Finding Race Conditions during Unit Testing with QuickCheck

John Hughes (Quviq/Chalmers)

Koen Claessen, Michal Palka, Nick Smallbone (Chalmers)

Thomas Arts, Hans Svensson (Quviq/Chalmers)

Ulf Wiger, (Erlang Training and Consulting)

Race Conditions

- Everybody's nightmare!
  - Timing dependent, often don't show up until system testing
  - Hard to reproduce
  - More likely to strike on multicore processors
  - Erlang is not immune
- **Goal:** find race conditions in *unit testing*, using QuickCheck and PULSE
- **Story:** Ulf Wiger's extended process registry

From Unit Testing to QuickCheck

- **Example**: lists:delete/2 removes an element from a list

```
delete_present_test() ->
  ?assertEqual([1,3],lists:delete(2,[1,2,3])).

delete_absent_test() ->
  ?assertEqual([1,2,3],lists:delete(4,[1,2,3])).
```

- Did I think of enough cases?
- How much time/energy/code am I prepared to spend on this?

Property Based Testing

- Generate test cases instead
  - As many as you like!
  - **Challenge**: from what universe?
  - **Challenge**: understandable failures

int() and list(int())

A deleted
element is
really gone?

- Decide test outcome with a *prope*
  - **Challenge**: no ”expected value” anymore?
  - Need to formulate a general property

A property of lists:delete

Test case

Test case generator

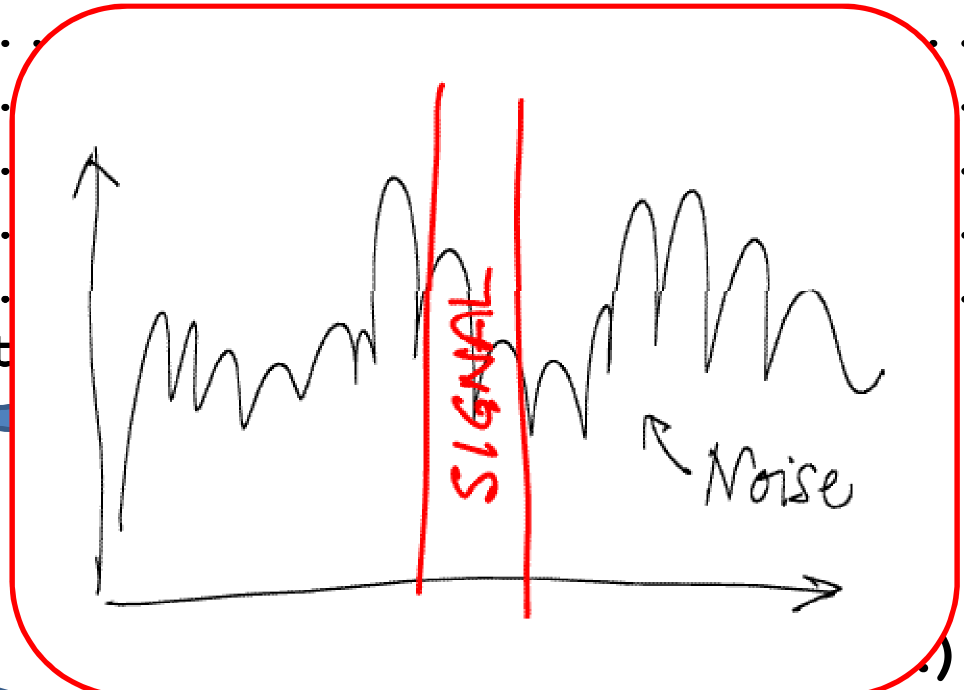
```
prop_delete() ->
   ?FORALL({I,L},
           {int(),list(int())},
           not lists:member(I,
                   lists:delete(I,L))).
```

Test

```
21> eqc:quickcheck(examples:prop_delete()).
.....................................................
..........................................
OK, passed 100 tests
```

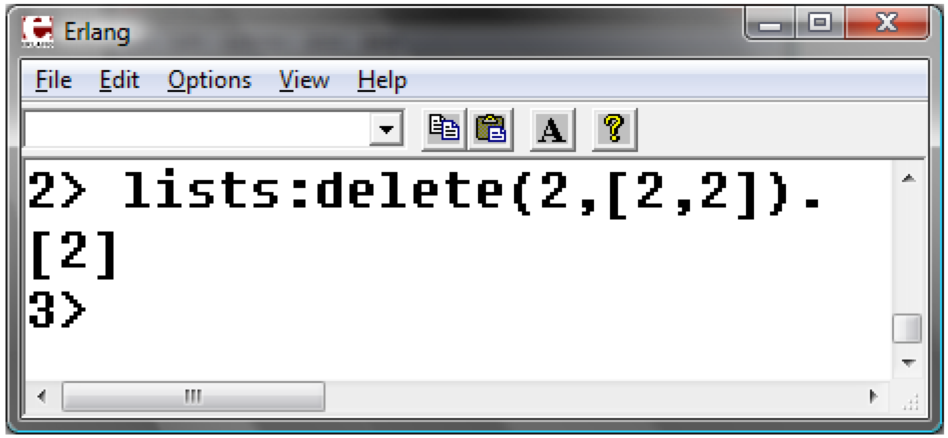
Or maybe not…

```
29> eqc:quickcheck(eqc:numtests(1000,examples:prop_delete())).
..........................................................
..........................                           ....
..........................                           ...
..........................                           ...
..........................                           ...
..........................                           ...
...Failed! After 346 test
{2,[-7,-13,-15,2,2]}
Shrinking.(1 times)
{2,[2,2]}
false
```



