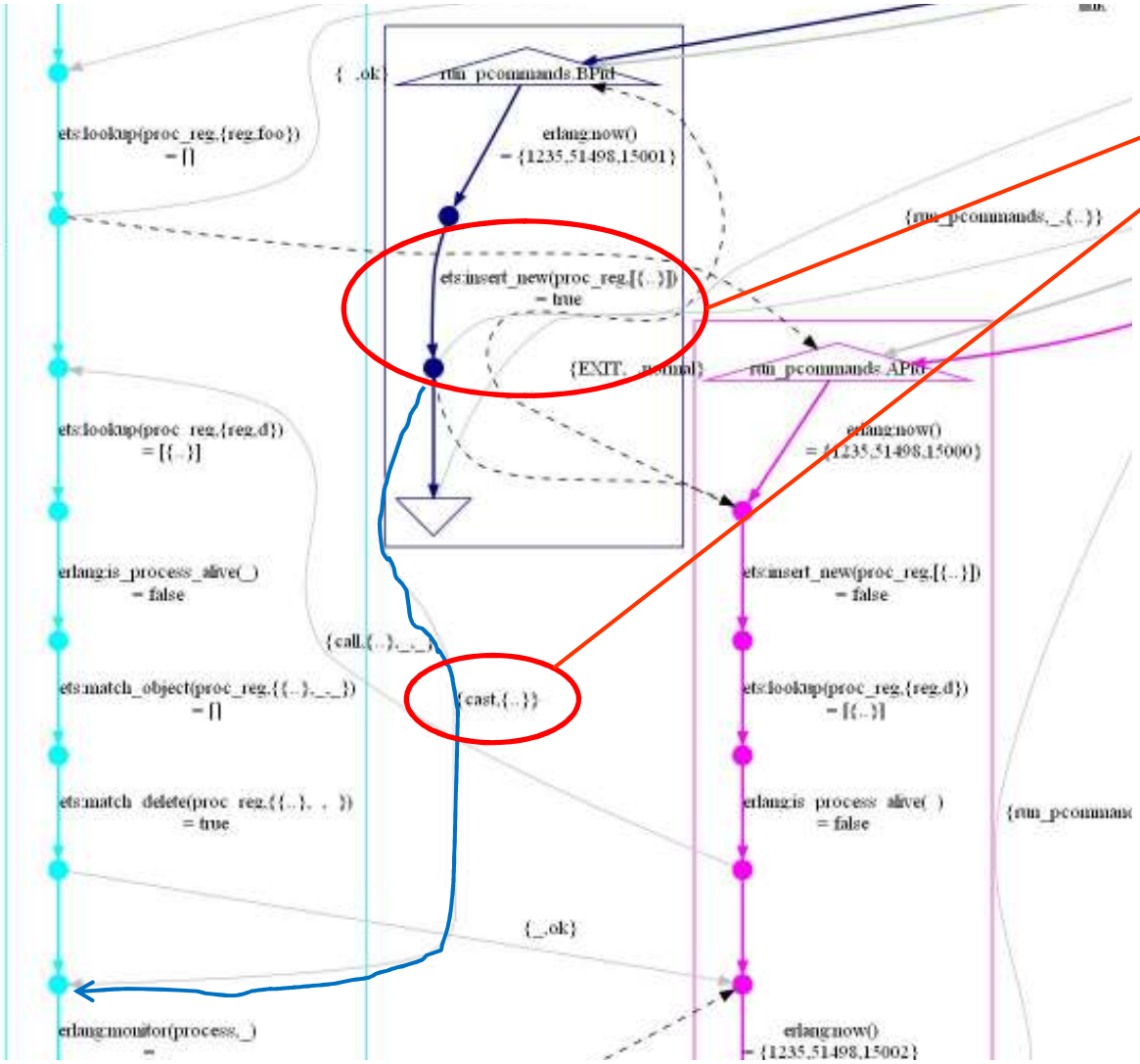
State chart detail



Dotted arrows indicate a non-obvious ordering (usually a sign of trouble)

