Software Testing with QuickCheck

Lecture 1

Properties and Generators

Testing

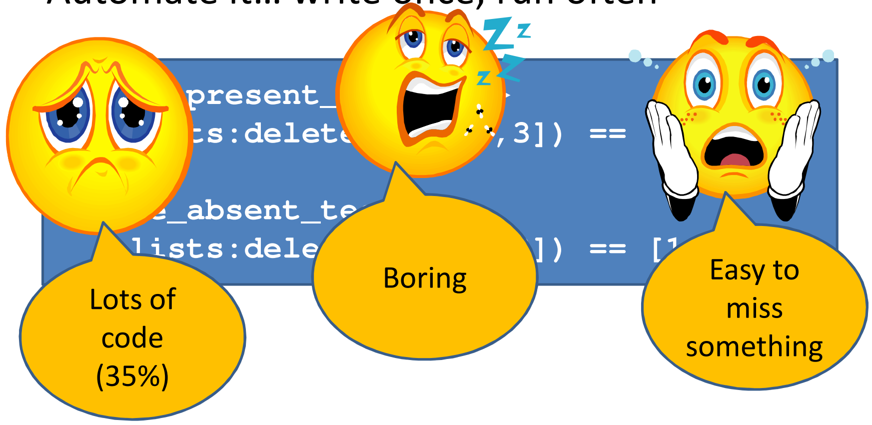
- How do we know software works?
  - We test it!
- ”lists:delete removes an element from a list”

```
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



Property-based testing

- Generalise test cases

$\forall\{I,L\} \in \mathrm{int}() \times \mathrm{list}(\mathrm{int}())$

```
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

Bound variable

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

Test case generator

Test oracle

- We test directly against a formal specification

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
```

A failed test

A simplest failing test

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 h… duplicates

```
no_duplicates(L) ->
    lists:usort(L)
  == lists:sort(L).
```

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

```
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

Q…

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

```
ulist(Elem) ->
    ?LET(L,list(Elem),
         lists:usort(L)).
```

First: generate a list **L**

Then: sort it and remove duplicates

- Use as **?FORALL(L,uli** … **)**
- Generators are an *abstrac* … **?LET** for sequencing

Why was the error hard to find?

Q…

- $I \in$ int()
- $L \in$ list(int())

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

```
prop_delete() ->
  ?FORALL({I,L},
          {int(),list(int())},
          collect(lists:member(I,L),
                  not lists:member(I,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 *from* **L**

```
prop_delete_2() ->
  ?FORALL(L,list(int()),

     ?FORALL(I,elements(L),
       not lists:member(I,lists:delete(I,L)))).
```

```
45> eqc:quickcheck(examples:prop_delete_2()).
.xx.x.x.xx...x.x...x....x.......xx.....Failed! After 28 tests.
[-8,0,7,0]
0
Shrinking...(3 times)
[0,0]
0
```

Documenting misconceptions

- Useful to record that an expected property is *not* true

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

```
49> eqc:quickcheck(examples:prop_delete_misconception()).
x...x.x....x..........OK, failed as expected. After 19
tests.
```

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…

```
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))])
       ).
```

A c

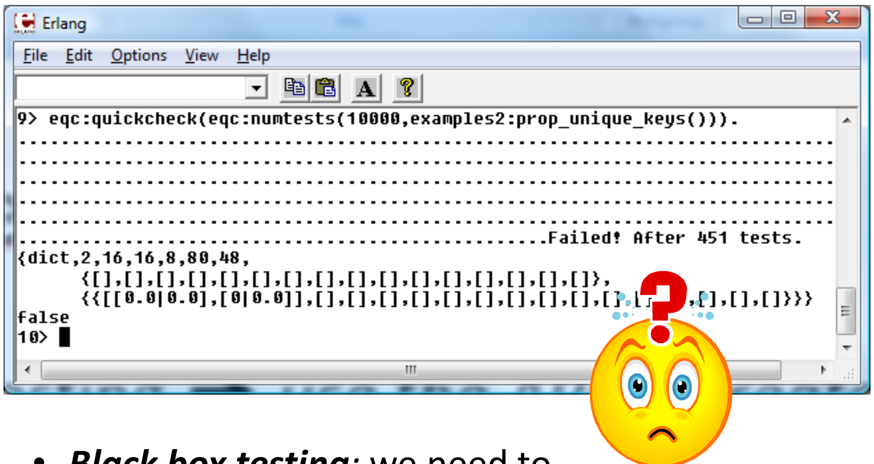
```
oneof([int(),real(),atom()]).
>
```

recursion!

Constant time

- 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()]}
  ])).
```

