

Erlang Solutions Ltd.

# Testing for the Unexpected

Ulf Wiger, CTO Erlang Solutions Ltd  
QCon, London 2011



© 1999-2011 Erlang Solutions Ltd.

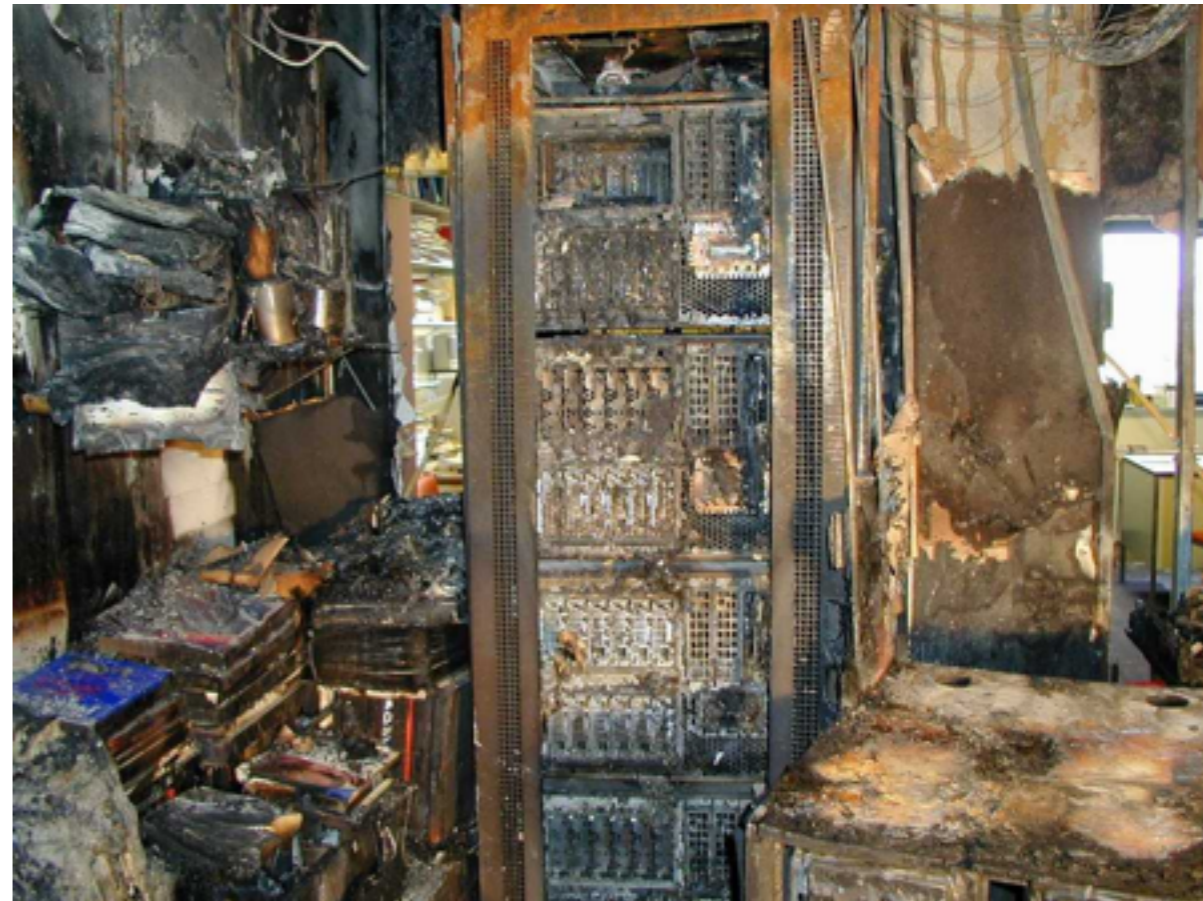
# About me

- Spent 4 years in Alaska working on Military Command & Control and Disaster Response
- 13 years at Ericsson building Phone and Multimedia switches
- Now CTO at Erlang Solutions Ltd



# When Things Break

- Non-stop systems must have a disaster plan
- Everything breaks — eventually! Plan for it
- This has deep impact on
  - ▬ architecture
  - ▬ testing
  - ▬ deployment



# How to certify quality?

- Testing — *lots* of testing...
- Example: Ericsson AXD 301 (launched 1998)
  - Ca 2,000 manual test cases before release
  - Introduced automation => ca 11,000 tests
  - Fault density lowered by 2.5x
- Test automation is mandatory!