Erlang

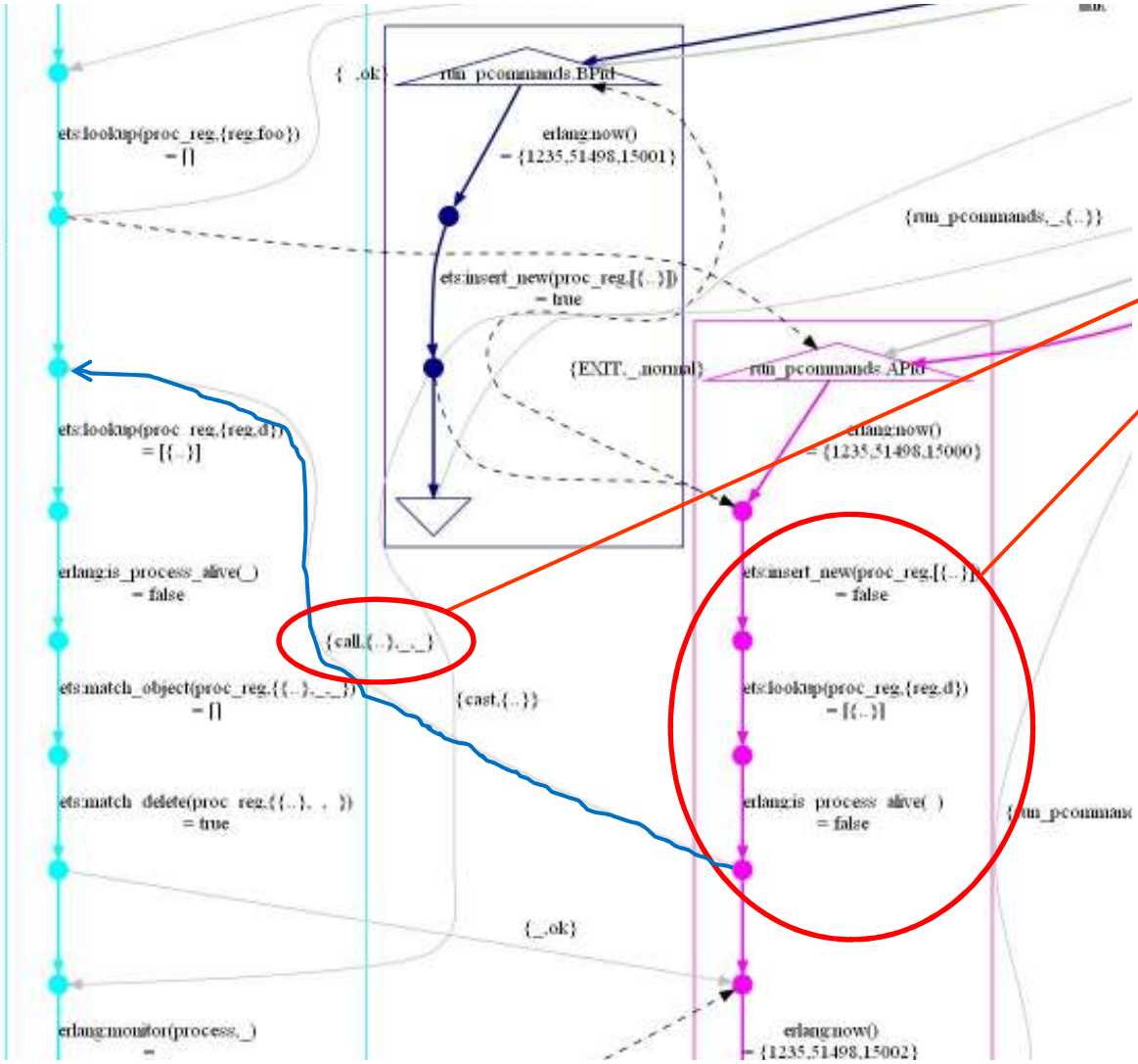
State chart detail



BPid's registration succeeds, and it sends a cast to the admin server

Dotted arrows indicate a non-deterministic ordering (usually a sign of trouble)