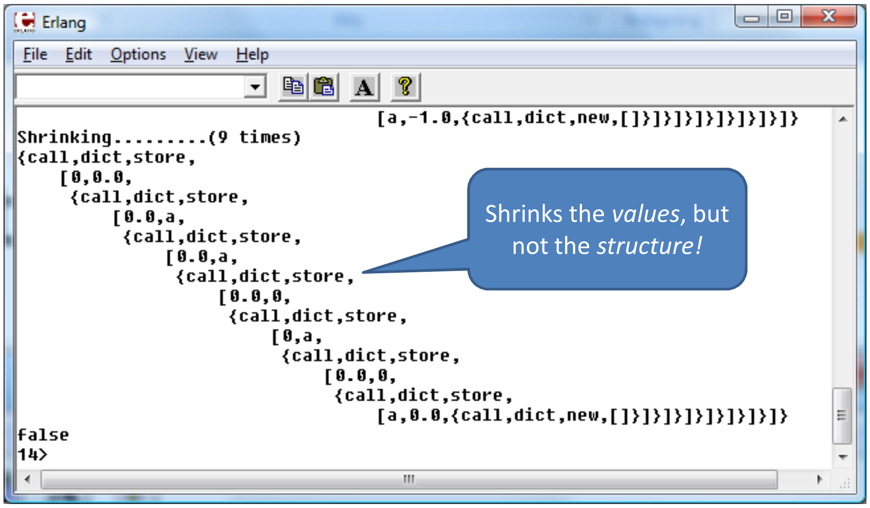
- {call,M,F,Args} *represents* a function call M:F(…Args…) symbolically.

Test case is now *symbolic*

```
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,…))
```

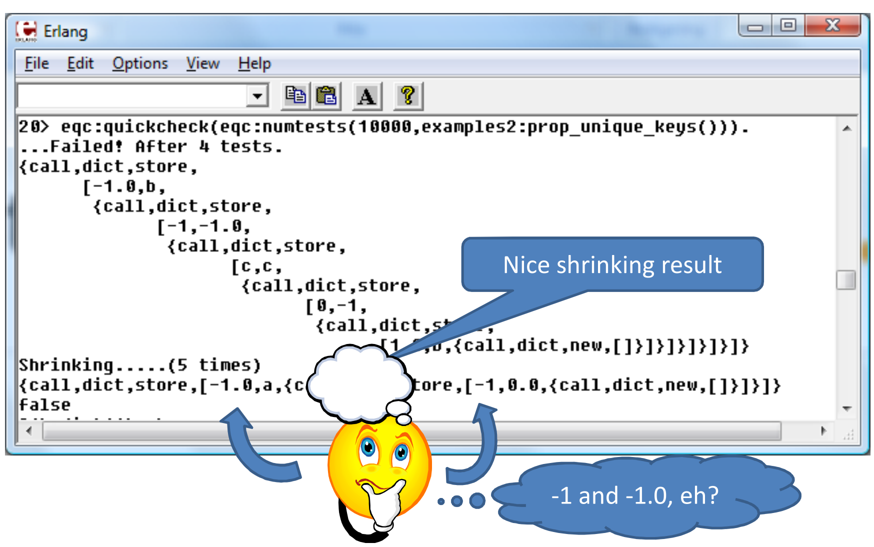
- Binds X1, X2… to values generated by G1, G2…
- Generates Result
- Can *shrink* to any of the Xi

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!



