QuickCheck Tutorial

Thomas Arts
John Hughes

Quviq AB

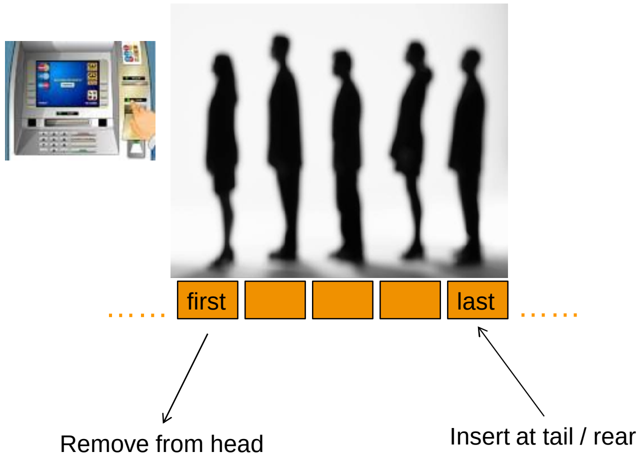
Queues

Erlang contains a queue data structure (see stdlib documentation)

We want to test that these queues behave as expected

> What is “expected” behaviour?
>
> We have a mental model of queues that the software should conform to.

Mental model of a fifo queue



Traditional test cases could look like:

```
Q0 = queue:new(),
Q1 = queue:cons(1,Q0),
1 = queue:last(Q1).
```

We want to check for arbitrary elements that **if we add an element, it's there**.

```
Q0 = queue:new(),
Q1 = queue:cons(8,Q0),
Q2 = queue:cons(0,Q1),
0 = queue:last(Q2),
```

We want to check for arbitrary queues that **last added element is "last"**

Property is like an abstraction of a test case

QuickCheck property

We want to know that for any element, when we add it, it's there

```
prop_itsthere() ->
   ?FORALL(I,int(),
           I == queue:last(
                   queue:cons(I,
                      queue:new()))).
```


QuickCheck property

We want to know that for any element, when we add it, it's there

```
prop_itsthere() ->
   ?FORALL(I,int(),
           I == queue:last(
                   queue:cons(I,
                      queue:new()))).
```

This is a property
Test cases are generasted from such properties

QuickCheck property

We want to know that for any element, when we add it, it's there

int() is a generator for random integers

```
prop_itsthere() ->
   ?FORALL(I,int(),
           I == queue:last(
                   queue:cons(I,
                      queue:new()))).
```

I represents one randomly chosen integer

A boolean expression. If false, the test fails

QuickCheck

Run QuickCheck

```
1> eqc:quickcheck(queue_eqc:prop_itsthere()).
..................................................
.................................................
OK, passed 100 tests
true
2>
```

but we want more variation in our test data...

Build a symbolic representation for a queue

This representation can be used to both **create the queue** and to **inspect queue creation**

Why Symbolic?

1. We want to be able to see how a value is created as well as its result
2. We do not want tests to depend on a specific representation of a data structure
3. We want to be able to manipulate the test itself

Generating random symbolic queues

```
queue() ->


queue(0) ->
  {call,queue,new,[]};
queue(Size) ->
  oneof([queue(0),
         {call,queue,cons,[int(),queue(Size-1)]}]).
```

oneof is a QuickCheck primitive to choose a random element from a list

Generating random symbolic queues

```
queue() ->
   ?SIZED(Size,queue(Size)).

queue(0) ->
  {call,queue,new,[]};
queue(Size) ->
  oneof([queue(0),
         {call,queue,cons,[int(),queue(Size-1)]}]).
```


Generating random symbolic queues

```
eqc_gen:sample(queue_eqc:queue()).
{call,queue,cons,[-8,{call,queue,new,[]}]}
{call,queue,new,[]}
{call,queue,
      cons,
      [12,
       {call,queue,
             cons,
             [-5,
              {call,queue,
                    cons,
                    [-18,{call,queue,cons,[19,{call,queue,new,[]}]}]}]}]}
{call,queue,
      cons,
      [-18,
       {call,queue,cons,[-11,{call,queue,cons,
                                [-18,{call,queue,new,[]}]}]}]}
```