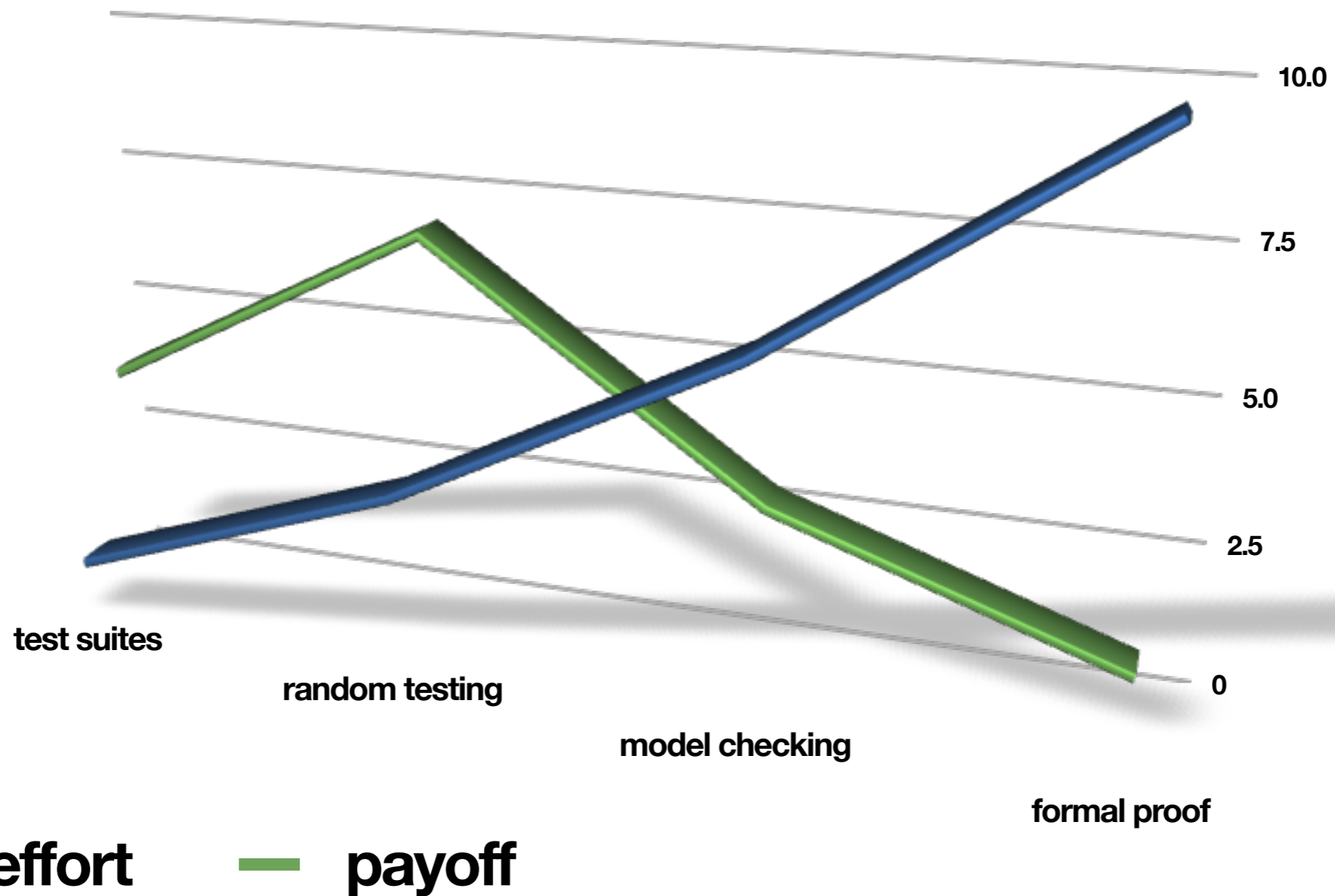
# This talk

- Standard test automation should be the low-water mark
- Need to evolve towards more powerful testing
- Find the right balance between effort and results
- The techniques will/should affect
  - How you describe your requirements
  - How you structure your code
  - How you organize your project