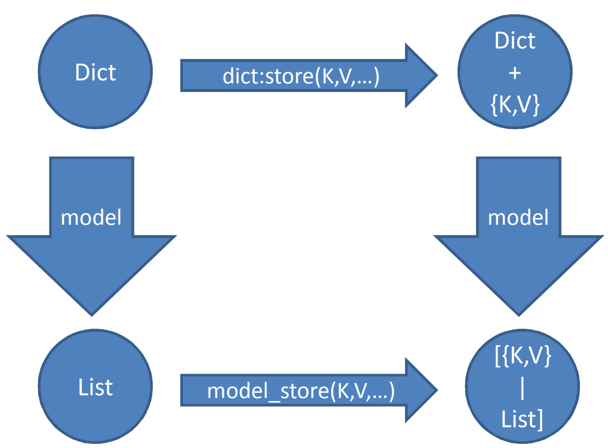
Testing vs. an Abstract Model

- How *should* **`dict`** operations behave?
  - The ”same” as a list of key-value pairs
  - Use the **`proplists`** library as a reference
- Make comparisons in the ”model” world

```
model(Dict) ->
    dict:to_list(Dict).
```

Returns list of key-value pairs

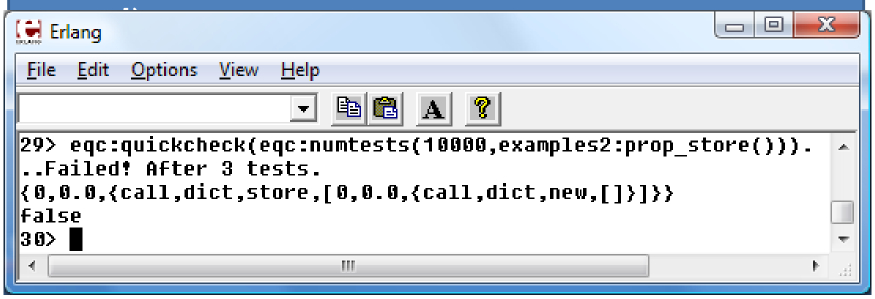
Commuting Diagrams



Hoare: *Proof of Correctness of Data Representations*, 1972

Testing store

```
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))
```

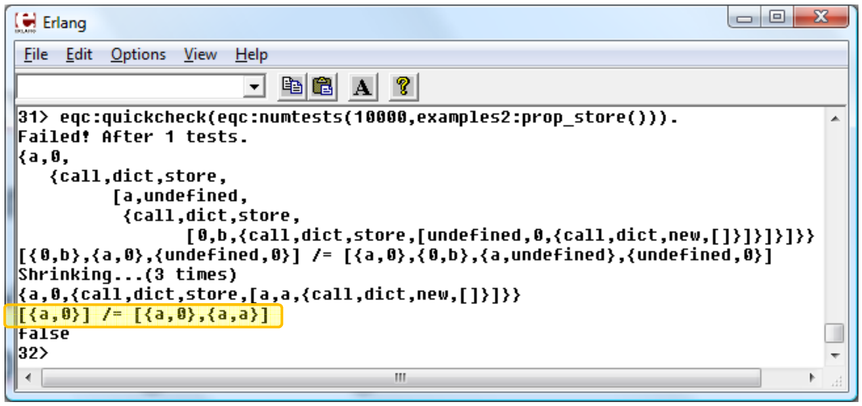


Next Steps…

- 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 *sequ*... ...ls

Generat... comr...

