Software Testing with QuickCheck

Lecture 1
Properties and Generators

Testing

• How do we know software works?
  – We test it!
• "lists:delete removes an element from a list"

4> lists:delete(2, [1, 2, 3]).
   [1, 3]
5> lists:delete(4, [1, 2, 3]).
   [1, 2, 3]

• ... seems to work!
Automated Testing

- Testing accounts for ~50% of software cost!
- Automate it... write once, run often

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

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

Lots of code (35%)

Boring

Easy to miss something

Property-based testing

- Generalise test cases

```
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
```
Properties

• We test directly against a formal specification

?FORALL(N, int(), N*N >= 0)

Bound variable

Test case generator

Test oracle

More tests...

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

c.f. ?FORALL({I, L}, ..., ...)
The fault explained

lists:delete(2,[2,2])

lists:member(2,[2])

not true

false

Properties with preconditions

• The property holds provided \( L \) contains no duplicates

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

```prolog
39> eqc:quickcheck(examples:prop_delete()).
....................x........x..................x.x
......xx...........x....x....xx....x..x..........x.
x.........x..
OK, passed 100 tests
```

Skipped tests
**Custom generators**

- Why not *generate* lists without duplicates in the first place?

  \[
  \text{ulist(Elem)} \rightarrow \text{?LET}(L, \text{list}(\text{Elem}), \text{lists:usort}(L)).
  \]

- Use as $\text{?FORALL}(L, \text{ulist}(\text{int}(), \ldots))$

- Generators are an abstract data type 
  $\text{?LET}$ for sequencing

**Why was the error hard to find?**

- $I \in \text{int()}$
- $L \in \text{list(int())}$

  What is the probability that $I$ occurs in $L$—twice?

  \[
  \text{prop_delete()} \rightarrow \text{?FORALL}(I,L), (\text{int()}, \text{list(}\text{int}())\), \text{collect}(\text{lists:member}(I,L), \text{not lists:member}(I,\text{lists:delete}(I,L))).
  \]

```
34> eqc:quickcheck(examples:prop_delete()).
.................................................................
.................................................................
OK, passed 100 tests
88% false
12% true
true
```

Usually $I$ doesn’t even occur once
Generate relevant tests

- Ensure that \( I \) is a member of \( L \)
  - Generate it \textit{from} \( L \)

```plaintext
prop_delete_2() ->
  \(?\text{FORALL}(L, \text{list(int())}),
  \(?\text{FORALL}(I, \text{elements}(L),
    \text{not lists:member}(I, \text{lists:delete}(I, L))))\).
```

45> eqc:quickcheck(examples:prop_delete_2()).

\( xx.x.x.xx...x.x....x........xx.....\) Failed! \textbf{After 28 tests.}\n
\([-8, 0, 7, 0]\)
0
Shrinking... (3 times)
[0, 0]
0

Documenting misconceptions

- Useful to record that an expected property is \textit{not} true

```plaintext
prop_delete_misconception() ->
  \textbf{fails(}
  \(?\text{FORALL}(L, \text{list(int())}),
  \?\text{IMPLIES}(L /= [],
    \?\text{FORALL}(I, \text{elements}(L),
      \text{not lists:member}(I, \text{lists:delete}(I, L))))).
```

49> eqc:quickcheck(examples:prop_delete_misconception()).

\( xx.x.x....x..........\) OK, failed as expected. \textbf{After 19 tests.}\n
**Good distribution ensures we falsify the property quickly**
Remember!

• We test against a formal specification!
  – Often it is the specification which is wrong!

• We don’t see the test data!
  – 100 passing tests can give a false sense of security

• Collect statistics!
  – Ensure a good test case distribution

Exercises
Software Testing with QuickCheck

Lecture 2
Symbolic Test Cases

Testing Data Structure Libraries

- **dict** — purely functional key-value store
  - `new()` — create empty dict
  - `store(Key,Val,Dict)`
  - `fetch(Key,Dict)`
  - ...

- Complex representation... just test the API
  - "black box testing"
  - In contrast to testing e.g. dict invariants
A Simple Property

- Perhaps the keys are unique...

```prolog
prop_unique_keys() ->
    ?FORALL(D, dict(),
        no_duplicates(dict:fetch_keys(D))).
```

- Cannot test this without a *generator for dicts*

---

Generating dicts

- Black box testing ➔ *use the API to create dicts*

```
dict() ->
    ?LAZY(
        oneof([dict:new(),
            ?LET({K, V, D}, {key(), value(), dict(),
            dict:store(K, V, D)})])
    ).
```

- We simulate lazy evaluation where we need it
Let’s test it!

- **Black box testing**: we need to know how this was constructed!

Symbolic Test Cases

dict() -> ?LAZY(oneof([ {call,dict,new,[]}, {call,dict,store,[key(),value(),dict()]} ])).

- \{call,M,F,Args\} represents a function call M:F(...Args...) symbolically.

prop_unique_keys() -> ?FORALL(D,dict()), no_duplicates(dict:fetch_keys(eval(D)))).

- eval(X) evaluates symbolic calls in a term X
Let’s test again!

Shrinking the structure

- QuickCheck has many shrinking operations; here we use

```
?LETSHRINK([X1,X2,...],
[G1,G2,...],
Result(X1,X2,...))
```

- Binds X1, X2... to values generated by G1, G2...
- Generates Result
- Can shrink to any of the Xi
dict() with Shrinking

```
dict() ->
  ?LAZY(oneof(
      [{call,dict,new,[]}],
      ?LETSHRINK([D],[dict()],
        {call,dict,store,[key(),value(),D]})))
```

- ?LETSHRINK makes shrinking recursive types very easy

Let’s test and shrink!