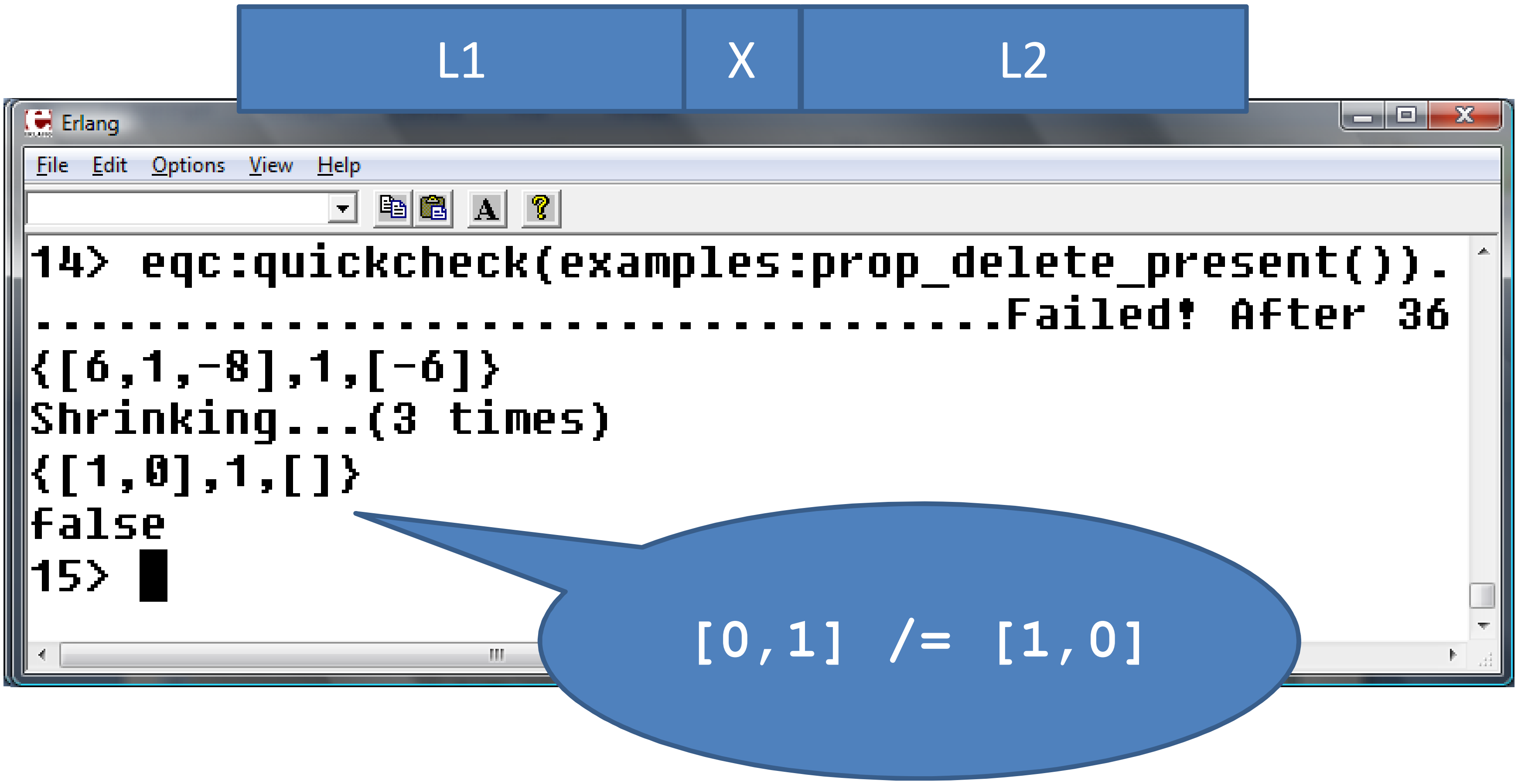
A simplest failing test

What's going on?



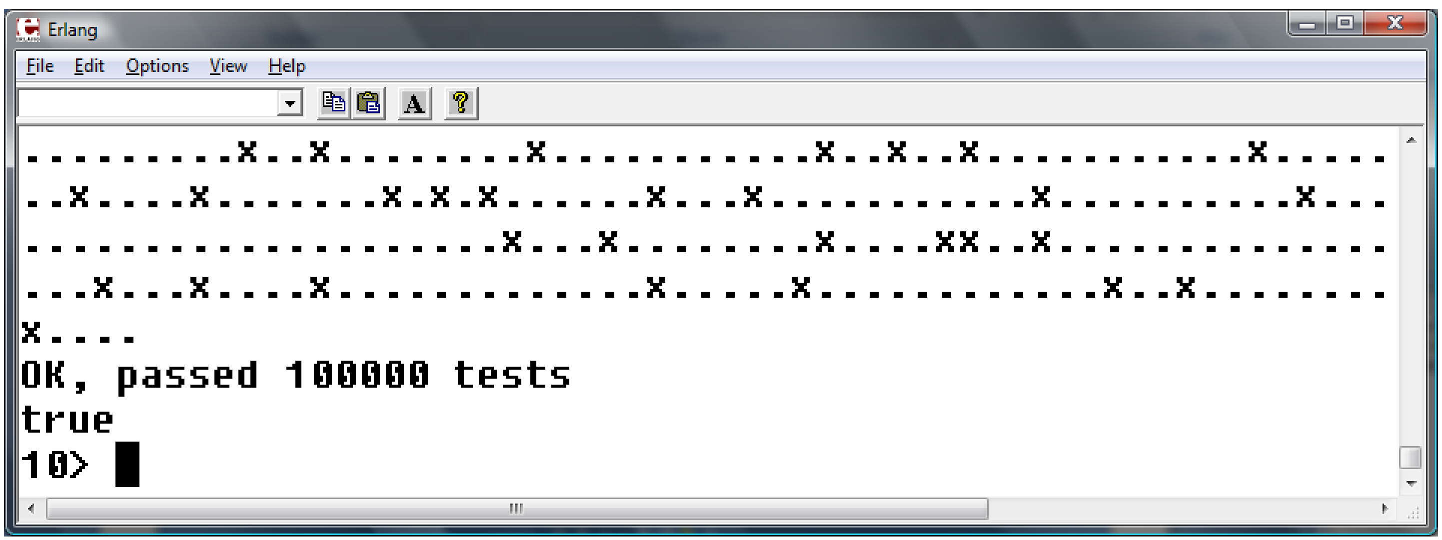
- This is supposed to happen!
  - lists:delete removes *one* occurrence
  - We need a test case where the element occurs twice

A Richer Property



Fixing the Property

```
prop_delete_first() ->
   ?FORALL({L1,X,L2},
           {list(int()),int(),list(int())},
     ?IMPLIES(not lists:member(X,L1),
              lists:delete(X,L1++[X]++L2)
               == L1++L2)).
```



Process Registry is Stateful

- What functions do we want to test?
  - register(Name,Pid), unregister(Name)
  - spawn(), kill(Pid)
- Test cases?
  - Sequences of *calls* to API under test

```
[{set,{var,1},{call,reg_eqc,spawn,[]}},
 {set,{var,2},{call,erlang,register,[a,{var,1}]}}]
```

Just Erlang terms…
symbolic

```
V1 = spawn(),
V2 = register(a,V1).
```

Abstract Mod

- Model

g

```
-reco
  {
   re                                                d pids
```

- De

```
next_                    ,                ng,register,[Name,Pid]}) ->
    S#state              gs=[ Name,Pid} | S#state.regs]};
......
```

Command
generators

P

tions

What's the property?

- For all sequences of API call…
- …where all the pre…
- …no uncaught exce…
- …and all the postcondition… true.

The meat is in the pre- and postconditions and the state model

```
prop_registration() ->
  ?FORALL(Cmds,commands(?MODULE),
    begin
      {H,S,Res} = run_commands(?MODULE,Cmds),
      [catch unregister(N) || N<-?names],
      Res==ok
    end).
```