State chart detail

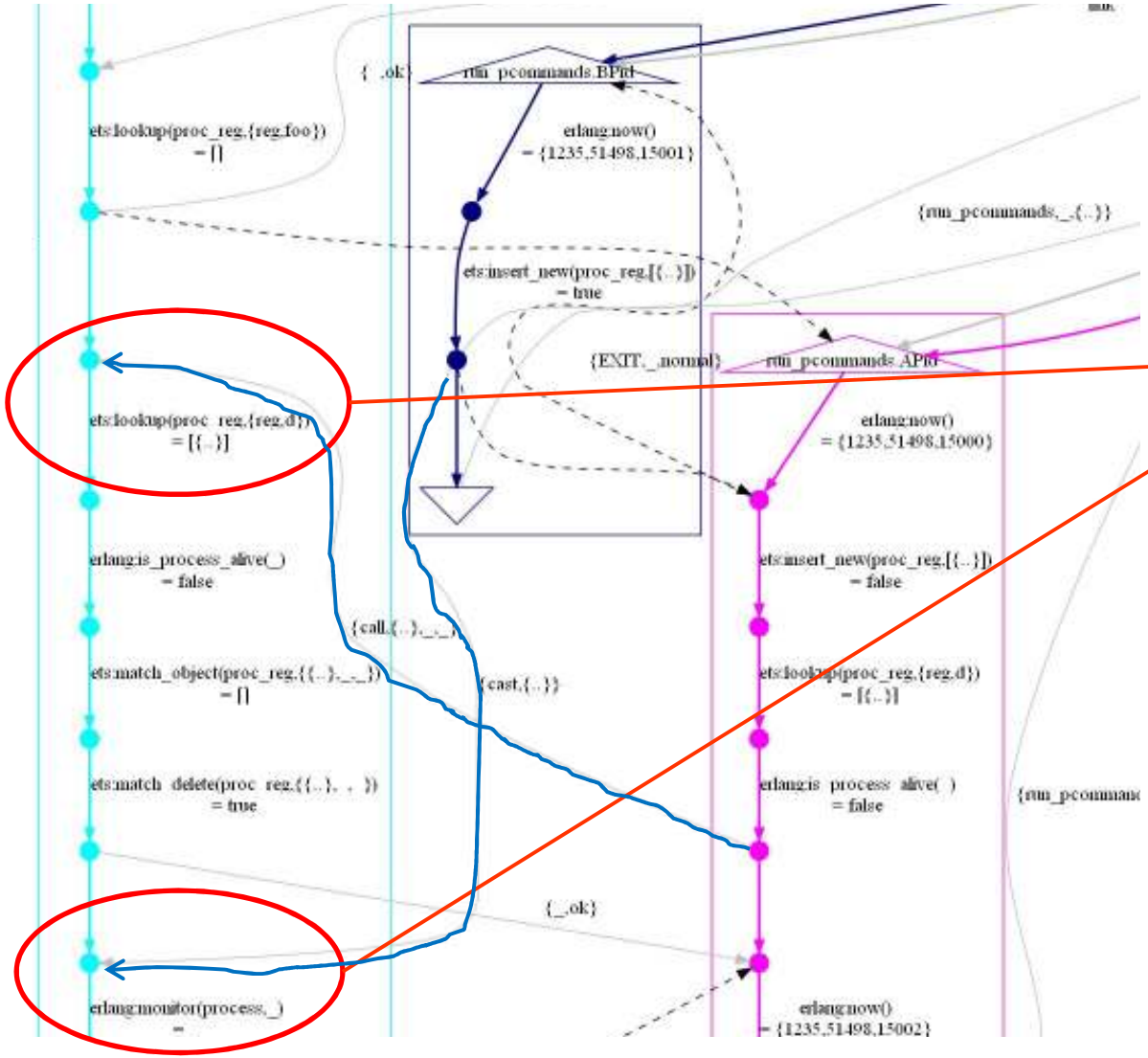


BPid's registration succeeds, and it sends a cast to the admin server

APid finds that the name is taken, but by a dead process – it asks the server for an audit (the call)

Dotted arrows indicate a non-deterministic ordering (usually a sign of trouble)

State chart detail



BPid's registration succeeds, and it sends a cast to the admin server

APid finds that the name is taken, but by a dead process – it asks the server for an audit (the call)

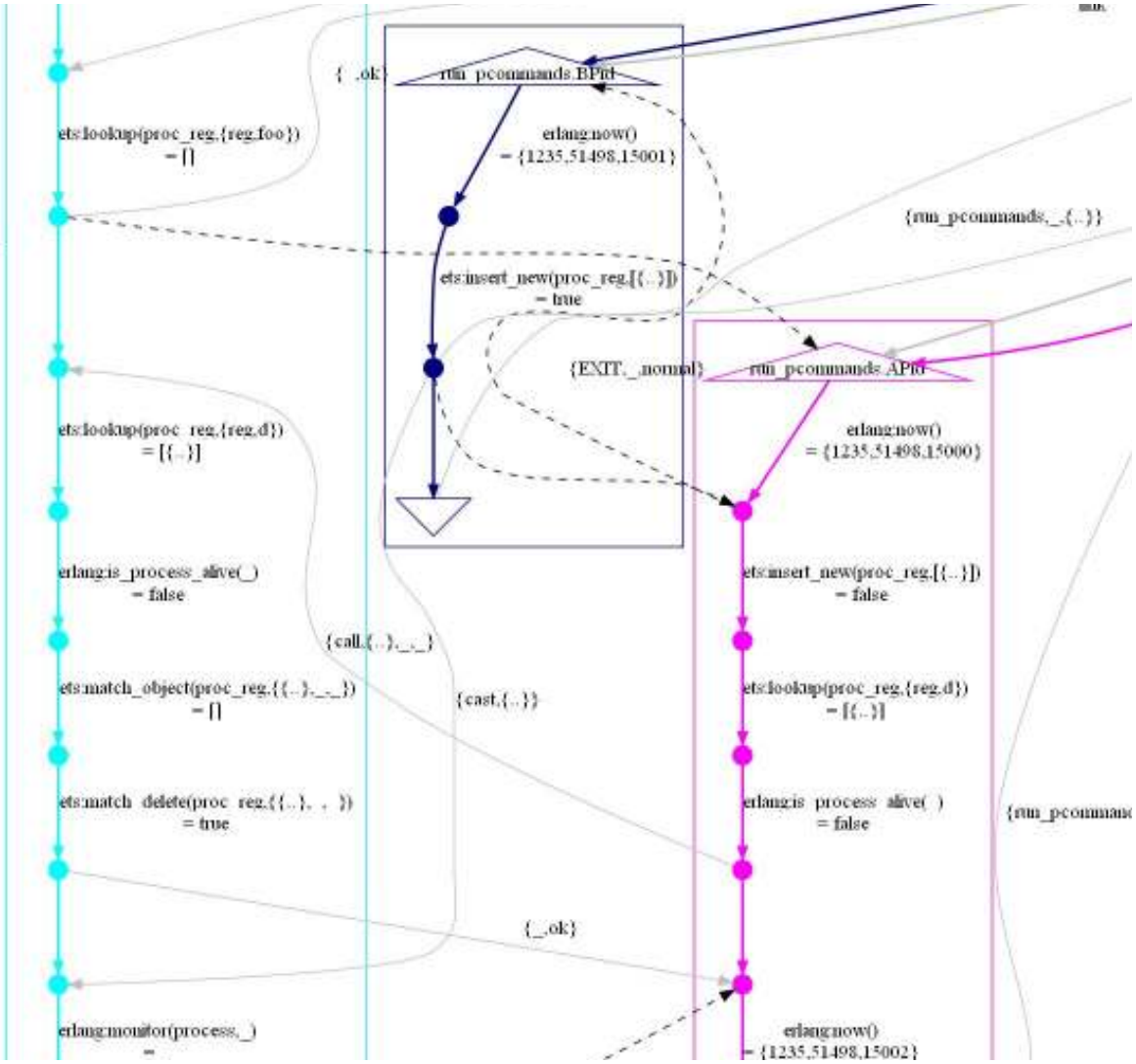
But the call reaches the server before the cast