A more general property

```
prop_cons() ->
 ?FORALL({I,Q},{int(),queue()},
          queue:last(queue:cons(I,eval(Q))) == I).
```

```
eqc:quickcheck(queue_eqc:prop_cons_tail()).
...Failed! After 4 tests.
{3,{call,queue,cons,[-1,{call,queue,new,[]}]}}
Shrinking..(2 times)
{0,{call,queue,cons,[1,{call,queue,new,[]}]}}
false
```

clear how queue is created

Symbolic representation helps to understand test data

Symbolic representation helps in manipulating test data (e.g. shrinking)

Model Queue

Compare to traditional test cases:

```
        REAL DATA                      MODEL

Q0 = queue:new(),                      []
Q1 = queue:cons(1,Q0),                 [1]
Q2 = queue:cons(2,Q1),                 [1,2]
2 = queue:last(Q2).                       ↑ (inspect)

Q0 = queue:new(),                      []
Q1 = queue:cons(8,Q0),                 [8]
Q2 = queue:cons(0,Q1),                 [8,0]
 0 = queue:last(Q2);.                     ↑ (inspect)
```


Model Queue

Do we understand queues correctly: what is first and what last?

```
prop_cons() ->
   ?FORALL({I,Q},{int(),queue()},
           model(queue:cons(I,eval(Q)) ==
                         model(eval(Q)) ++ [I]).
```

Write a model function from queues to list

(or use the function queue:to_list, which is already present in the library)

Model Queue property

```
eqc:quickcheck(queue_eqc:prop_cons()).
...Failed! After 4 tests.
{0,{call,queue,cons,[1,{call,queue,new,[]}]}}
false
```


Queue manual page

cons(Item, Q1) -> Q2

Types: **Item = term(), Q1 = Q2 = queue()**
Inserts Item at the head of queue Q1. Returns the new queue Q2.

head(Q) -> Item

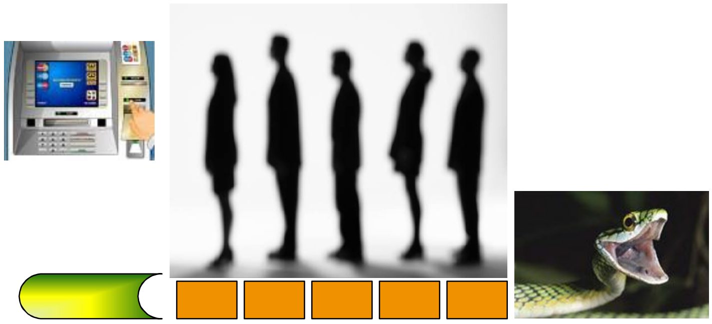
Types: **Item = term(), Q = queue()**
Returns Item from the head of queue Q.
Fails with reason empty if Q is empty.

last(Q) -> Item

Types: **Item = term(), Q = queue()**
Returns the last item of queue Q. This is the opposite of head(Q).
Fails with reason empty if Q is empty.

Queue

Mental model of a fifo queue



tail head

tail head

Model Queue

Change property to express new understanding

```
prop_cons() ->
   ?FORALL({I,Q},{int(),queue()},
           model(queue:cons(I,eval(Q)) ==
                        [I | model(eval(Q))]).
```

```
eqc:quickcheck(queue_eqc:prop_cons()).
..................................................
..................................................
OK, passed 100 tests
true
```

Add properties

```
prop_cons() ->
   ?FORALL({I,Q},{int(),queue()},
           model(queue:cons(I,eval(Q))) == [I | model(eval(Q))]).

prop_last() ->
   ?FORALL(Q,queue(),
      begin
        QVal = eval(Q),
        queue:is_empty(QVal) orelse
                 queue:last(QVal) == lists:last(model(QVal))
      end).
```

similar `queue:head (Qval) == hd(model(Qval))`

There are more constructors for queues, e.g., **tail**, snoc, in, out, etc. All constructors should respect queue model

We need to

1) add all queue constructors to the generator
2) add a property for each constructor / destructor

Tail removes last added element from the queue

```
queue() ->
   ?SIZED(Size,queue(Size)).

queue(0) ->
  {call,queue,new,[]};
queue(Size) ->
  oneof([queue(0),
         {call,queue,cons,[int(),queue(Size-1)]},
         {call,queue,tail,[queue(Size-1)]}
        ]).
```


Check properties again

```
eqc:quickcheck(queue_eqc:prop_cons()).
...Failed! Reason:
{'EXIT',{empty,[{queue,tail,[{[],[]}]},
                {queue_eqc,'-prop_cons2/0-fun-0',1},
                 ...
After 4 tests.
{0,{call,queue,tail,[{call,queue,new,[]}]}}
false
```

cause immediately clear: advantage of symbolic representation

Only generate well defined queues

```
queue() ->
   ?SIZED(Size,well_defined(queue(Size))).

defined(E) ->
   case catch {ok,eval(E)} of
        {ok,_} -> true;
        {'EXIT',_} -> false
   end.

well_defined(G) ->
  ?SUCHTHAT(X,G,defined(X)).
```