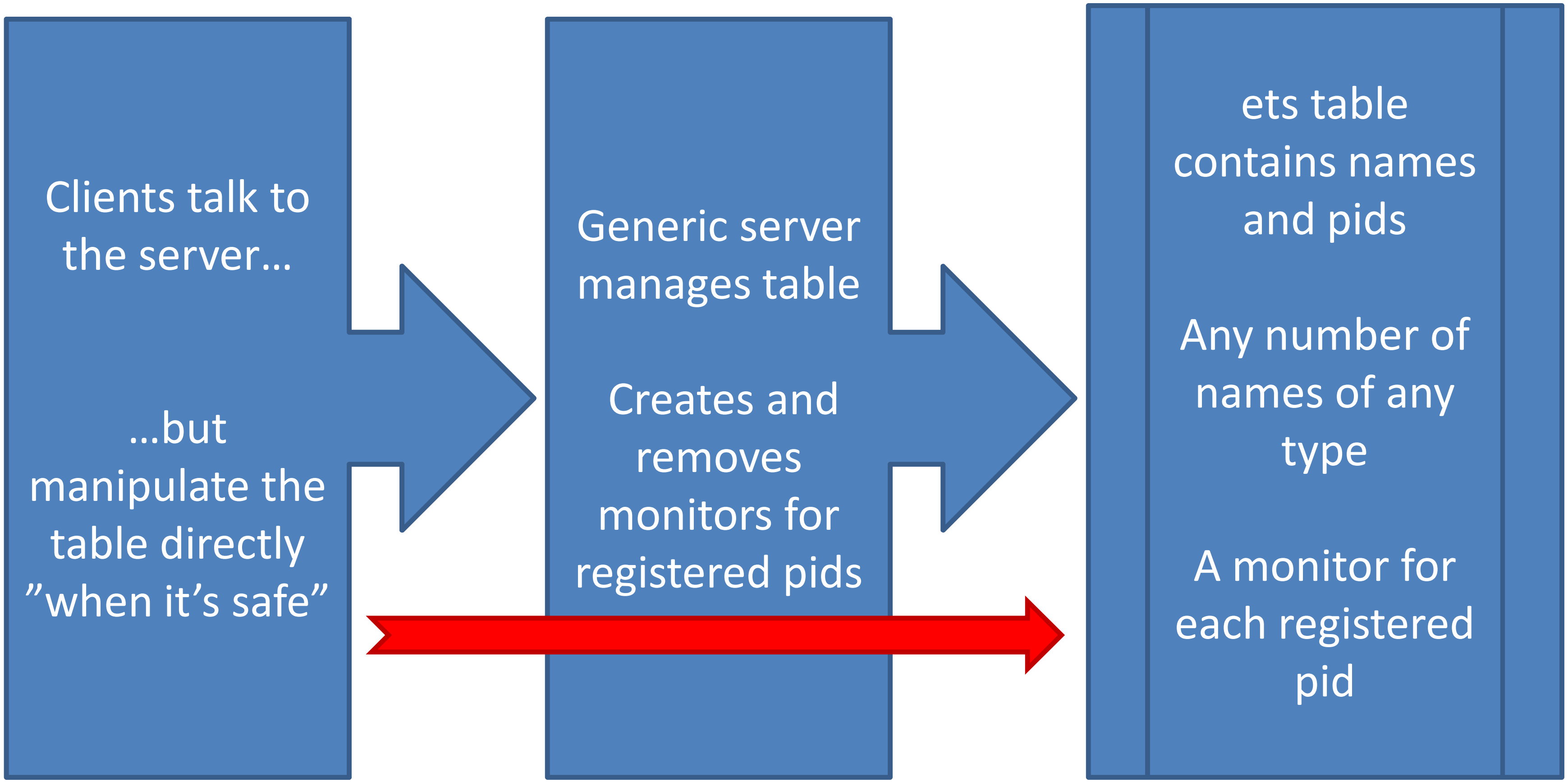
```
postcondition(S,{call,?MODULE,register,[Name,Pid]},V) ->
    case register_ok(S,Name,Pid) of
        true -> V==true;
        false ->  is_exit(V)
    end.

register_ok(S,Name,Pid) ->
    not lists:keymember(Name,1,S#state.regs) andalso
    not lists:keymember(Pid, 2,S#state.regs).
```

A Pid can only be registered with *one* name!

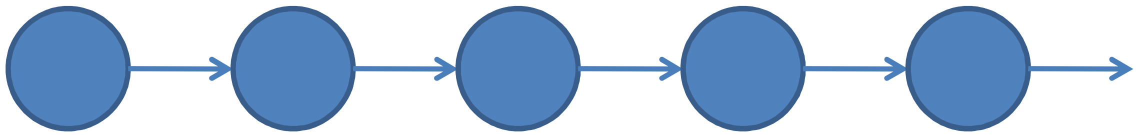
```
V2 = spawn(),
V3 = register(a,V2),
V5 = register(b,V2).
```

Extended Process Registry

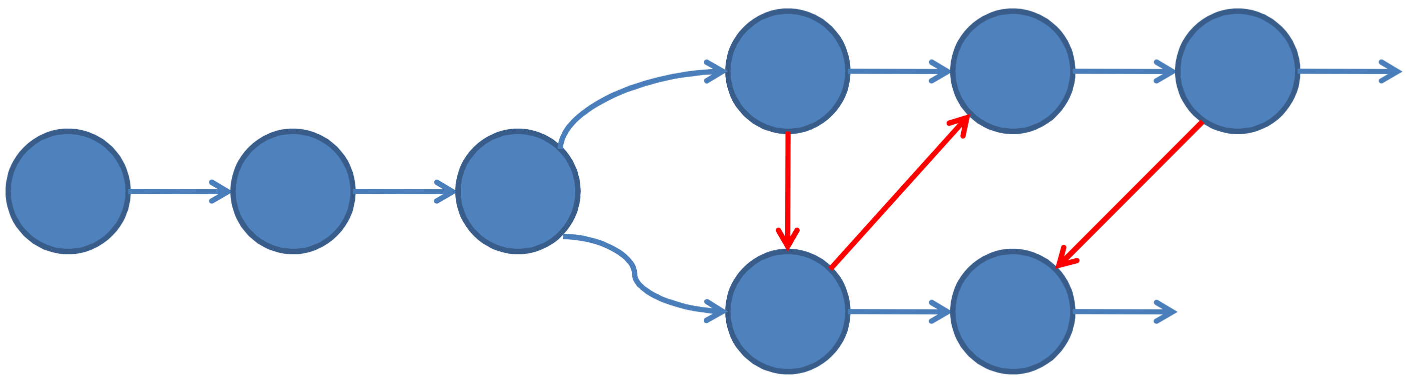


What is a Parallel Test Case?

- Sequential test case:

