Property-based testing, race conditions, and QuickCheck

John Hughes
QuickCheck in a Nutshell

Properties → Test case → Minimal Test case

Diagram showing the process flow from properties to test cases and then to a minimal test case.
Benefits

• Less time spent writing test code
  – One property replaces many tests

• Better testing
  – Lots of combinations you’d never test by hand

• Less time spent on diagnosis
  – Failures minimized automagically
Tests for Base 64 encoding

```php
base64_encode(Config) when is_list(Config) ->
  %% Two pads
  <<"QWxhZGRpbjpvcGVuIHNlc2FtZQ==">> =
    base64:encode("Aladdin:open sesame"),

  %% One pad
  <<"SGVsbG8gV29ybGQ=">> = base64:encode(<<"Hello World">>),

  %% No pad
  "QWxhZGRpbjpvcGVuIHNlc2Ft" =
    base64:encode_to_string("Aladdin:open sesame"),

  "MDEyMzQ1Njc4OSFAIzBeJiooKTs6PD4sLiBbXXt9" =
    base64:encode_to_string(<<"0123456789!@#0^*();:<>,. []{}">>),
ok.
```

Writing a Property

```
prop_base64() ->
  ?FORALL(Data,list(choose(0,255)),
    base64:encode(Data)==???) .
```
Back to the tests...

base64_encode(Config) when is_list(Config) ->
  %% Two pads
  "QWxhZGRpbyjpcGVuIHNlc2FtZQ==" =
  base64:encode("Aladdin:open sesame"),

  %% One pad
  "SGVsbG8gV29ybGQ=" = base64:encode("Hello World"),

  %% No pad
  "QWxhZGRpbyjpcGVuIHNlc2Ft" =
  base64:encode_to_string("Aladdin:open sesam"),

  "MDEyMzQ1Njc4OSFAIzBeJiooKTs6PD4sLiBbXXt9" =
  base64:encode_to_string(
    "0123456789!@#0^*();:<>,. []{}"),
  ok.

Where did these come from?
Possibilities

• Someone converted the data by hand
• Another base64 encoder
• The same base64 encoder!

Use the other encoder as an oracle

Use an old version (or a simpler version) as an oracle
Round-trip Properties

prop_encode_decode() ->
  ?FORALL(L,list(choose(0,255)),
    base64:decode(base64:encode(L))
    == list_to_binary(L)).

What does this test?

• **NOT** a complete test—will not find a consistent misunderstanding of base64
• **WILL** find mistakes in encoder or decoder

Simple properties find a lot of bugs!
Let’s test some C!
Modelling in Erlang

API Calls → API Calls → API Calls → API Calls

Model state → Model state → Model state → Model state

postconditions
Example

```
```

1. `put 1`
2. `put 2`
3. `get`
4. `put 3`
5. `get`
Code Fragments

next_state_data(_, _, S, _, {call, _, put, [_, X]}) ->
    S#state{elements=S#state.elements++[X]};

next_state_data(_, _, S, _, {call, _, get, _}) ->
    S#state{elements=tl(S#state.elements)};

postcondition(_, _, S, {call, _, get, _}, Res) ->
    Res == hd(S#state.elements);

postcondition(_, _, S, {call, _, size, _}, Res) ->
    Res == length(S#state.elements);
prop_q() ->
    ?FORALL(Cmds, commands(?MODULE),
    begin
        {H, S, Res} = run_commands(?MODULE, Cmds),
        Res == ok
    end).

A QuickCheck Property
Let’s run some tests...
Lessons

• One property can find many bugs

• Shrinking makes diagnosis very simple
Doing it for real

• AUTOSAR
  – Embedded software in cars
  – ~50 standard components

• Test code from three suppliers against QuickCheck models

• 95 issues found already!

COM component: 500 pages of documents, 250 pages of C code... 25 pages of QuickCheck specification
Example: Mixed features

Standard CAN Id

Extended CAN Id

11 bits

29 bits

Priority: lowest number has highest priority

Example:
Extended Id 113 has higher priority than standard Id 114

Buffered higher priority messages should be sent first
Example: Mixed features

Standard CAN Id

Extended CAN Id

11 bits

29 bits

1 extended
0 standard

transmit,[1,112,[67],'CAN_OK'],
transmit,[2,113,[0],'CAN_BUSY'],
transmit,[3,114,[0],'CAN_BUSY'],
tx_confirmation,[1,112,[67]]

Check callouts: 112, 114 sent, why?