(true preemption, or delayed message delivery can cause this)

Dotted arrows indicate a non-deterministic ordering (usually a sign of trouble)

Erlang

State chart detail



BPid's registration succeeds, and it sends a cast to the admin server

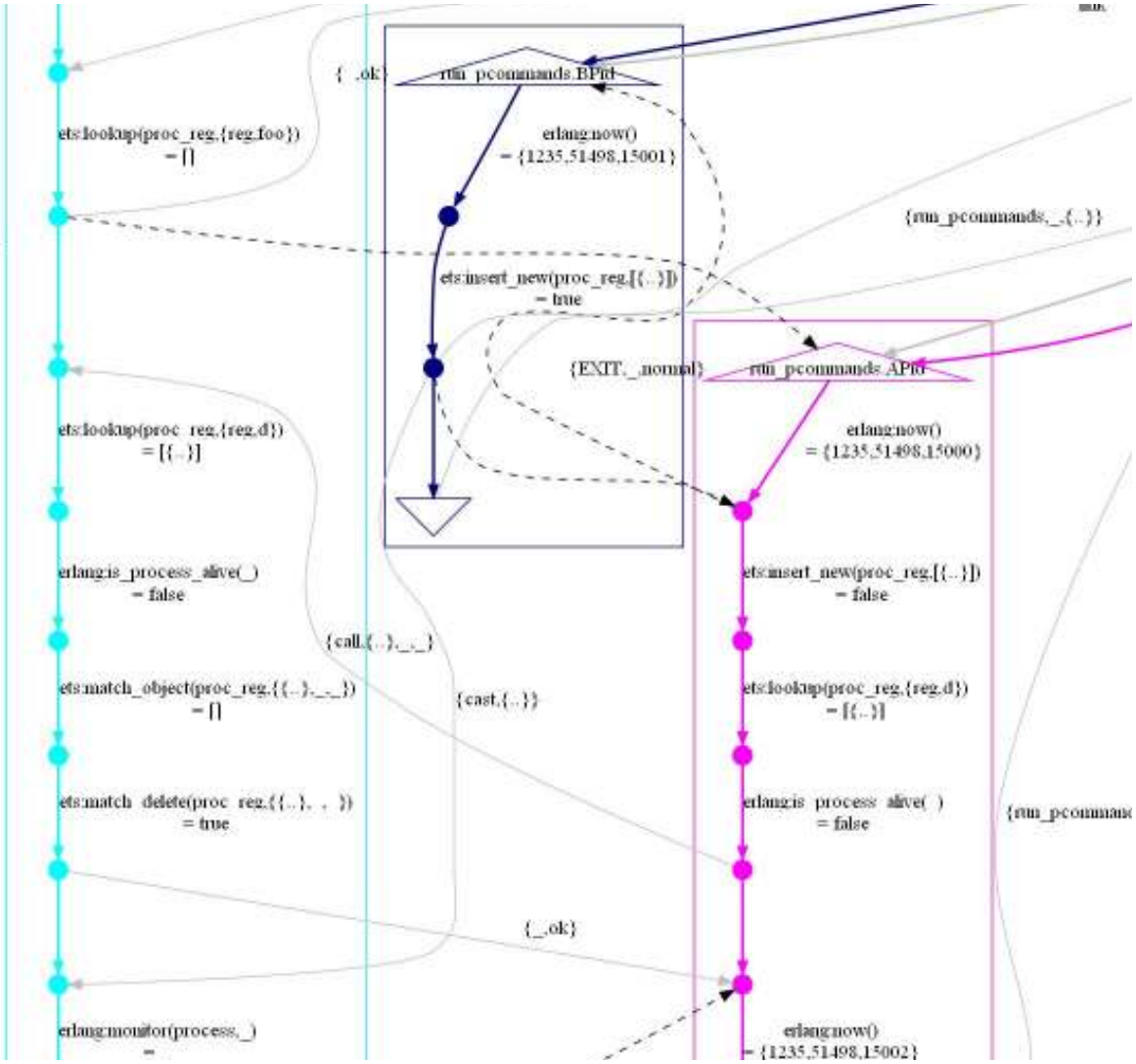
APid finds that the name is taken, but by a dead process – it asks the server for an audit (the call)

But the call reaches the server before the cast (true preemption)

The server uses a reverse lookup (updated when the cast is received) to find the name.

