Real World QuickCheck
DTU

Jesper Louis Andersen
jesper.louis.andersen@gmail.com
Shopgun Aps

January 10, 2017

Who is Jesper?

- Functional programming, type theory, semantics
- 10 years of Erlang experience
- 5+ years with QuickCheck
- Works for a small startup, ShopGun Aps.,
- BA in CS from Copenhagen University

Erlang — in one slide

Erlang is an (untyped) functional language, with three main focal points:

- Communication—Concurrency, Protocol handling
- Robustness—Self-healing, Hardware failure tolerance, soft-realtime
- Continuous operation—Hot code loading, late binding, dynamic reconfiguration

Built originally for telecommunications, in 1986.
Most of the Erlang in this talk is by osmosis.

QuickCheck — summary

We would like to prove a property, such as

$$\forall a \in \text{array}(byte) : \\ unzip(zip(a)) = a \tag{1}$$

- Proof—requires lemmas and theorems about zip/unzip, exploiting their structure
- Check all arrays up to size n is extremely time-consuming, even on large clusters
- Quickcheck—Generate samples of a via highly skewed distributions

QuickCheck – summary

Erlang QuickCheck notation for the example would be:

```
prop_zip_inverse() ->
    ?FORALL(Arr, list(choose(0, 255)),
        begin
            Bin = list_to_binary(Arr),
            Z = zlib:uncompress(zlib:compress(Bin)),
            measure(array_length, length(Arr),
                equals(Bin, Z))
        end).
```

Example output:

```
prop_zip_inverse: … OK, passed 100 tests
array_length:
    Count: 100   Min: 0   Max: 10   Avg: 2.850
    StdDev: 2.350   Total: 285
```

QuickCheck - summary

"Program testing can be used to show the presence of bugs, but never to show their absence!" — Dikjstra EWD249 1970

- Do not generate a uniform distribution.
- Use heuristics and target where the needle/bug is most likely to be in the haystack
- Generate the nastiest examples which are unlikely to occur in normal operation
- Once a counterexample is found, employ a shrinking strategy to search for a smaller one
- The power stems from statistical amplification and turning clock cycles into error-finding

Doing this well is (part of) the secret, which is the intellectual property of e.g., Quviq's commercial "Erlang QuickCheck" implementation.

Story

Around 10 years ago (2007), a major change in software development started:

Virtualization Fractional hardware

Leasing/Elasticity You don't own the machine

Containerization Standardized components

Problem statement

- Systems are now scattered over multiple machines
- Highly distributed
- You don't control the lifetime of systems(!)
- Must cope with the problem of cascading failures in components with dependencies:

$$A \rightarrow B \rightarrow C \tag{2}$$

If C fails, then resource buildup happens on B due to timeouts. Once B fails, cascades to A

Blame management

- Under cascading failure, often the wrong component is blamed
- Think it is A or B while the underlying problem is C
- Takes debugging time, often in production

- A “circuit breaker” is a pattern (Michael Nygard, Release It! 2007)
- Imagine a very big off-switch
- If a component is failing too often, hit the switch
- After a while try to re-enable the failed component

Consequence:

- Turn timeouts into fast errors
- No resource buildup
- Better response times (close to a couple microseconds)
- A client knows only about itself, the circuit has holistic knowledge
- Give the failing component a break so it can recover from transient errors
- Monitoring of broken circuits places blame accurately

Our problem

- Major outages due to failing cascades
- Want graceful degrade of system
- We had no direct control over all components, so it was not possible to fix all of them

Solution

Build an Erlang component, `fuse`, for handling circuit breaking:

```
Configuration = ...,
fuse:install(Name, Configuration)

case fuse:ask(Name) of
    ok ->
        normal_operation;
    blown ->
        return_error
end

%% When a system fails
ok = fuse:melt(Name)
```

Example

- A configured policy defines how many errors to tolerate
- Policy configures when to try re-enabling the circuit again
- Decouples policy from implementation and the protected component
- Policy code is not on the fast-path: better latency and efficiency of the system

Robustness

- Circuit breakers are critical components
- If there is a bug in the breaker, then the system is less robust overall
- In Erlang programming we say that it is part of the "error kernel" – the part of the system which absolutely must be correct for correct operation
- General robustness principle: trust few components. Use them to make the rest of the system robust

QuickCheck!