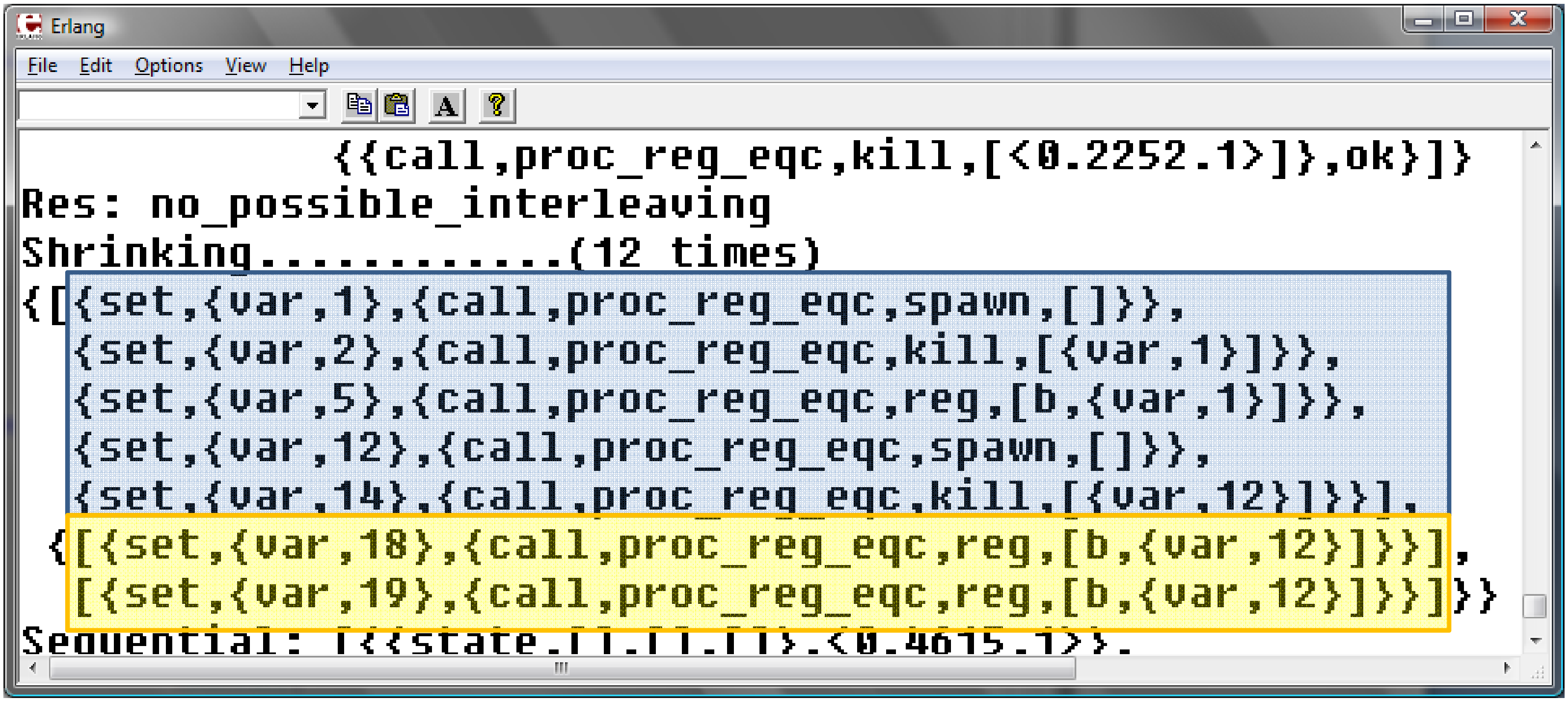


- Parallel test case:



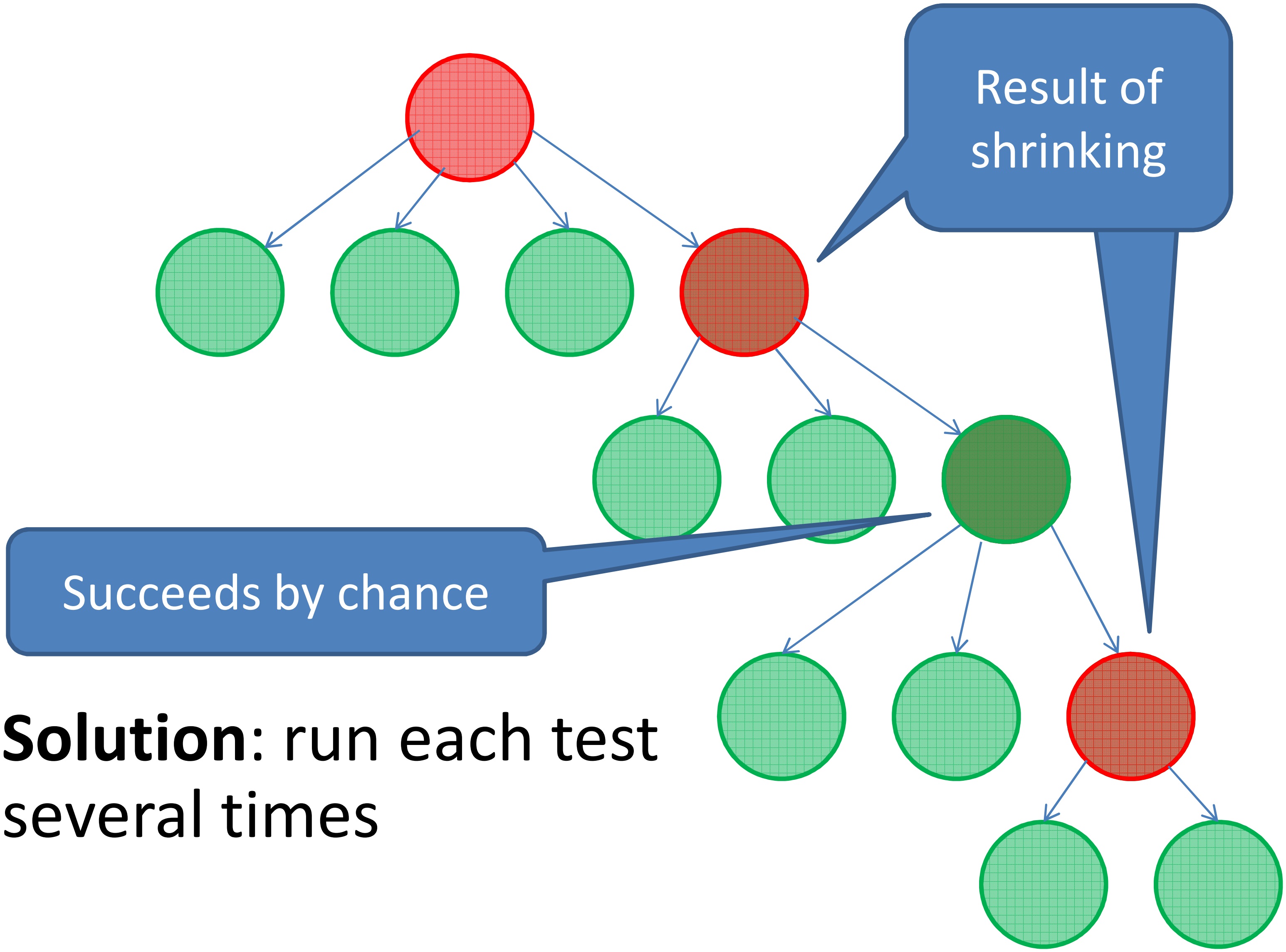
- We *reuse* the specification of the sequential case

Testing the EPR



- Must it be so complicated?