# Effort vs payoff

Warning: Post-normal science



# Complexity vs effort

More post-normal science



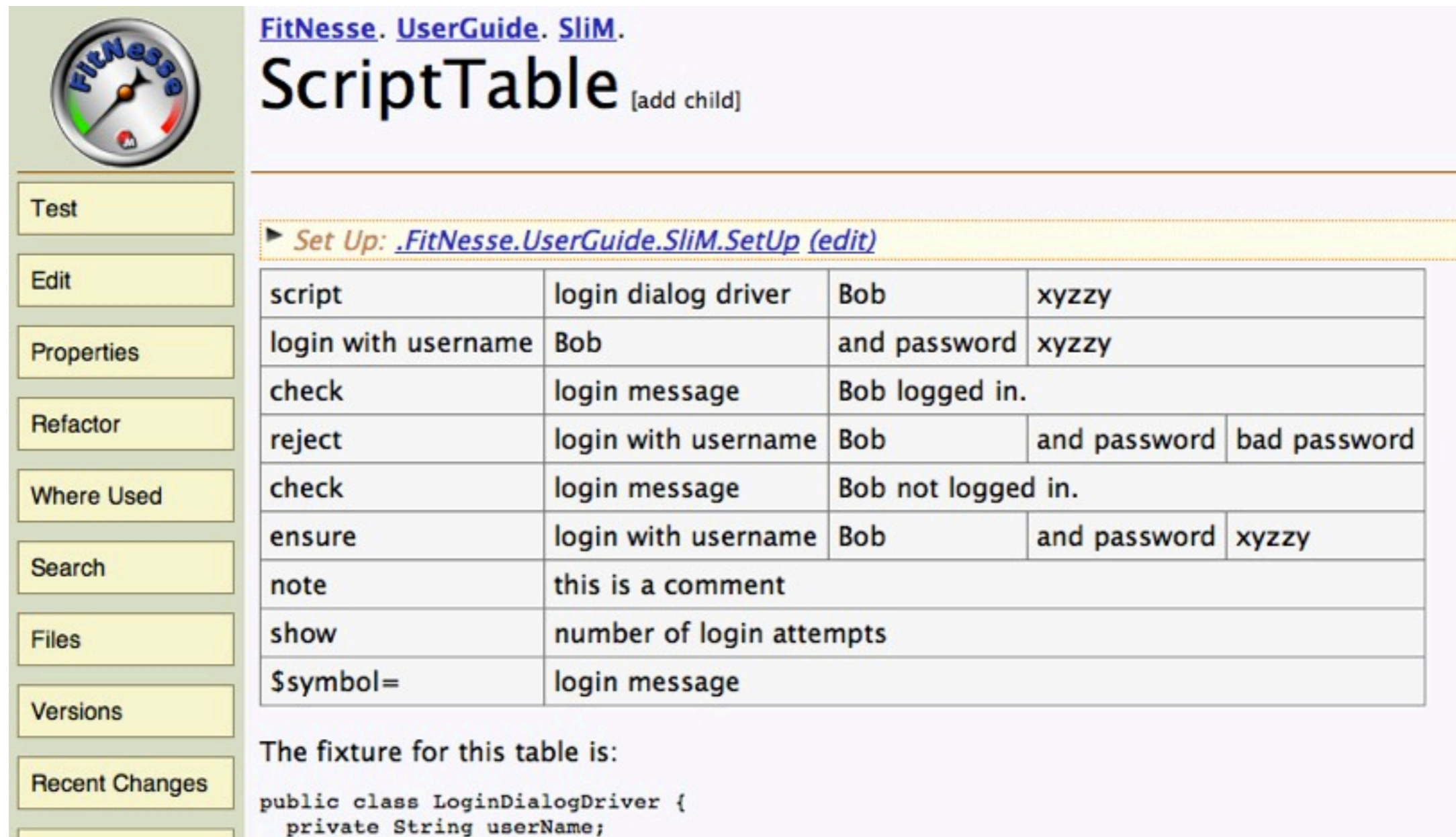
QTronic pilot: (<http://epubl.ltu.se/1402-1617/2010/070/LTU-EX-10070-SE.pdf>)