- Cost/Benefit analysis makes core components highly eligible for more extensive testing.
- QuickCheck is an excellent light-weight formal method: for relatively little work, you gain lots of robustness in the system
- Implementation time is roughly 3-5 times normal development
- But components have almost no maintenance afterwards

Strategy

Build software like the 'Dogme 95' manifesto for movies:

- Write the model in QuickCheck first
- Then write the code for the system
- Any feature must have a main user and use case (dogfood principle)
- If you can't specify the model, the feature is not going in

Result

- One minor bug in 3 years of operation
- Almost no reported issues
- Almost no maintenance
- The model is a specification
- Documentation is far easier to write with a good specification
- Extremely small code base
- Many bugs caught early on, when they are cheap to fix

Approach: first version

- Stateful model
- The major commands of a fuse is modeled (install, ask, melt)
- Pick a static configuration, and keep it simple (1 failure allowed in a 60 second window)
- Once the static configuration works, extend the code to arbitrary configurations
- Then implement minor commands (reset, remove)

Whiteboard time

```
%% Manual reset
reset(Name) ->
    fuse:reset(Name).

reset_pre(S) ->
    has_fuses_installed(S).

reset_args(_S) ->
    [g_name()].

%% (Postcondition)
reset_return(S, [Name]) ->
    case is_installed(Name, S) of
        true -> ok;
        false -> {error, not_found}
    end.
```

```
%% Resetting a fuse resets its internal state
reset_next(S, _V, [Name]) ->
    case is_installed(Name, S) of
        false -> S;
        true ->
          clear_blown(Name,
            clear_melts(Name,
              S))
     end.

reset_features(S, [Name], _V) ->
    case is_installed(Name, S) of
        false -> [{fuse_eqc, r05,
                   reset_uninstalled_fuse}];
        true -> [{fuse_eqc, r06,
                   reset_installed,
                      {blown, is_blown(Name, S)}}]
    end.
```

Feature sets

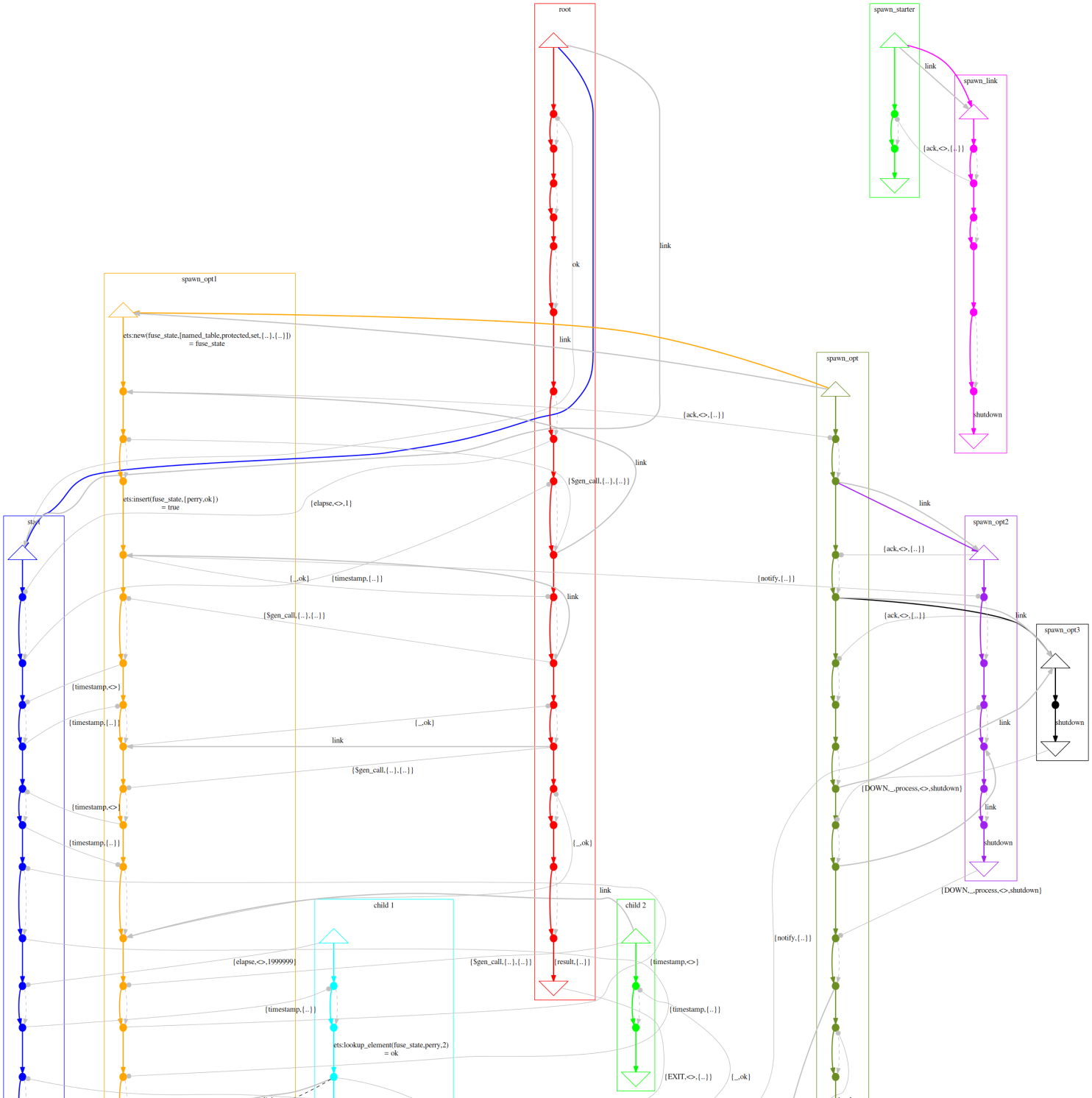
```
Group heal:
R01 - Heal non-installed fuse
R02 - Heal installed fuse

Group install:
R03 - Installation of a fuse with invalid conf
R04 - Installation of a fuse with valid conf

Group Reset:
R05 - Reset of an uninstalled fuse
R06 - Reset of an installed fuse
```