How Shrinking Works



- **Solution**: run each test several times

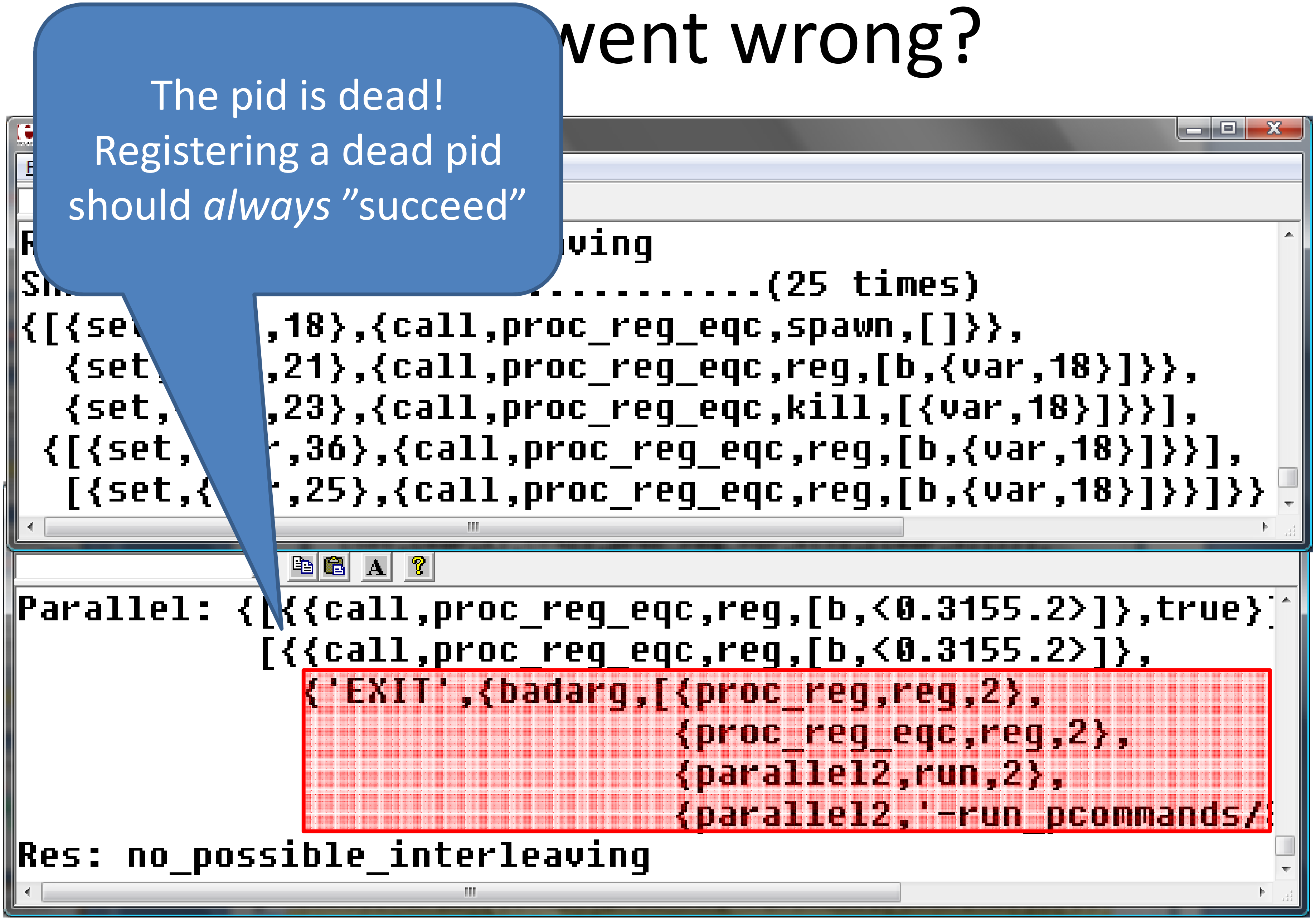
Shrinking the EPR failure

- With test repetition…

```
Res: no_possible_interleaving
Shrinking..........................(25 times)
{[{set,{var,18},{call,proc_reg_eqc,spawn,[]}},
  {set,{var,21},{call,proc_reg_eqc,reg,[b,{var,18}]}},
  {set,{var,23},{call,proc_reg_eqc,kill,[{var,18}]}}],
 {[{set,{var,36},{call,proc_reg_eqc,reg,[b,{var,18}]}}],
  [{set,{var,25},{call,proc_reg_eqc,reg,[b,{var,18}]}}]}}
```

- Every step is necessary
- The last two *must* be in parallel

went wrong?



The pid is dead!
Registering a dead pid
should *always* ”succeed”
ving
...........(25 times)
{[{se ,18},{call,proc_reg_eqc,spawn,[]}},
{set ,21},{call,proc_reg_eqc,reg,[b,{var,18}]}},
{set, ,23},{call,proc_reg_eqc,kill,[{var,18}]}}],
{[{set, ,36},{call,proc_reg_eqc,reg,[b,{var,18}]}}],
[{set,{ ,25},{call,proc_reg_eqc,reg,[b,{var,18}]}}]}}
Parallel: {[{{call,proc_reg_eqc,reg,[b,<0.3155.2>]},true}
[{{call,proc_reg_eqc,reg,[b,<0.3155.2>]},
{'EXIT',{badarg,[{proc_reg,reg,2},
{proc_reg_eqc,reg,2},
{parallel2,run,2},
{parallel2,'-run_pcommands/
Res: no_possible_interleaving

But what happened?

ProTest

User

Level

Scheduler

for **E**rlang

- Instruments Erlang code
  - To make it talk to…
- A *user-level scheduler*
  - Which tells each process when to ru…
- Rando… …le
  - To… …ions
- Recor…ng a detailed trace

Works with any OTP release!

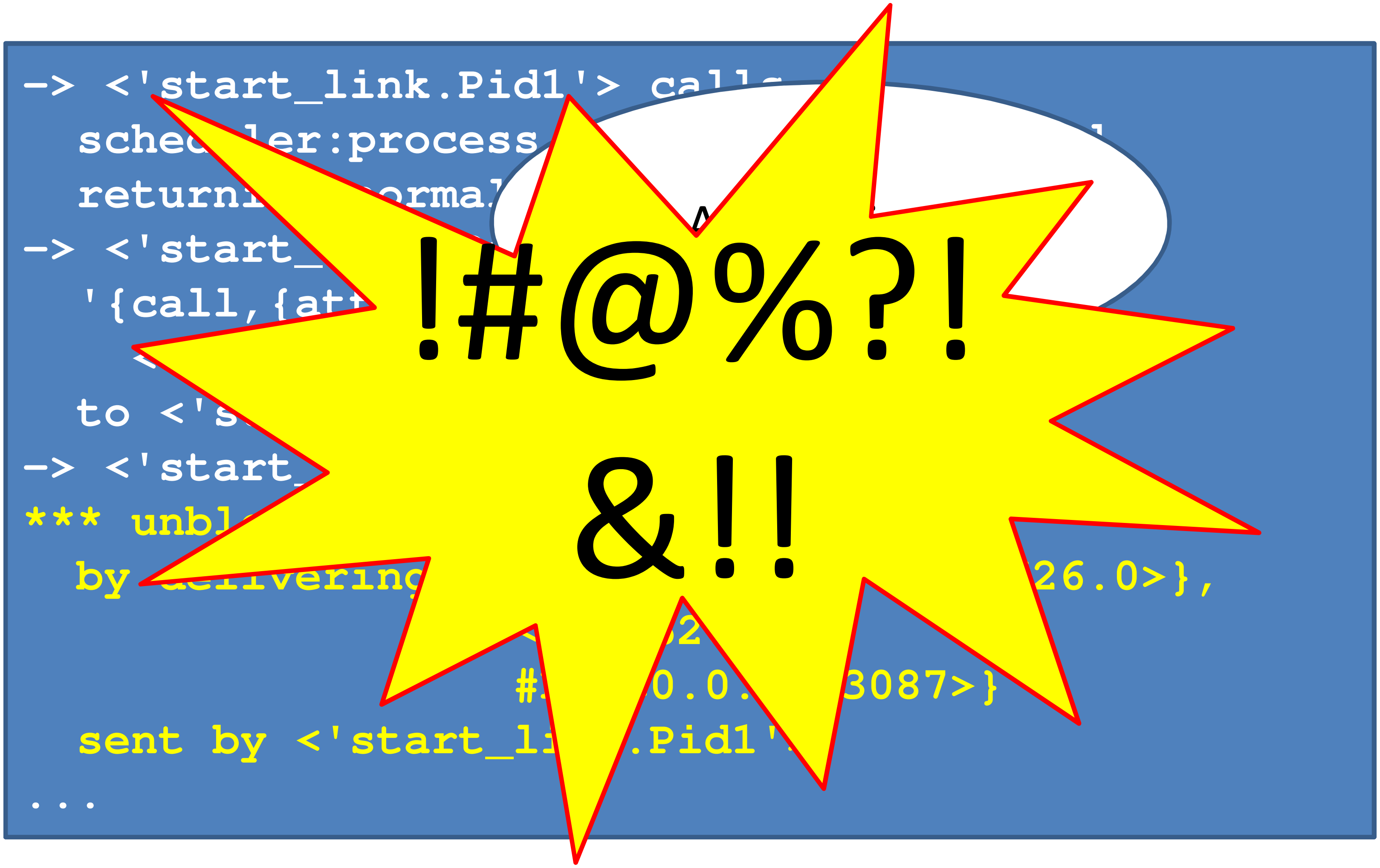
Pulsing the EPR

- PULSE provokes an even simpler counterexample:

```
{[{set,{var,9},{call,proc_reg_eqc,spawn,[]}},
  {set,{var,10},{call,proc_reg_eqc,kill,[{var,9}]}}],
 {[{set,{var,15},{call,proc_reg_eqc,reg,[c,{var,9}]}}],
  [{set,{var,12},{call,proc_reg_eqc,reg,[c,{var,9}]}}]}}
```

- As before, one of the calls to reg raises an exception.
- All we need is a dead process!