Dotted arrows indicate a non-deterministic ordering (usually a sign of trouble)

State chart detail



BPid's registration succeeds, and it sends a cast to the admin server

APid finds that the name is taken, but by a dead process – it asks the server for an audit (the call)

But the call reaches the server before the cast (true preemption)

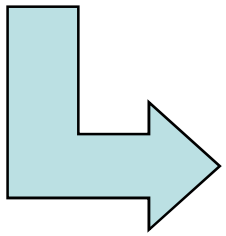
The server uses a reverse lookup (updated when the cast is received) to find the name.

Since the messages arrived in an unexpected order, the name remains after the audit, and APid fails unexpectedly.

Dotted arrows indicate a non-deterministic ordering (usually a sign of trouble)

Bug fix #1

```
do_reg(Id, Pid) ->
    Now = erlang:now(),
    RegEntry = {Id, Pid, Now},
    case ets:insert_new(proc_reg, RegEntry) of
          false ->
              false;
          true ->
              ?gen_server:cast(proc_reg, {new_reg, Id, Pid, Now}),
              true
    end.
```



```
do_reg(Id, Pid) ->
    Now = erlang:now(),
    RegEntry = {{reg, Id}, Pid, Now},
    RevEntry = {{rev, Pid, Id}, undefined, undefined},
    case ets:insert_new(proc_reg, [RegEntry, RevEntry]) of
        false ->
            false;
        true ->
            ?gen_server:cast(proc_reg, {new_reg, Id, Pid, Now}),
            true
    end.
```

- Insert a dummy reverse mapping together with the registration entry
- ets:insert_new(Objects) is atomic
- The server simply overwrites the dummy entry

Erlang

Bug # 2

```
Shrinking.............(12 times)
{[{set,{var,5},{call,proc_reg_eqc,spawn,[]}},
  {set,{var,23},{call,proc_reg_eqc,kill,[{var,5}]}},
  {set,{var,24},{call,proc_reg_eqc,reg,[b,{var,5}]}}],
 {[{set,{var,25},{call,proc_reg_eqc,reg,[b,{var,5}]}}],
  [{set,{var,26},{call,proc_reg_eqc,reg,[b,{var,5}]}}]}}
{-9065357021,-6036499020,-6410890974}
Sequential: [{{state,[],[],[]},<0.26845.2>},
             {{state,[<0.26845.2>],[],[]},ok},
             {{state,[<0.26845.2>],[],[<0.26845.2>]},true},
             {{state,[<0.26845.2>],[],[<0.26845.2>]},ok}]
Parallel: {[{{call,proc_reg_eqc,reg,[b,<0.26845.2>]},
             {'EXIT',{badarg,[{proc_reg,reg,2},
                              {proc_reg_eqc,reg,2},
                              {parallel2,run,2},
                              {parallel2,'-run_pcommands/3-fun-0-',3}]}}}],
           [{{call,proc_reg_eqc,reg,[b,<0.26845.2>]},true}]}
Res: no_possible_interleaving
```