Approach: second version

- Stateful model, parallel version
- Ask Erlang QuickCheck to run the model in parallel with multiple clients asking at the same time
- One-line code change for this to happen(!)
- 1. Run the model sequentially up to a point
- 2. Run the model in parallel from this point on
- 3. Look for a linearization: If we can't find one, the code is wrong

Approach: third version

- Stateful model, parallel, PULSE
- Parallel test cases are hard to shrink
- Usually the error does not reappear under shrinking
- PULSE allows us to control the parallel schedule of the system
- Thus we can keep the bad interleave of concurrency while shrinking
- Uncovered an asynchronous bug in the system


root
spawn_starter
link
spawn_link
{ack,<>,{..}}
shutdown
link
ok
spawn_opt1
ets:new(fuse_state,[named_table,protected,set,{..},{..}])
= fuse_state
link
spawn_opt
{ack,<>,{..}}
link
{$gen_call,{..},{..}}
ets:insert(fuse_state,{perry,ok})
= true
{elapse,<>,1}
link
spawn_opt2
start
{ack,<>,{..}}
{_,ok}
{timestamp,{..}}
{notify,{..}}
link
{$gen_call,{..},{..}}
{ack,<>,{..}}
link
spawn_opt3
{timestamp,<>}
{timestamp,{..}}
{_,ok}
link
shutdown
link
{$gen_call,{..},{..}}
{timestamp,<>}
{DOWN,_,process,<>,shutdown}
link
{timestamp,{..}}
{_,ok}
shutdown
link
{DOWN,_,process,<>,shutdown}
child 1
child 2
{notify,{..}}
{elapse,<>,1999999}
{$gen_call,{..},{..}}
{result,{..}}
{timestamp,<>}
{timestamp,{..}}
{timestamp,{..}}
ets:lookup_element(fuse_state,perry,2)
= ok
{EXIT,<>,{..}}
{_,ok}

Approach: Current Version

- Clustered Component model
- 2 stateful models: one handles fuse, the other handles time
- Clustered into a complete system
- Time and timers are mocked - taken over by the model
- Advancing of time is controlled by the model and simulated
- Allows us to efficiently check timing related questions

Time injection is a key trick in most real-world-systems!

Mocking

- In addition to checking postconditions of the system:
- Check that the system calls the correct functions in a callout section
- Domain-specific-language for defining callout sequences and verifying their correctness

Clusters

- 2 Stateful models
- We can invoke commands in either model: either for advancing time, or for doing some command in fuse
- fuse calls into the timing model whenever it needs time or timing information
- We construct answers from the timing model to feed back into fuse
- We can do model-internal transitions: Change the state of the model without invoking a command in the system-under-test

In the timing module:

```
monotonic_time_callouts(#state {time = T }, []) ->
  ?CALLOUT(fuse_time, monotonic_time, [], T),
  ?RET(T).

monotonic_time_return(#state { time = T }, []) -> T.
```

In the fuse model:

```
melt_installed_callouts(_S, [Name]) ->
    ?MATCH(Ts, ?APPLY(fuse_time_eqc, monotonic_time, [])),
    ?APPLY(process_melt, [Name, Ts]),
    ?RET(ok).
```

- Apply the operations of another module
- Apply a model-internal transition in `process_melt`

(For the Haskell-nerds in the room: callouts is a monad)

Errors found

- Under development 15 errors were removed
- The model was guiding the implementation along, suggested better approaches
- The model forces you into simple, coherent solutions

Use

- Used in various industries around the world:
- Game servers
- Ad Tracking services
- Ad bidding services
- Database systems
- Some very big corporations are using it (top 10)