- recursive data type requires recursive generators (use QuickCheck size control)
- symbolic representation make counter examples readable
- Define property for each data type operation: compare result operation on real queue and model

  model(queue:operator(Q)) == model_operator(model(Q))

- Only generate well-defined data structures
  (properties spot error for those undefined)

Side effects

Real software contains more than data structures

What if we have side-effects?

Queue server

We build a simple server around the queue data structure

```
new() ->
  spawn(fun() -> loop(queue:new()) end).

loop(Queue) ->
  receive
    {cons,Element} ->
        loop(queue:cons(Element,Queue));
    {last,Pid} ->
        Pid ! {last,queue:last(Queue)},
       loop(Queue)
  end.
```

Queue server

Some interface functions:

```
cons(Element,Queue) ->
  Queue ! {cons,Element}.

last(Queue) ->
  Queue ! {last,self()},
  receive
    {last,Element} -> Element
  end.
```


Side effects

The state is hidden, can only be inspected by inspecting the effects of operations on the state

Same property no longer usable:

```
prop_last() ->
   ?FORALL(Q,queue(),
      begin
        QVal = eval(Q),
        queue:is_empty(QVal) orelse
               queue:last(QVal) == lists:last(model(QVal))
      end).
```

"eval" should now be replaced by sending messages

State cannot be inspected that easy!

State Machine model

State machines are ideal to model systems with side-effects. We model what we believe the state of the system is and check whether action on real state have same effect as on the model.

```
-record(state,{ref,model}).

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

Events for state transitions are defined by the interface commands

State Machine model

```
command(S) ->
  oneof([{call,?MODULE,new,[]},
         {call,?MODULE,cons,[int(),S#state.ref]}]).

next_state(S,V,{call,_,new,[]}) ->
  S#state{ref = V, model = []};
next_state(S,V,{call,_,cons,[E,_]}) ->
  S#state{model = S#state.model++[E]}.
```

but we should **not** send a "cons" message to an undefined process...

The same mistake, although we should understand the model now!

We use preconditions to eliminate unwanted sequences of messages

```
precondition(S,{call,_,new,[]}) ->
  S#state.ref == undefined;
precondition(S,{call,_,cons,[E,Ref]}) ->
  Ref /= undefined.
```

With this state machine, we can generate random sequences of messages to our server (with random data in the messages)

Property:

```
prop_last() ->
   ?FORALL(Cmds,commands(?MODULE),
      begin
        {H,S,Res} = run_commands(?MODULE,Cmds),
         CorrectLast = [     ] ,
        cleanup(S#state.ref),
        Res == ok andalso CorrectLast
      end).

cleanup(undefined) ->
  ok;
cleanup(Pid) ->
  exit(Pid,kill)
```

```
S#state.model==[] orelse
     last(S#state.ref) ==
         lists:last(S#state.model)
```

State Machine model

Run QuickCheck

```
5> eqc:quickcheck(queue_eqc:prop_last()).
.Failed! Reason:
{'EXIT',{badarg,[{queue_eqc,last,1},
                 {queue_eqc,'-prop_last/0-fun-2-',1},
                 ....]}}
After 2 tests.
[]
false
```

we need to make sure that server is started!

Idea: why not put "last"
in the sequences

State Machine model

```
command(S) ->
  oneof([{call,?MODULE,new,[]},
         [{call,?MODULE,cons,[int(),S#state.ref]},
         [{call,?MODULE,last,[S#state.ref]}]).

next_state(S,V,{call,_,new,[]}) ->
  S#state{ref = V, model = []};
next_state(S,V,{call,_,cons,[E,_]}) ->
  S#state{model = S#state.model++[E]};
next_state(S,V,{call,_,last,[_]}) ->
  S.
```

State Machine model

```
precondition(S,{call,_,new,[]}) ->
  S#state.ref == undefined;
precondition(S,{call,_,cons,[E,Ref]}) ->
  Ref /= undefined;
precondition(S,{call,_,last,[Ref]}) ->
  Ref /= undefined.

postcondition(S,{call,_,last,[Ref]},R) ->
   S#state.model==[] orelse
      R == lists:last(S#state.model);
postcondition(S,_,_) ->
    true.
```


State Machine model

Property:

```
prop_last() ->
   ?FORALL(Cmds,commands(?MODULE),
     begin
       {H,S,Res} = run_commands(?MODULE,Cmds),
       cleanup(S#state.ref),
       Res == ok
     end).
```

Checking of property fails, since we add the element at the tail.

State Machine model

6> eqc:quickcheck(queue_eqc:prop_last()).

......Failed! After 7 tests.