Nice shrinking result

-1 and -1.0, eh?
Testing vs. an Abstract Model

• How should \texttt{dict} operations behave?
  – The ”same” as a list of key-value pairs
  – Use the \texttt{proplists} library as a reference
• Make comparisons in the ”model” world

![Diagram](model(Dict) \rightarrow \texttt{dict:to_list}(Dict).

Returns list of key-value pairs

Commuting Diagrams

![Diagram](Dict \xrightarrow{\texttt{dict:store}(K,V,...)} Dict + \{K,V\}

Dict \xrightarrow{\texttt{model}} List

List \xrightarrow{\texttt{model_store}(K,V,...)} \{[K,V] | List\}

Hoare: \textit{Proof of Correctness of Data Representations}, 1972
Testing store

```erlang
prop_store() ->
  ?FORALL({K,V,D},
    {key(),value(),dict()},
    begin
      Dict = eval(D),
      model(dict:store(K,V,Dict)) ==
      model_store(K,V,model(Dict))
    end).
```

```erlang
model_store(K,V,L) -> [{K,V}|L].
```

Next Steps...

- Write similar properties for all the dict operations
- Extend the dict generator to use all the dict-returning operations
  - Each property tests many operations
- ...and, of course, correct the specification!
Debugging properties

• Why is a property false?
  – We need more information!

Exercises
Software Testing with QuickCheck

Lecture 3
Testing Stateful Systems

The Process Registry

• Erlang provides a local name server to find node services
  – register(Name,Pid)—associate Pid with Name
  – unregister(Name)—remove any association for Name
  – whereis(Name)—look up Pid associated with Name

• Another key-value store!
  – Test against a model as before
Stateful Interfaces

- The state is an *implicit* argument and result of every call
  - We cannot *observe* the state, and map it into a model state
  - We can *compute* the model state, using state transition functions
  - We detect test failures by observing the *results* of API calls

Stateful Test Cases

- Test cases are sequences of (symbolic) API calls

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

The model behaviour is defined by *callbacks* in this module

Check they ran OK
Generating Commands

- We generate *symbolic calls* as before:

```erlang
command(S) ->
    oneof([{'call', erlang, register, [name()], pid(S)}],
          [{'call', erlang, unregister, [name()]},
           {'call', erlang, whereis, [name()]},
           {'call', ?MODULE, spawn, []}]).
```

- But what is `pid()`?
- Pids must be dynamically created in each test
  - Intermediate results must be saved in the state and reused

The Model State

- The model must track *both* the key-value pairs, and the spawned processes

```erlang
-record(state, {pids=[], % pids spawned in this test
               regs=[], % list of {Name,Pid} pairs
               }).
```

- Now Pids can be generated from the state

```erlang
pid(S) -> elements(S#state.pids).
```
State Transitions

- Specify behaviour of the model

```erlang
initial_state() -> #state{}.
```

```erlang
next_state(S, V, {call, _, spawn, _}) ->
    S#state{pids = [V | S#state.pids]};
next_state(_, V, {call, _, spawn, _}) ->
    S#state{pids = [V | S#state.pids]};
next_state(S, V, {call, _, register, [Name, Pid]}) ->
    S#state{regs = [{Name, Pid} | S#state.regs]};
next_state(_, V, {call, _, register, [Name, Pid]}) ->
    S#state{regs = [proplists:delete(Name, S#state.regs)]};
next_state(S, V, {call, _, unregister, [Name]}) ->
    S#state{regs = proplists:delete(Name, S#state.regs)};
next_state(_, V, {call, _, unregister, _}) ->
    S.
```

- Much like the model functions of the previous lecture

Let’s Test It!

```erlang
40> eqc:quickcheck(reg_eqc:prop_registry()).
Failed! Reason: {'EXIT',[{eqc,elements,[]}]
After 1 tests.
false
41>
```

- `pid(S)` raises an exception if `S#state.pids` is empty
Conditional Generation

• A little trick makes it easy to include a generator only under certain conditions

\[
\text{command}(S) \rightarrow
\begin{cases}
\text{oneof}([\{\text{call,erlang,register,}[\text{name()}],\text{pid}(S)]
\quad |\quad S\#\text{state.pids}=[]] ++
\{\text{call,?MODULE,unregister,}[\text{name()}],
\{\text{call,erlang,whereis,}[\text{name()}],
\{\text{call,?MODULE,spawn,[]}\}
\})
\end{cases}
\]

– Since \([X \mid \text{true}] = [X], [X \mid \text{false}] = []\)

Let’s Test It!

Just one command!
Preconditions

- Preconditions can be specified for each operation, in terms of the model state

```prolog
precondition(S, {call, _, unregister, [Name]}) ->
  unregister_ok(S, Name);
precondition(_S, {call, _, _, _}) ->
  true.

unregister_ok(S, Name) ->
  proplists:is_defined(Name, S#state.regs).
```

- Generated test cases satisfy all preconditions

Postconditions

- Postconditions are checked after each API call, with access to the actual result

```prolog
postcondition(S, {call, _, unregister, [Name]}, Res) ->
  case Res of
    ['EXIT', _] ->
      not unregister_ok(S, Name);
    true ->
      unregister_ok(S, Name)
  end;
postcondition(_S, {call, _, _, _}, _Res) ->
  true.

unregister(Name) ->
  catch
    erlang:unregister(Name).

command(S) ->
  oneof([..{call, ?MODULE, unregister, [name()]}, ...]).
```
Let’s Test It!

The same property can reveal many different bugs

Typical Industrial Scenario

• QuickCheck finds a minimal sequence of messages that provokes a failure
3G Radio Base Station

- Setup
- OK
- Setup
- Reject

Media Proxy

- Multimedia IP-telephony (IMS)
- Connects calls across a firewall
- Test adding and removing callers from a call
Exercises