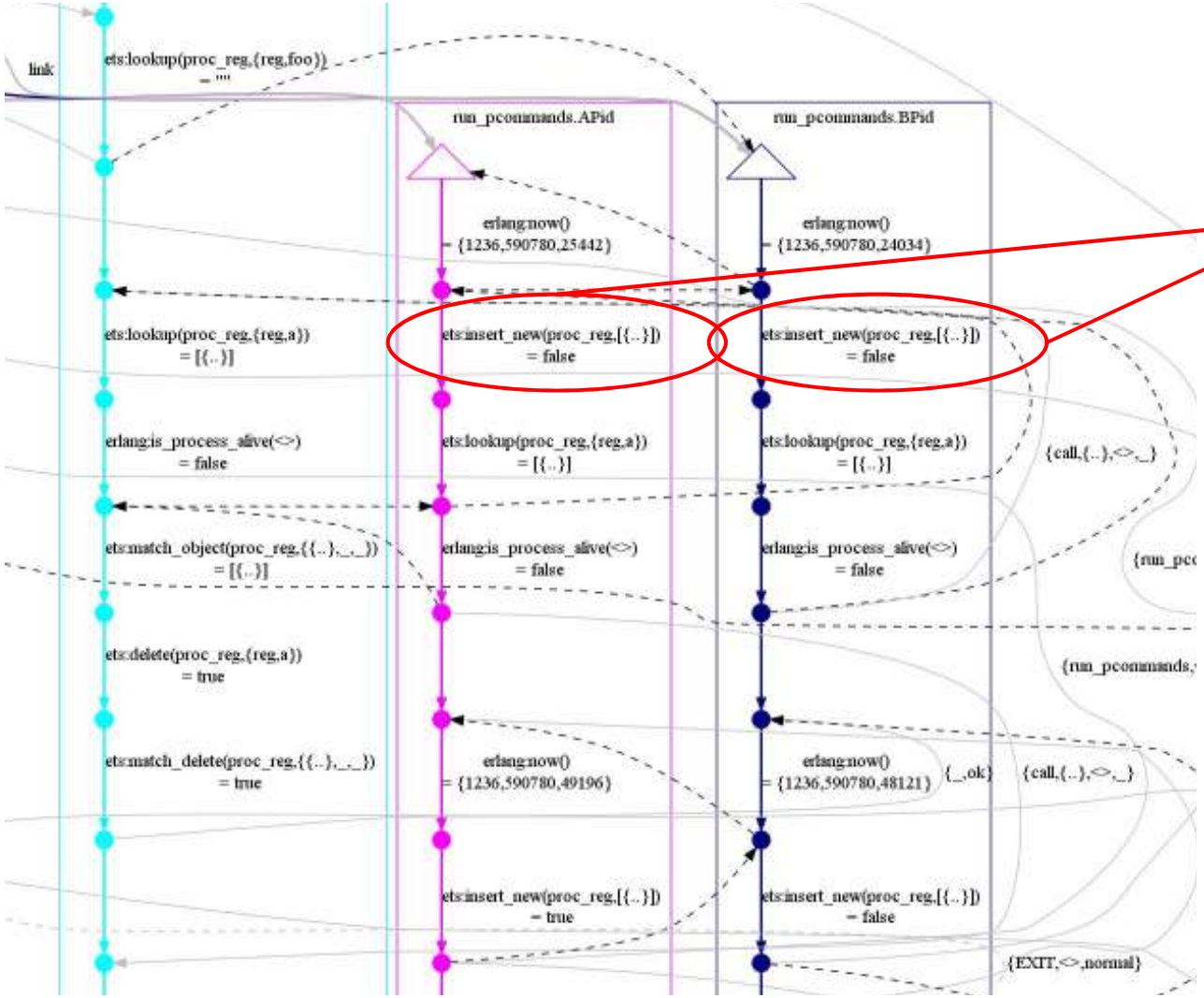
”Sequential prefix”

Parallel component

- Still problematic to register a dead process!
- Let’s look at the graph...