```
[{set,{var,1},{call,queue_eqc,new,[]}},
 {set,{var,2},{call,queue_eqc,cons,[-1,{var,1}]}},
 {set,{var,3},{call,queue_eqc,cons,[1,{var,1}]}},
 {set,{var,4},{call,queue_eqc,last,[{var,1}]}},
 {set,{var,5},{call,queue_eqc,cons,[0,{var,1}]}},
 {set,{var,6},{call,queue_eqc,cons,[-1,{var,1}]}},
 {set,{var,7},{call,queue_eqc,last,[{var,1}]}},
 {set,{var,8},{call,queue_eqc,last,[{var,1}]}},
 {set,{var,9},{call,queue_eqc,cons,[1,{var,1}]}},
 {set,{var,10},{call,queue_eqc,cons,[1,{var,1}]}},
 {set,{var,11},{call,queue_eqc,last,[{var,1}]}},
 {set,{var,12},{call,queue_eqc,cons,[0,{var,1}]}}]
```

{postcondition,false}

Shrinking....(4 times)

State Machine model

6> eqc:quickcheck(queuesdemo:prop_last()).

......Failed! After 7 tests.

```
[{set,{var,1},{call,queue_eqc,new,[]}},
 {set,{var,2},{call,queue_eqc,cons,[-1,{var,1}]}},
 {set,{var,3},{call,queue_eqc,cons,[1,{var,1}]}},
 {set,{var,4},{call,queue_eqc,last,[{var,1}]}},
 {set,{var,5},{call,queue_eqc,cons,[0,{var,1}]}},
 {set,{var,6},{call,queue_eqc,cons,[-1,{var,1}]}},
 {set,{var,7},{call,queue_eqc,last,[{var,1}]}},
 {set,{var,8},{call,queue_eqc,last,[{var,1}]}},
 {set,{var,9},{call,queue_eqc,cons,[1,{var,1}]}},
 {set,{var,10},{call,queue_eqc,cons,[1,{var,1}]}},
 {set,{var,11},{call,queue_eqc,last,[{var,1}]}},
 {set,{var,12},{call,queue_eqc,cons,[0,{var,1}]}}]
```

{postcondition,false}

Shrinking....(4 times)

State Machine model

Shrinking....(4 times)

```
[{set,{var,1},{call,queue_eqc,new,[]}},
 {set,{var,2},{call,queue_eqc,cons,[0,{var,1}]}},
 {set,{var,3},{call,queue_eqc,cons,[1,{var,1}]}},
 {set,{var,4},{call,queue_eqc,last,[{var,1}]}}]
{postcondition,false}
false
```

Thus, we see the same misunderstanding of queue behaviour, even though we cannot inspect the state of the system directly.

Summary of Important Points

- Code with side effects can often be modeled by a state machine
- 2-stage process
  - Generation of *symbolic* tests
  - Execution
- Tests must be *independent*—start in a known state and clean up!

C code

System under test can also be written in a different language than Erlang

. As long as we can interface to it, we can use QuickCheck to test that system.

For example, C++ implementation of a queue

http://www.cplusplus.happycodings.com/Beginners_Lab_Assignments/

Queue in C++

```
# define SIZE 20

#include <stdio.h>
#include "eqc.h"

class queue
{
    int a[SIZE];
    int front, rear;
public:
    queue();
    ~queue();
    int insert(int i),  remove(), isempty(), isfull();
    last();
};
```

We add our functions/macros

Not present, we add this

Queue in C++

```
queue::queue()
{
   front=0, rear=0;
}

int queue::insert(int i)
{
   if(isfull())
   {
       return 0;
   }
   a[rear] = i;
   rear++;
   return 1;
}
```

```
int queue::last()
{
   if(isempty())
       return (-9999);
   else
       return(a[rear]);
}
```

We add this to the implementation

Queue in C++

Different ways to connect to C code, we choose to generate C source code for this example