- 20% time savings on writing initial test cases
- 90% savings on modifying existing functionality



# Test automation examples

- **Fitnessse** — intuitive for non-programmers



The screenshot shows the FitNesse web interface. On the left is a sidebar with navigation buttons: Test, Edit, Properties, Refactor, Where Used, Search, Files, Versions, and Recent Changes. The main content area displays the title "FitNesse. UserGuide. SLiM. ScriptTable" with a link to "add child". Below the title is a "Set Up" section with a link to ".FitNesse.UserGuide.SLiM.SetUp (edit)". The main part of the page is a table representing a ScriptTable with the following rows:

script	login dialog driver	Bob	xyzyy	
login with username	Bob	and password	xyzyy	
check	login message	Bob logged in.		
reject	login with username	Bob	and password	bad password
check	login message	Bob not logged in.		
ensure	login with username	Bob	and password	xyzyy
note	this is a comment			
show	number of login attempts			
\$symbol=	login message			

Below the table, it states "The fixture for this table is:" followed by a code block:

```
public class LoginDialogDriver {  
    private String userName;
```



# Test automation examples

- Ruby — straightforward for unit test

```
require "simpleNumber"
require "test/unit"

class TestSimpleNumber < Test::Unit::TestCase

  def test_simple
    assert_equal(4, SimpleNumber.new(2).add(2) )
    assert_equal(4, SimpleNumber.new(2).multiply(2) )
  end

  def test_typecheck
    assert_raise( RuntimeError ) { SimpleNumber.new('a') }
  end

  def test_failure
    assert_equal(3, SimpleNumber.new(2).add(2), "Adding doesn't work" )
  end

end
```



# Test automation examples

- TTCN-3 — an ETSI/ITU standard test language

```
// Coffee Machine

type port IntegerInputPortType message { in integer }
type port CharstringOutputPortType message { out charstring }

type component CoffeeMachineComponentType {
  port IntegerInputPortType InputPort;
  port CharstringOutputPortType OutputPort;
}

function CoffeeMachineFunction() runs on CoffeeMachineComponentType
{
  const integer Price := 50;
  var integer Amount, Cents;
  Amount := 0;
  while (true) {
    InputPort.receive(integer:?) -> value Cents;
    Amount := Amount+Cents;
    while (Amount >= Price) {
      OutputPort.send(charstring:"coffee");
      Amount := Amount-Price;
    }
  }
}
```

Not a general-purpose language  
—must be extended with e.g. C



# Test automation examples

- Erlang
- Handy for complex concurrency tests

```
t_await() ->
  Me = self(),
  {_Pid,Ref} = spawn_monitor(
    fun() -> exit(?assert(gproc:await({n,l,t_await}) == {Me,val})) end),
  ?assert(gproc:reg({n,l,t_await},val) == true),
  receive
    {'DOWN', Ref, _, _, R} ->
      ?assertEqual(R, ok)
  after 10000 ->
    erlang:error(timeout)
  end.
```



# How to know what to test?

- Read the requirements spec, try to imagine the important cases
- Standard conformance tests, perhaps?
- Measure code coverage, invent tests until coverage is high...
- TDD: Stories => test cases => code
  
- Some obvious problems...



# Problems

- Have to trust the requirements spec
  - which is usually not very formally written
  - ...usually not even *correct*
- Easy to say what the system is *supposed* to do
- Harder to describe what you don't expect
- Code coverage is an unreliable metric
  - Low coverage means you likely have a problem
  - High coverage doesn't necessarily = quality