Force buffering

Trigger sending
"We know there is a lurking bug somewhere in the dets code. We have got 'bad object' and 'premature eof' every other month the last year. We have not been able to track the bug down since the dets files is repaired automatically next time it is opened."

Tobbe Törnqvist, Klarna, 2007
What is it?

- **Application**
  - Mnesia
  - Dets
  - File system

- **klarna**
  - Invoicing services for web shops
  - Distributed database: transactions, distribution, replication
  - Tuple storage

- **500+ people in 5 years**

**Race conditions?**
Imagine Testing This...

dispenser:take_ticket()
dispenser:reset()
A Unit Test in Erlang

test_dispenser() ->
    ok = reset(),
    1 = take_ticket(),
    2 = take_ticket(),
    3 = take_ticket(),
    ok = reset(),
    1 = take_ticket().

Expected results

BUT...
A Parallel Unit Test

- Three possible correct outcomes!
Another Parallel Test

- 42 possible correct outcomes!
Modelling the dispenser

- reset
- take
- take
- take

- 0
- 0
- 1
- 2

- ok
- 1
- 2
- 3
Parallel Test Cases

reset ➞ ok

take ➞ 1 ➞ take ➞ 3

take ➞ 2 ➞ 0 ➞ 0 ➞ 1 ➞ 2
prop_parallel() ->
  ?FORALL(Cmds, parallel_commands(?MODULE),
  begin
    start(),
    {H,Par,Res} =
    run_parallel_commands(?MODULE,Cmds),
    Res == ok)
  end).

Generate parallel test cases

Run tests, check for a matching serialization
Let’s run some tests
take_ticket() --> 1

2. take_ticket() --> 1

Result: no_possible_interleaving
dets

• Tuple store:
  \{Key, Value1, Value2...\}

• Operations:
  – insert(Table,ListOfTuples)
  – delete(Table,Key)
  – insert_new(Table,ListOfTuples)
  – ...

• Model:
  – List of tuples (almost)
QuickCheck Specification

<100 LOC

> 6,000 LOC
DEMO
Bug #1

Prefix:

open_file(dets_table)

Parallel:
1. insert(dets_table, [])
2. insert_new(dets_table, []) --> ok

Result: no_possible_interleaving

insert_new(Name, Objects) -> Bool

Types:
Name = name()
Objects = object() | [object()]
Bool = bool()
Prefix:
   open_file(dets_table, [{type, set}]) --> dets_table

Parallel:
1. insert(dets_table, {0, 0}) --> ok

2. insert_new(dets_table, {0, 0}) --> ...time out...

=ERROR REPORT==== 4-Oct-2010::17:08:21 ===
** dets: Bug was found when accessing table dets_table
Bug #3

Prefix:

open_file(dets_table, [{type, set}]) --> dets_table

Parallel:
1. open_file(dets_table, [{type, set}]) --> dets_table

2. insert(dets_table, {0, 0}) --> ok
   get_contents(dets_table) --> []

Result: no_possible_interleaving
Is the file corrupt?
Bug #4

Prefix:

```
open_file(dets_table, [{type, bag}])  -->  dets_table
close(dets_table)  -->  ok
open_file(dets_table, [{type, bag}])  -->  dets_table
```

Parallel:
1. `lookup(dets_table, 0)  -->  []`
2. `insert(dets_table, {0, 0})  -->  ok`
3. `insert(dets_table, {0, 0})  -->  ok`

Result: `ok`

**premature eof**
Bug #5

Prefix:

    open_file(dets_table, [{type, set}]) --> dets_table
    insert(dets_table, [{1, 0}]) --> ok

Parallel:
1. lookup(dets_table, 0) --> []
   delete(dets_table, 1) --> ok

2. open_file(dets_table, [{type, set}]) --> dets_table

Result: ok
false
bad object
"We know there is a lurking bug somewhere in the dets code. We have got 'bad object' and 'premature eof' every other month the last year."

Tobbe Törnqvist, Klarna, 2007

Each bug fixed the day after reporting the failing case
Before

- Files over 1GB?
- Rehashing?
- > 6 weeks of effort!

After

- Database with one record!
- 5 calls to reproduce
- < 1 day to fix
• Particularly good for finding feature interactions—such as race conditions

• 100% code coverage is only the beginning...
Property-based testing

- Finds bugs in everything it’s applied to!
Better Testing—Less Work