```
void quicktest()
{
#include "generated.c"
}

void main()
{
   open_eqc();
   quicktest();
   close_eqc();
}
```

1. Open a file to write results to
2. Perform a test
3. Close the file

Recall the property we had defined before

```
prop_last() ->
   ?FORALL(Cmds,commands(?MODULE),
     begin
       {H,S,Res} = run_commands(?MODULE,Cmds),
       cleanup(S#state.ref),
       Res == ok
     end).
```

Now we run a C program instead

In our case, ending C program is sufficiently cleanup

Property

```
prop_last() ->
   ?FORALL(Cmds,commands(?MODULE),
     begin
       CCode = evaluate(Cmds),
       Vals = run(CCode),
       postconditions(?MODULE,Cmds,Vals)
     end).
```

Let each interface function return C code

We now check the postconditions after the complete run

Calling C from QuickCheck property

The interface functions return C code:

```
new() ->
  "queue q;\nINT(1);\n".

cons(Element,Queue) ->
  io_lib:format("INT(q.insert(~p));\n",[Element]).

last(Queue) ->
   "INT(q.last());\n".
```


Calling C from QuickCheck property

```
run(CCode) ->
    file:delete("to_eqc.txt"),
    ok = file:write_file("generated.c",CCode),
    %% Windows with visual studio
    String = os:cmd("cl main.cpp"),
    case string:str(String,"error") of
         0 ->
            case String of
                [] ->
                   exit({make_failure, "Start Erlang with correct env"});
                _ -> ok
            end;
         _ -> exit({compile_error,String})
    end,
    os:cmd("main.exe"),
    {ok,Vals} = file:consult("to_eqc.txt"),
    Vals.
```

QuickCheck a C program

Ok, let us run QuickCheck then:

```
31> eqc:quickcheck(queue_eqc:prop_last()).
Failed! Reason:
{'EXIT',postcondition}
After 1 tests.
....
Shrinking.(1 times)
Reason:
{'EXIT',postcondition}
[{set,{var,1},{call,queue_eqc,new,[]}},
 {set,{var,2},{call,queue_eqc,cons,[0,{var,1}]}},
 {set,{var,3},{call,queue_eqc,last,[{var,1}]}}]
returned from C: [1,1,4199407]
false
```


QuickCheck a C program

```
queue::queue()
{
  front=0, rear=0;
}

int queue::insert(int i)
{
  if(isfull())
  {
      return 0;
  }
  a[rear] = i;
  rear++;
  return 1;
}
```

```
int queue::last()
{
   if(isempty())
       return (-9999);
   else
       return(a[rear]);
}
```

rear-1 of course

Correct the error and run QuickCheck again

```
32> eqc:quickcheck(queuesdemo:prop_last()).
...................Failed! Reason:
{'EXIT',postcondition}
After 20 tests.
```

...(long sequence with 57 commands)...

Oh... yes, Erlang may hold the lock on the file ... Vista and duo core..%@!&...

Reason:

```
{'EXIT',{compile_error,"main.cpp\r\nMicrosoft (R) Incremental
   Linker Version 9.00.21022.08\r\nCopyright (C) Microsoft
   Corporation.  All rights reserved.\r\n\r\n/out:main.exe
   \r\nmain.obj \r\nLINK : fatal error LNK1104: cannot open file
   'main.exe'\r\n"}}
[{set,{var,1},{call,queuesdemo,new,[]}},
 {set,{var,4},{call,queuesdemo,last,[{var,1}]}}]
false
```


Once more, now a yield in the `run` function. Failure after about 20 tests, shrinking to:

```
{'EXIT',postcondition}
[{set,{var,1},{call,queue_eqc,new,[]}},
 {set,{var,2},{call,queue_eqc,cons,[0,{var,1}]}},
 {set,{var,5},{call,queue_eqc,cons,[0,{var,1}]}},
 .... (21 inserts in total,but last is insert 1) ...
 {set,{var,37},{call,queue_eqc,cons,[1,{var,1}]}},
 {set,{var,41},{call,queue_eqc,cons,[0,{var,1}]}},
 {set,{var,42},{call,queue_eqc,last,[{var,1}]}}]
returned from C: [1,1,1,1,......,1,0,1]
false
```

Our model does not take into account that the C queue has bounded size! (max 20 elements)

Summary

- The same specification can be used to test several implementations even in different languages.
- Writing C source code is only an alternative when experimenting. Use ports or C nodes for real situations.

Conclusion

QuickCheck makes testing fun....

.... and ensures a high quality of your products