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

```
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).
```

Check they ran OK

Generating Commands

- We generate *symbolic calls* as before:

```
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

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

- Now Pids can be generated from the state

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

State Transitions

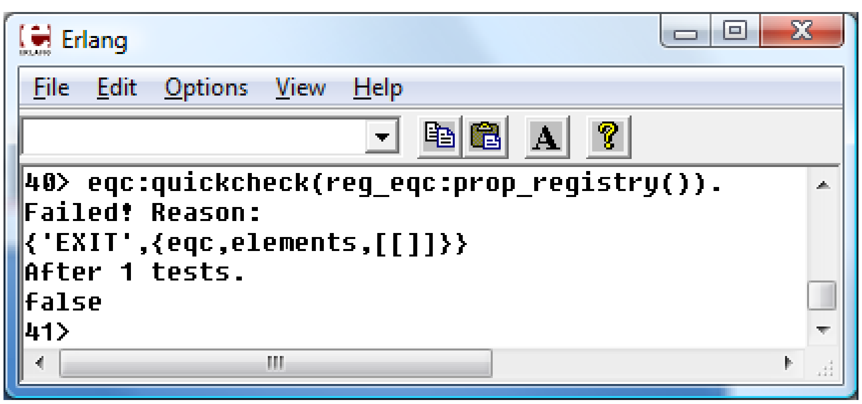
- Specify behaviour of the model

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

```
next_state(S,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(S,_V,{call,_,unregister,[Name]}) ->
  S#state{regs=proplists:delete(Name,S#state.regs)};
next_state(S,_V,{call,_,_,_}) ->
  S.
```

- Much like the model functions of the previous lecture

Let's Test It!



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

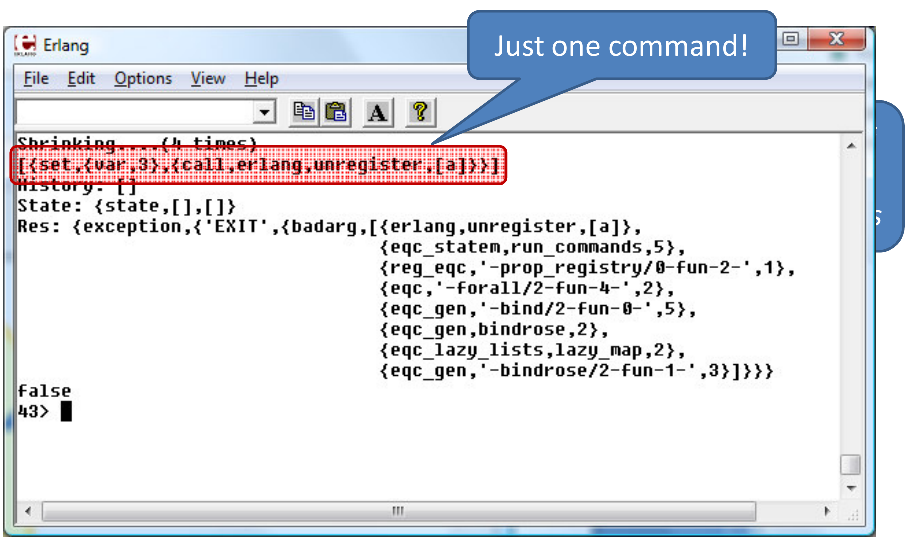
Conditional Generation

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

```
command(S) ->
  oneof([{call,erlang,register,[name(),pid(S)]}
          || S#state.pids/=[]] ++
        [{call,?MODULE,unregister,[name()]},
         {call,erlang,whereis,[name()]},
         {call,?MODULE,spawn,[]}
        ]).
```

  – Since [X || true] == [X], [X || false] == []

Let's Test It!



Preconditions

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

