



# 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* Generate commands that test calls

```
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:

```
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!



```
Erlang
File Edit Options View Help
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

```
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!



Shrinking... (4 times)

```
[{set, {var, 3}, {call, erlang, unregister, [a]}}]
```

History: []

State: {state, [], []}

Res: {exception, {EXIT, {badarg, [{erlang, unregister, [a]},  
 {eqc\_state, run\_commands, 5},  
 {reg\_eqc, '-prop\_registry/0-fun-2-', 1},  
 {eqc, '-forall/2-fun-4-', 2},  
 {eqc\_gen, '-bind/2-fun-0-', 5},  
 {eqc\_gen, bindrose, 2},  
 {eqc\_lazy\_lists, lazy\_map, 2},  
 {eqc\_gen, '-bindrose/2-fun-1-', 3}]}}}

false

43> █

## 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!

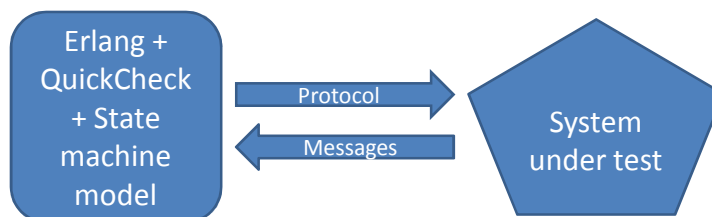


```

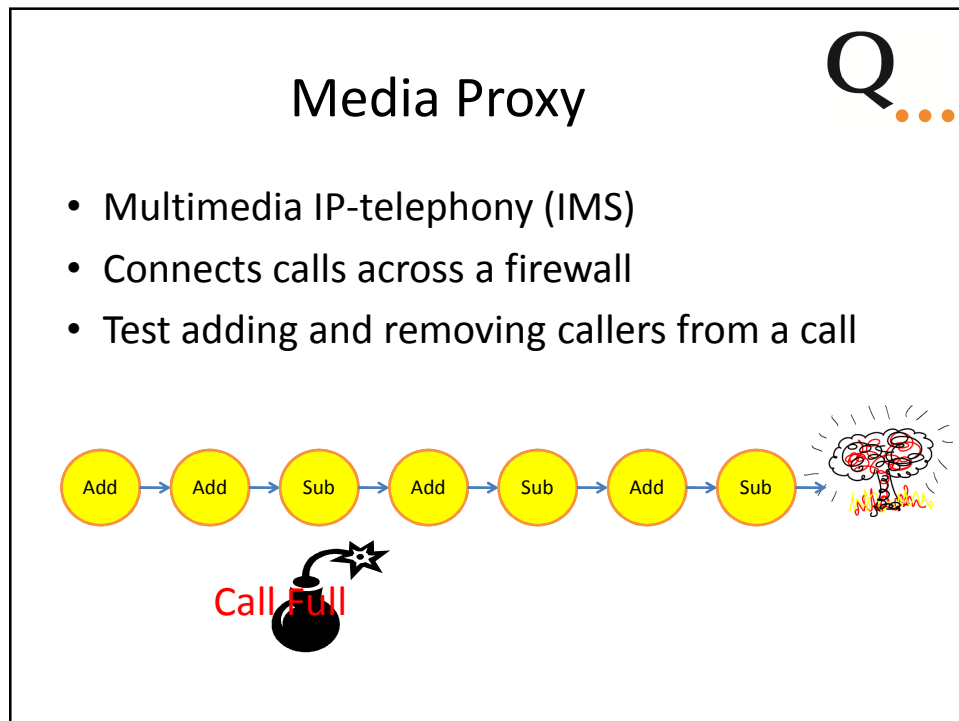
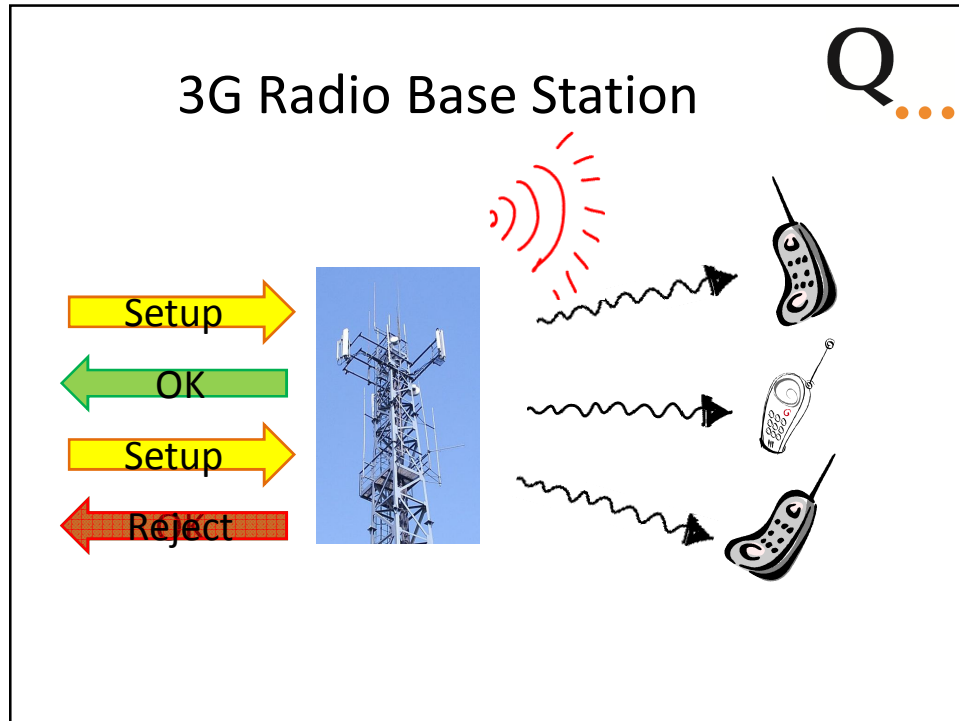
Erlang
File Edit Options View Help
Shrinking (6 times)
[{set,{var,1},{call,reg_eqc,spawn,[]}},
 {set,{var,3},{call,erlang,register,[a,{var,1}]}}],
 {set,{var,4},{call,erlang,register,[a,{var,1}]}}]
History: [{state,[],[],<0.2101.0>},{state,[<0.2101.0>,[],true]}]
State: {state,[<0.2101.0>],[a,<0.2101.0>]}
Res: {exception,EXIT,{badarg,[{erlang,register,[a,<0.2101.0>]}]}
      {reg_eqc_state,run_commands,5},
      {reg_eqc,'-prop_registry/0-fun-2-',1},
      {prop_forall/2-fun-4-',2},
      {prop_bind/2-fun-0-',5},
      {prop_bindrose,2},
      {prop_lazy_lists,lazy_map,2},
      {prop_bindrose/2-fun-1-',3}}]}
45>
  
```

The same property can reveal many different bugs

## Typical Industrial Scenario



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





# Exercises