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

```erl
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

• Decide test outcome with a property
  – **Challenge**: no "expected value" any more.
  – Need to formulate a general property

`int()` and `list(int())`

A deleted element is really gone?
A property of lists:delete

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

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 tests.
{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

prop_delete_present() ->

?FORALL({L1,X,L2},

{list(int()), int(), list(int())},

A RicherProperty L1 L2 X {list(int()), int(), list(int())},

lists:delete(X,L1++[X]++L2) == L1++L2).

[0,1] /= [1,0]
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 Model

- Model state, precisely enough to generate tests and decide outcomes.

Postconditions

Command generators

State transitions

Preconditions

next_state(S,V,{call,erlang,register,[Name,Pid]}) ->
S#state.regs=[{Name,Pid} | S#state.regs];

......
What’s the property?

- For all sequences of API calls...
- ...where all the preconditions are true...
- ...no uncaught exceptions are raised...
- ...and all the postconditions are true.

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

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

```erlang
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).
```

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

A Pid can only be registered with one name!
Extended Process Registry

Clients talk to the server...

...but manipulate the table directly "when it’s safe"

Generic server manages table

Creates and removes monitors for registered pids

ets table contains names and pids

Any number of names of any type

A monitor for each registered pid
What is a Parallel Test Case?

- Sequential test case:
  ![Sequential Test Case Diagram]

- Parallel test case:
  ![Parallel Test Case Diagram]

- 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

Result of shrinking

Succeeds by chance
Shrinking the EPR failure

- With test repetition...

```erlang
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
What went wrong?
The pid is dead!
Registering a dead pid should always "succeed"
But what happened?

- Instruments Erlang code
  - To make it talk to...
- A user-level scheduler
  - Which tells each process when to run...
- Randomising the schedule
  - To provoke race conditions more often...
- Recording a detailed trace

Works with any OTP release!
Pulsing the EPR

• PULSE provokes an even simpler counterexample:

```plaintext
{{{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

-> <'start_link.Pid1'> calls
  scheduler:process_flag [priority,high] returning normal.

-> <'start_link.Pid1'> sends
  '{call,{attach,<0.31626.0>},<0.31626.0>,#Ref<0.0.0.13087>}'
  to <'start_link.Pid'>.
-> <'start_link.Pid1'> blocks. *** unblocking <'start_link.Pid'>
  by delivering '{call,{attach,<0.31626.0>},<0.31626.0>, #Ref<0.0.0.13087>}'
  sent by <'start_link.Pid1'>.

...
• A simple example:

```erlang
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

Linking

Each process has a "lifeline"

Message delivery

Final result

EXIT reasons

Final result

EXIT reasons
Another possibility

- PidB dies before the link
- EXIT reason is noproc
Side-effect order

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

First insertion

First message

Second insertion

Server creates entry to be cleaned up

Second insertion attempt fails
A Fix

Client

```erlang
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

```erlang`
{{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