```
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

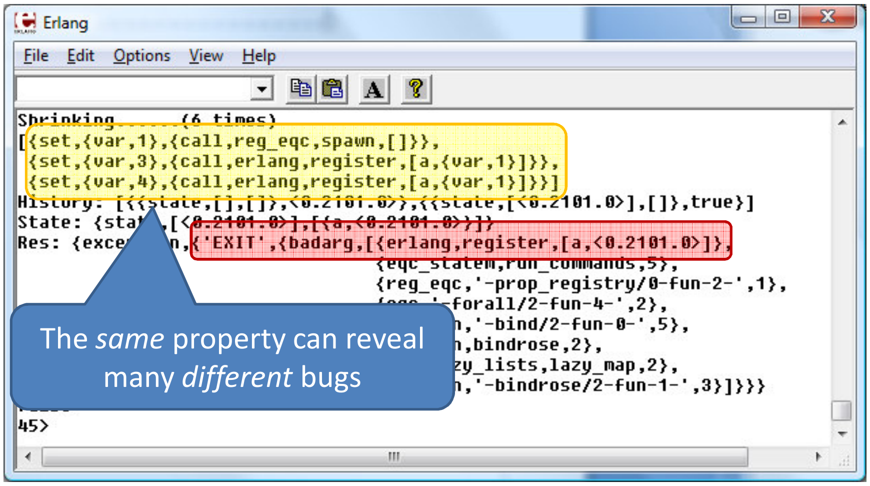
- Postconditions are checked after each API call, with access to the actual result

```
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.
```

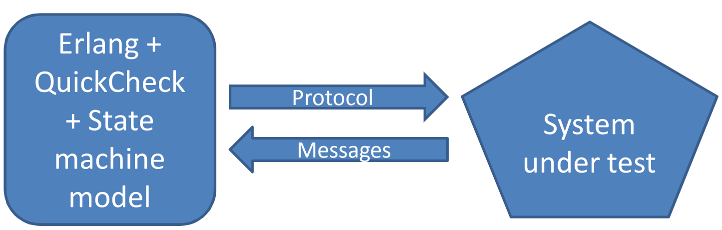
```
unregister(Name) -> catch erlang:unregister(Name).
```

```
command(S) ->
    oneof([…{call,?MODULE,unregister,[name()]},…]).
```

Let's Test It!

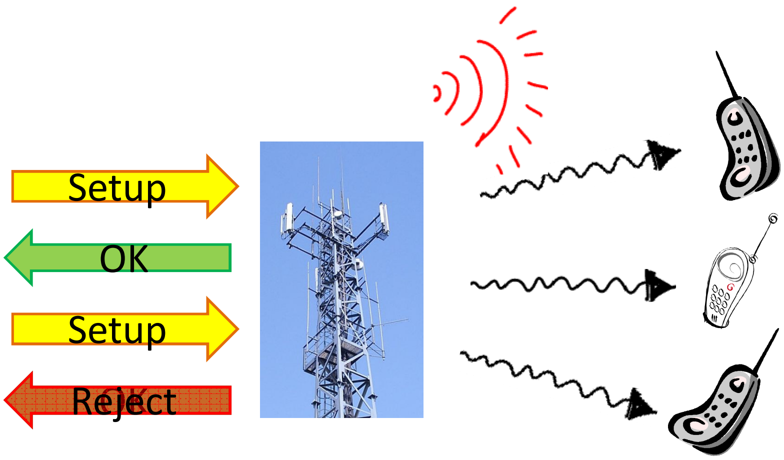


Typical Industrial Scenario



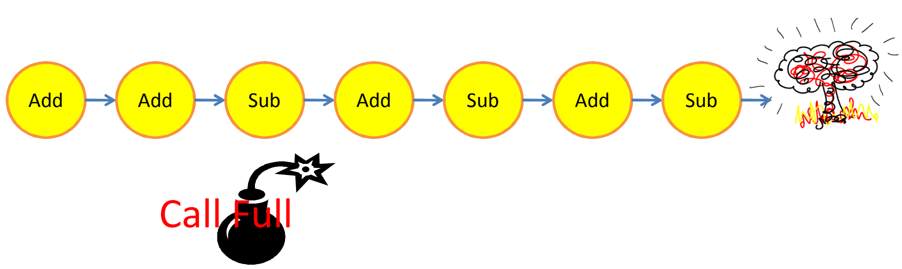
- QuickCheck finds a minimal sequence of messages that provokes a failure

3G Radio Base Station



Media Proxy

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



QuviQ

Exercises