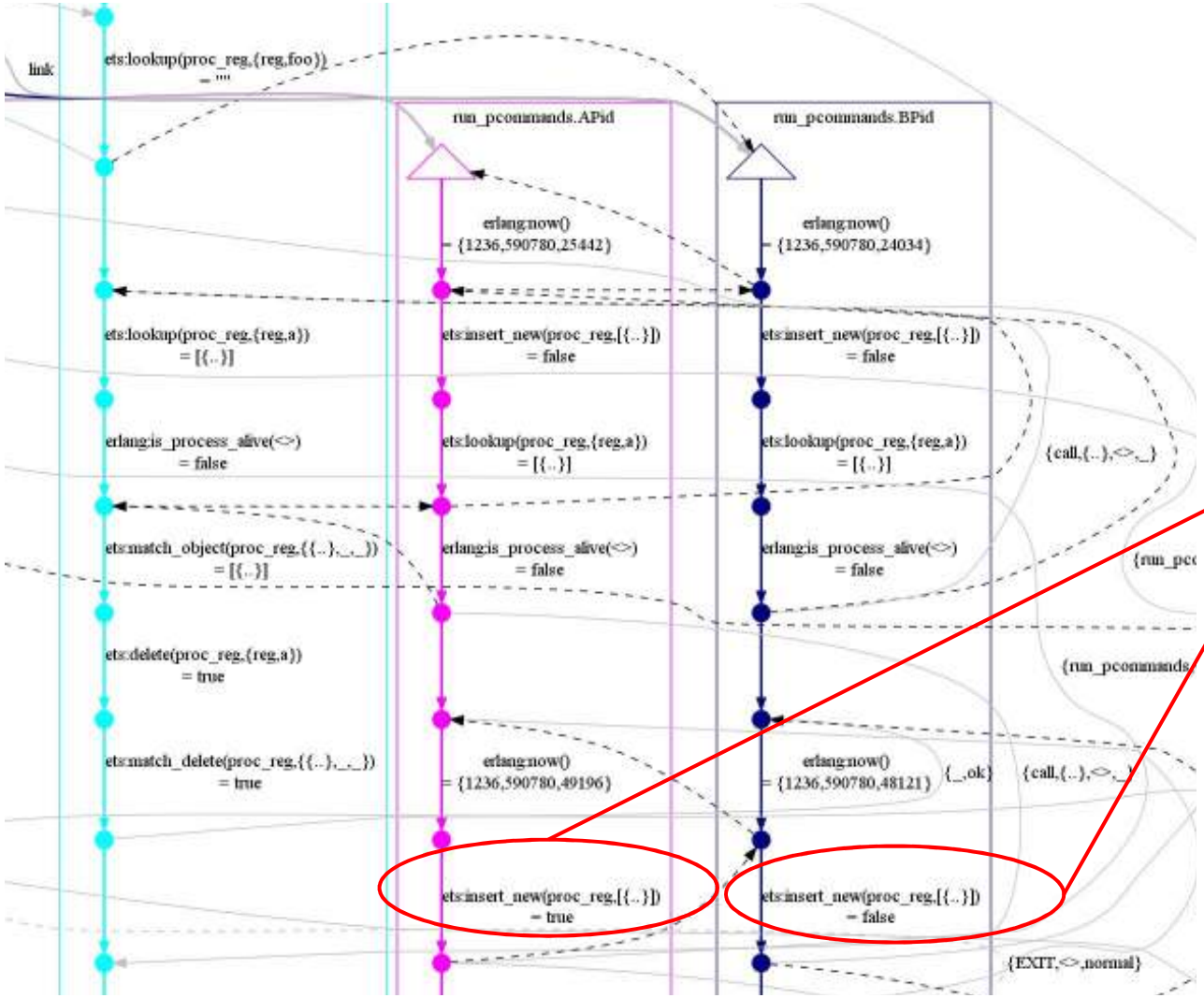
Erlang

Chart detail, Bug # 2



Since the name is already registered, both `APid` and `BPid` request an audit

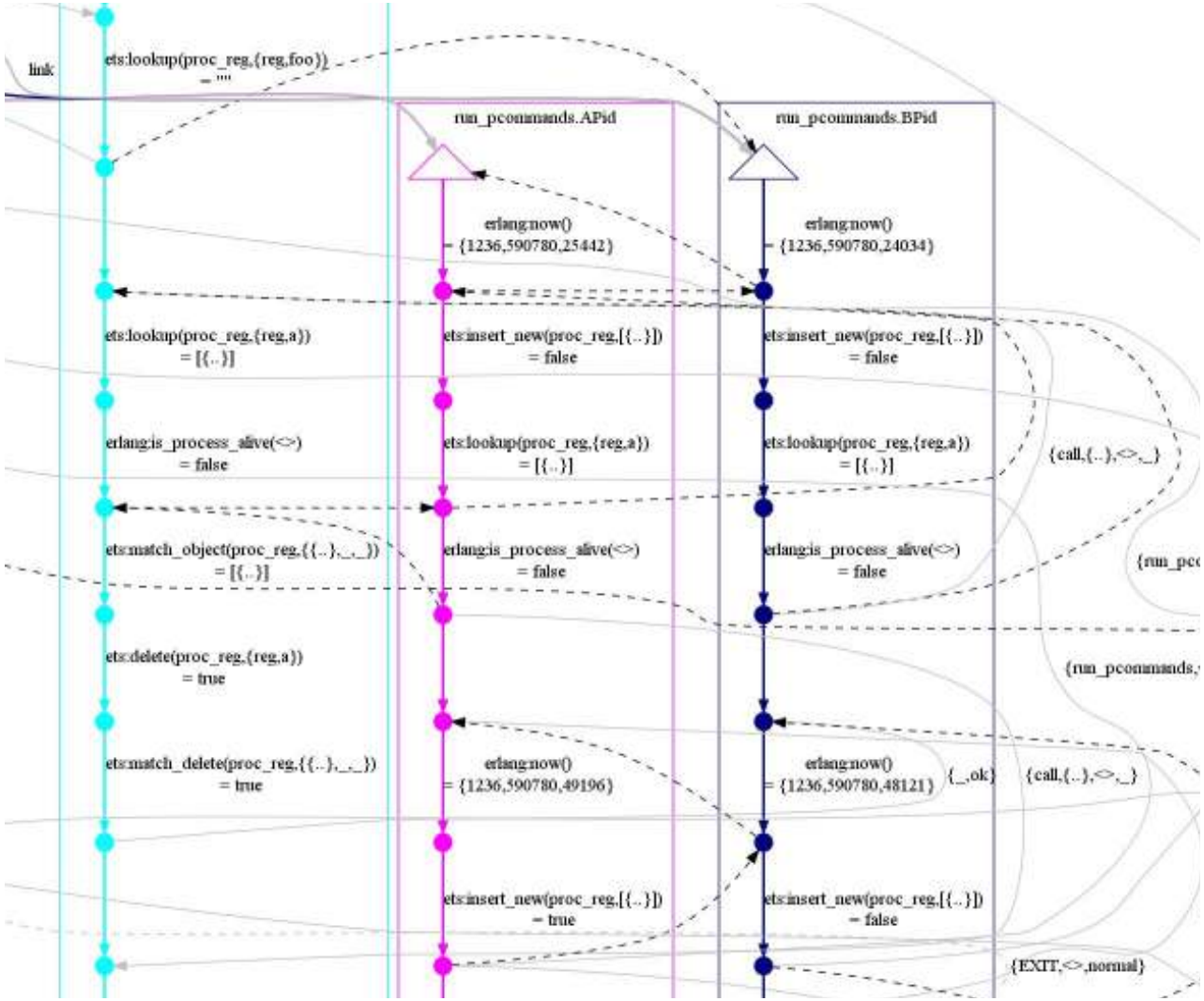
Chart detail, Bug # 2



Since the name is already registered, both `APid` and `BPid` request an audit

Both then assume that it will be ok to register, but one still fails.