Inspecting the Trace



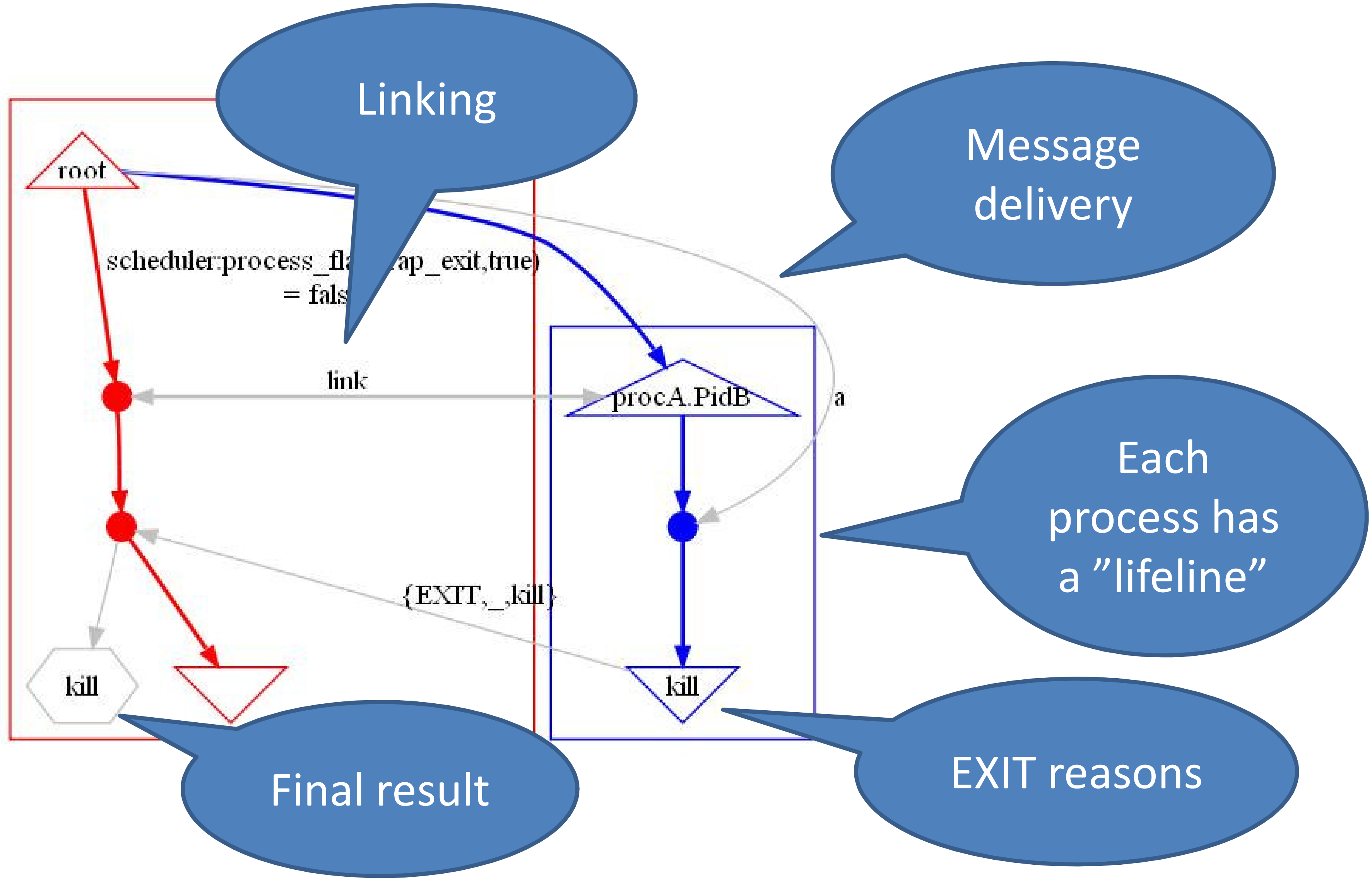
Trace Visualization

- A simple example:

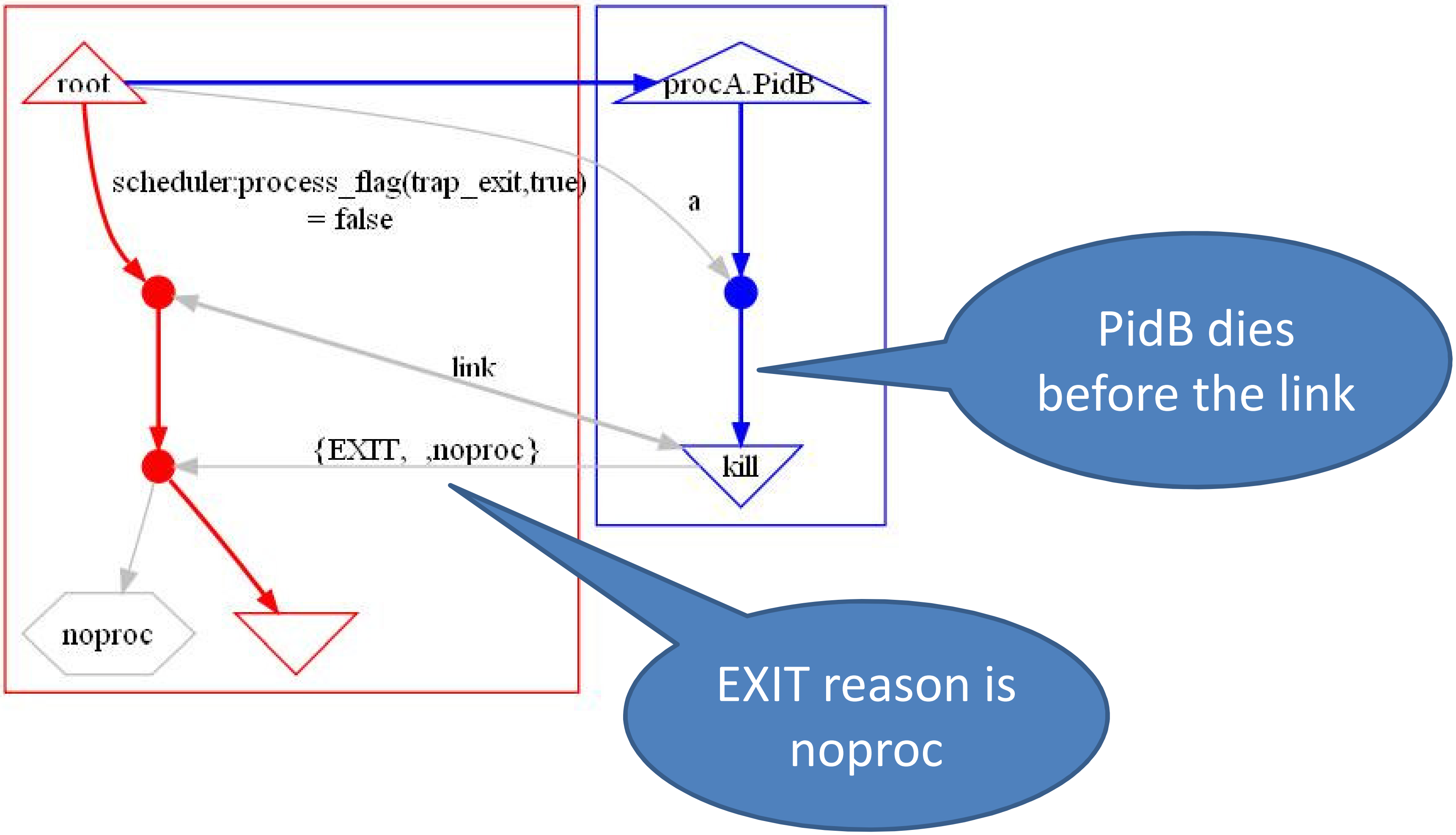
```
procA() ->
  PidB = spawn(fun procB/0),
  PidB ! a,
  process_flag(trap_exit, true),
  link(PidB),
  receive
    {'EXIT',_,Why} -> Why
  end.
```

```
procB() ->
  receive
    a ->
      exit(kill)
  end.
```

One possibility

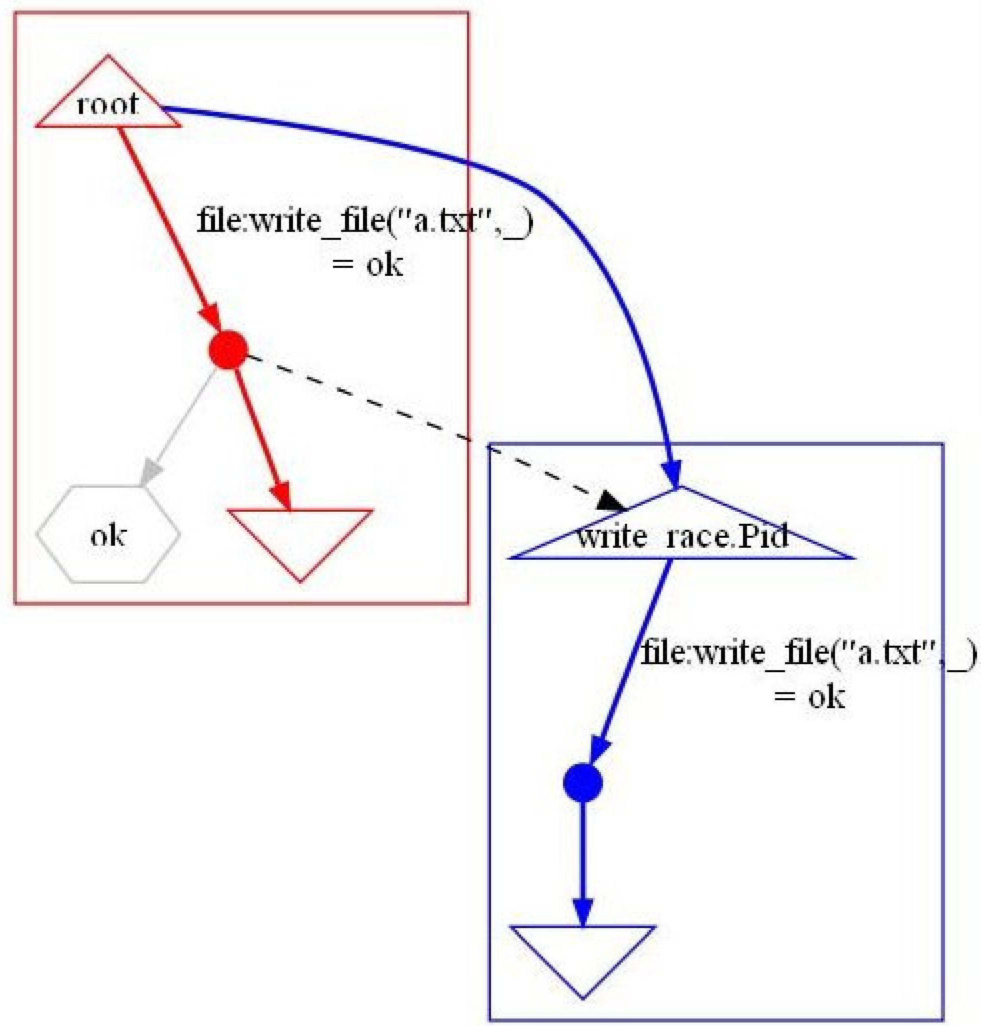


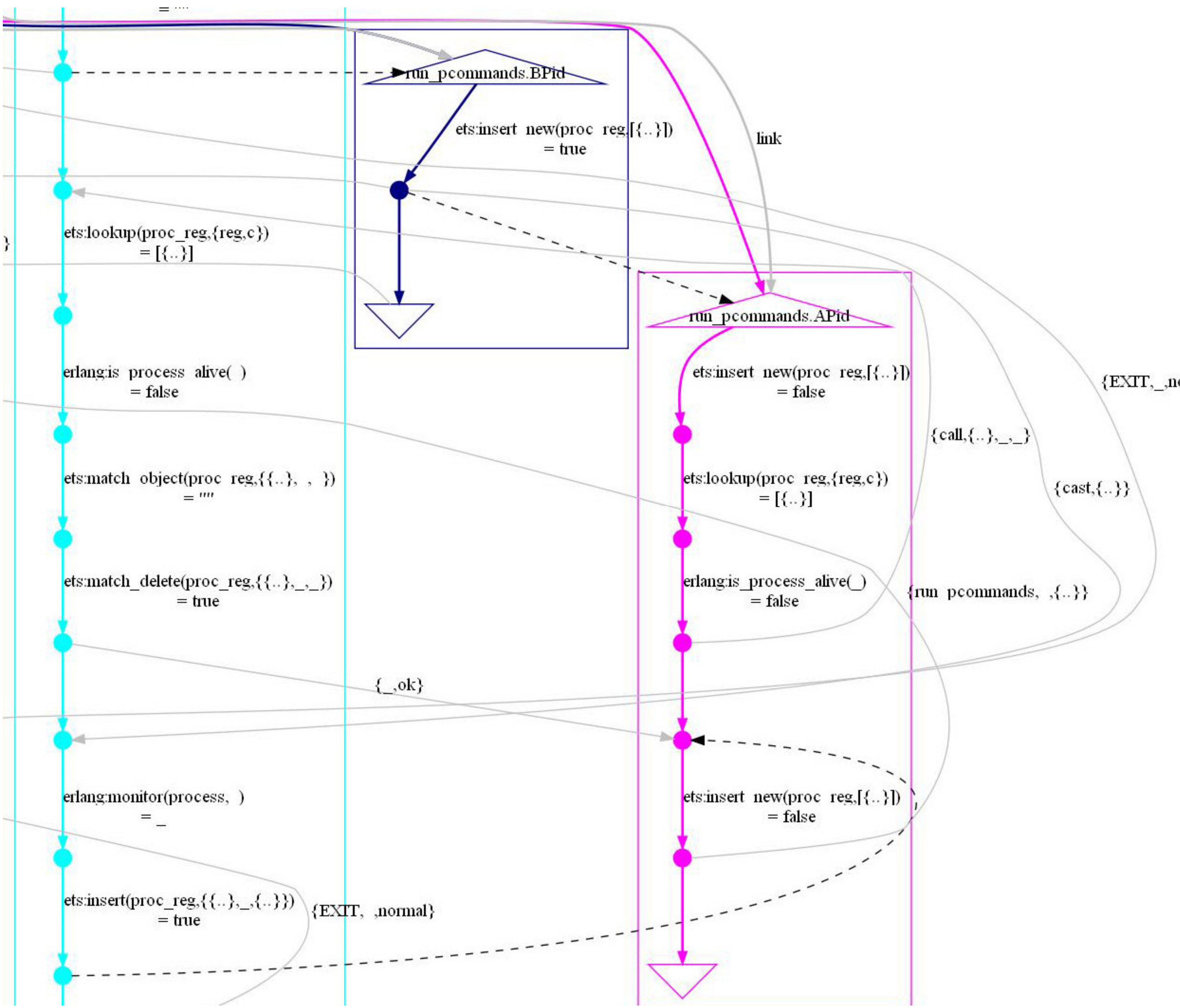
Another possibility



Side-effect order

- Two processes racing to write a file
- Order is not implied by message passing—so it needs to be shown explicitly





run_pcommands.BPid
ets:insert_new(proc_reg,[{..}]) = true
link
ets:lookup(proc_reg,{reg,c}) = [{..}]
run_pcommands.APid
erlang:is_process_alive(_) = false
ets:insert_new(proc_reg,[{..}]) = false
ets:match_object(proc_reg,{{..},_,_}) = ""
ets:lookup(proc_reg,{reg,c}) = [{..}]
{call,{..},_,_}
{cast,{..}}
ets:match_delete(proc_reg,{{..},_,_}) = true
erlang:is_process_alive(_) = false
{run_pcommands,_,{..}}
{_,ok}
erlang:monitor(process,_) = _
ets:insert_new(proc_reg,[{..}]) = false
ets:insert(proc_reg,{{..},_,{..}}) = true
{EXIT,_,normal}

How does it work?

Client

ets:insert_new to add {Name,Pid} to the registry

If successful, tells server to complete addition

Server

Creates a monitor and adds another entry {{Name,Pid},Monitor}

How does it work?

Client

ets:insert_new to add {Name,Pid} to the registry

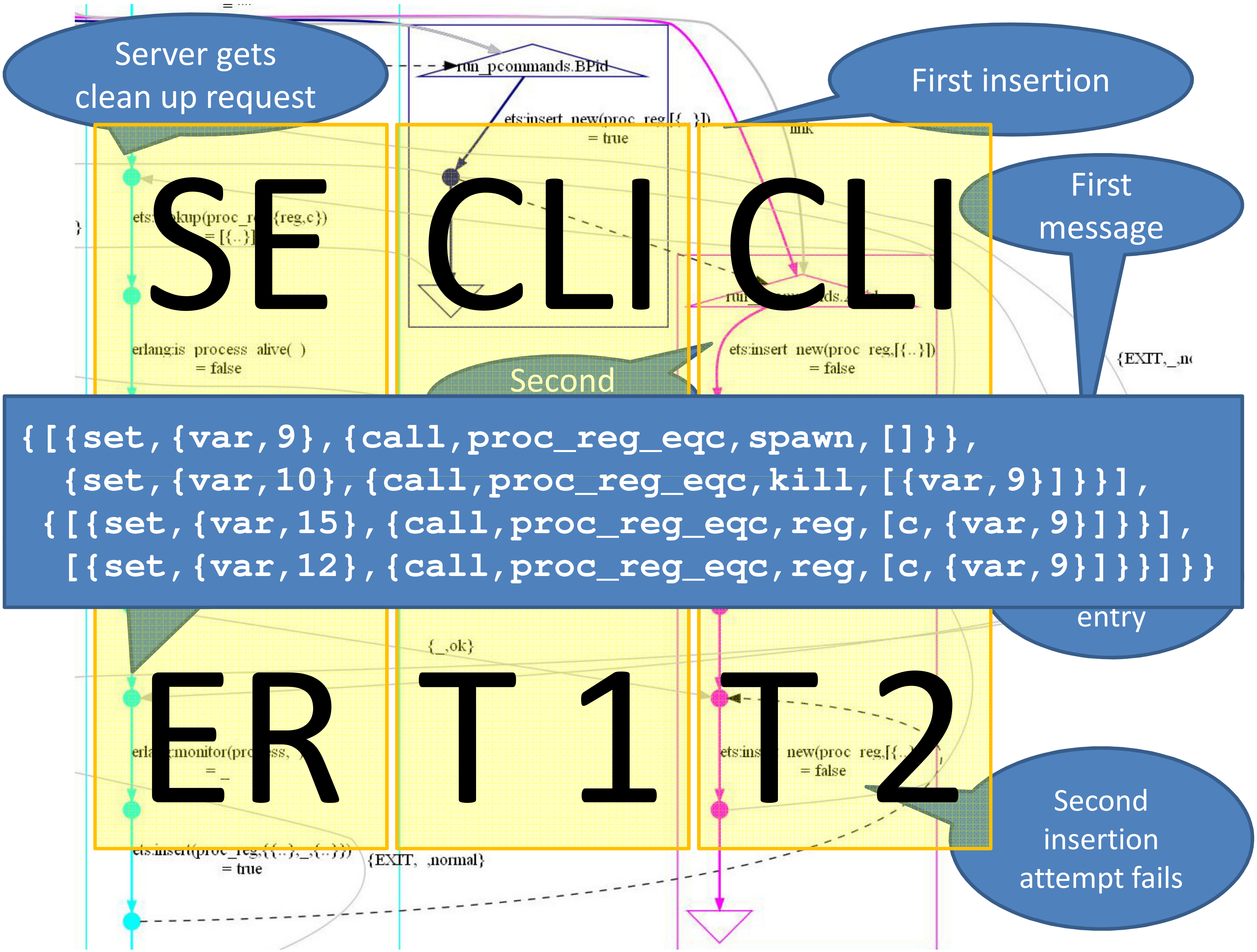