Chart detail, Bug # 2



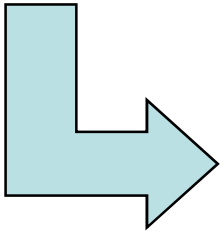
Since the name is already registered, both `APid` and `BPid` request an audit

Both then assume that it will be ok to register, but one still fails.

This is ok (valid race condition), but not if it's a dead process!!!!

Bug fix # 2

```
do_reg(Id, Pid) ->
    Now = erlang:now(),
    RegEntry = {{reg,Id}, Pid, Now},
    RevEntry = {{rev,Pid,Id},undefined,undefined},
    case ets:insert_new(proc_reg, [RegEntry,RevEntry]) of
        false ->
            false;
        true ->
            ?gen_server:cast(proc_reg, {new_reg, Id, Pid, Now}),
            true
    end.
```



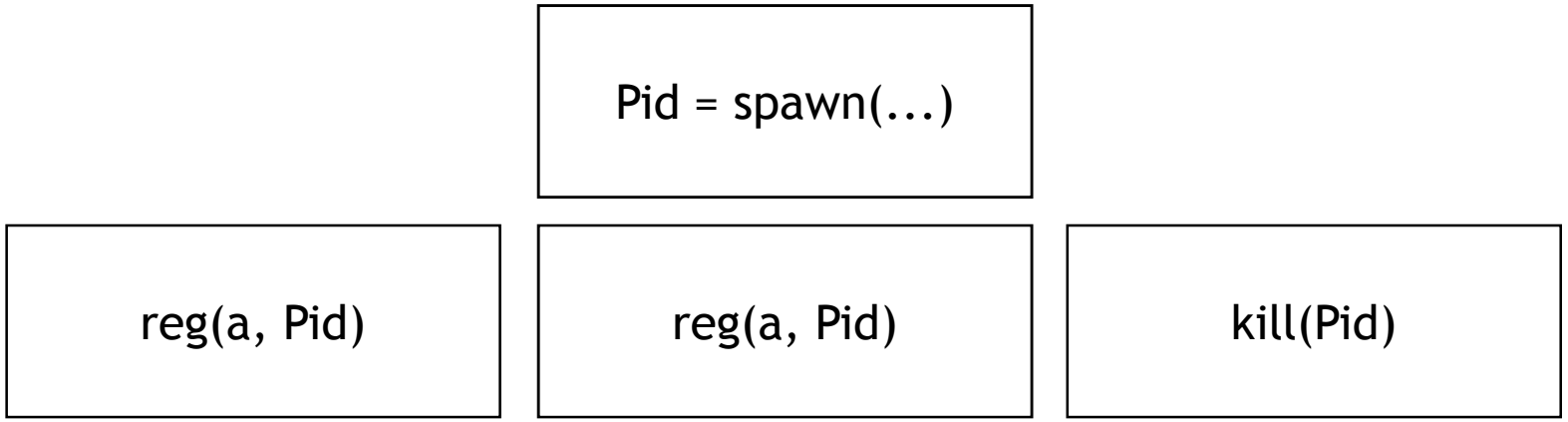
```
do_reg(Id, Pid) ->
 case is_process_alive(Pid) of
          false ->
              where(Id) == undefined;
          true ->
              Now = erlang:now(),
              ...
    end.
```

- Don't ever insert a dead process in the registry (duh...)
- After this fix, the code passed 20 000 tests with the custom scheduler.

Erlang

But wait...

The code still has a race condition!



- What does the specification say about a process dying while it's being registered?
- If it were ok to register the same (Name, Pid) twice, it would be easier (but the register/2 BIF doesn't allow it...)
- For now, QuickCheck only works with two parallel sequences.
- Alternative approach: Require processes to register themselves!

Erlang

Consequence for the Programmer

- SMP Erlang introduces non-deterministic scheduling
- Increased risk of subtle race conditions
  - ...yes, even in Erlang
- Hopefully, we can find them in unit test with tools like QuickCheck
- API semantics may have to be revisited

Erlang

Shameless plugs

http://www.erlang-factory.com

Erlang FACTORY
BAY AREA 2009
University | April 27 – 29
Conference | April 30 – May 1

Great lineup of speakers!

Erlang and QuickCheck courses

ProTest
property based testing

http://www.protest-project.eu/

Advanced testing research – not just for Erlang

Erlang