If it fails, but whereis(Pid) is dead, ask server to clean it up

Repeats the insert_new and request to server, assumes it succeeds

Server

Finds and deletes {{Name,Pid},Monitor} and the {Name,Pid} entry, replies ok

Creates the monitor and completes the job


Server gets clean up request
First insertion
First message
Second
entry
Second insertion attempt fails
SERVER
CLIENT 1
CLIENT 2
run_pcommands.BPid
= true
erlang:is_process_alive(_)
= false
ets:insert_new(proc_reg,[{..}])
= false
{_,ok}
{EXIT,_,normal}
{[{set,{var,9},{call,proc_reg_eqc,spawn,[]}},
{set,{var,10},{call,proc_reg_eqc,kill,[{var,9}]}}],
{[{set,{var,15},{call,proc_reg_eqc,reg,[c,{var,9}]}}],
[{set,{var,12},{call,proc_reg_eqc,reg,[c,{var,9}]}}]}}

A Fix

Client

ets:insert_new to add {Name,Pid} to the registry, *and a dummy {{Name,Pid},Monitor} entry*

If successful, tells server to complete addition

Server

Creates a monitor and adds the real entry {{Name,Pid},Monitor}

Conclusions

- Property-based testing works just fine to hunt for race conditions
- PULSE makes tests controllable, repeatable, and observable
- Visualization makes it possible to interpret test traces