# Silly coverage example

- Simple Erlang example with test suite
- 100% code coverage, yet obviously incorrect



```
-module(mymath).
-export([factorial/1]).
-include_lib("eunit/include/eunit.hrl").

factorial(0) ->
    1;
factorial(N) ->
    N * factorial(N-1).

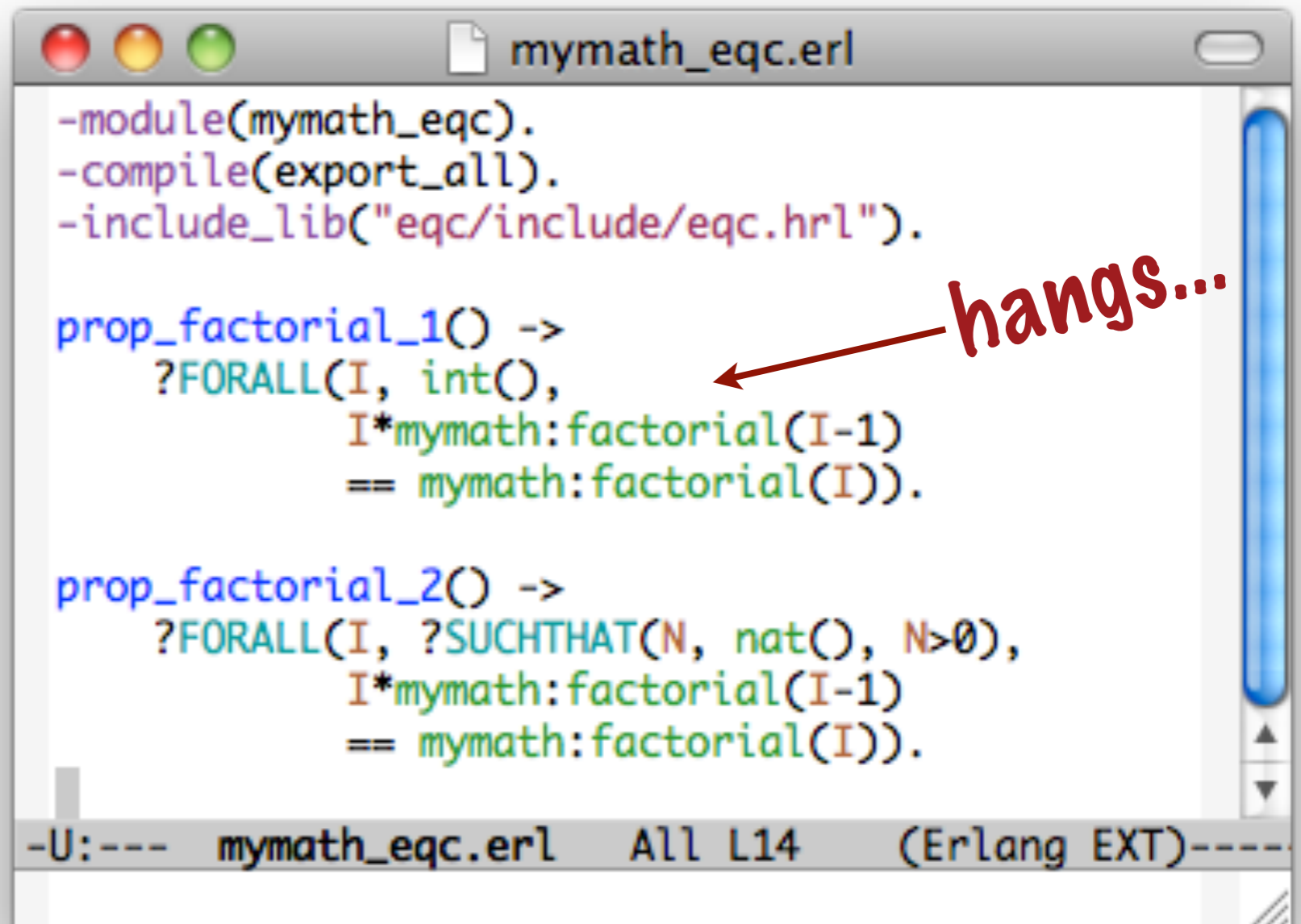
factorial_test() ->
    factorial(3) == 6.
```

---:--- mymath.erl Top L12 (Erlang EXT)



# Fixing our test suite

- Add another “pin-prick” test...
- Higher abstraction, e.g. with random testing.
- Example:  
QuickCheck



```
-module(mymath_eqc).
-compile(export_all).
-include_lib("eqc/include/eqc.hrl").

prop_factorial_1() ->
  ?FORALL(I, int(),
    I*mymath:factorial(I-1)
    == mymath:factorial(I)).

prop_factorial_2() ->
  ?FORALL(I, ?SUCHTHAT(N, nat(), N>0),
    I*mymath:factorial(I-1)
    == mymath:factorial(I)).
```

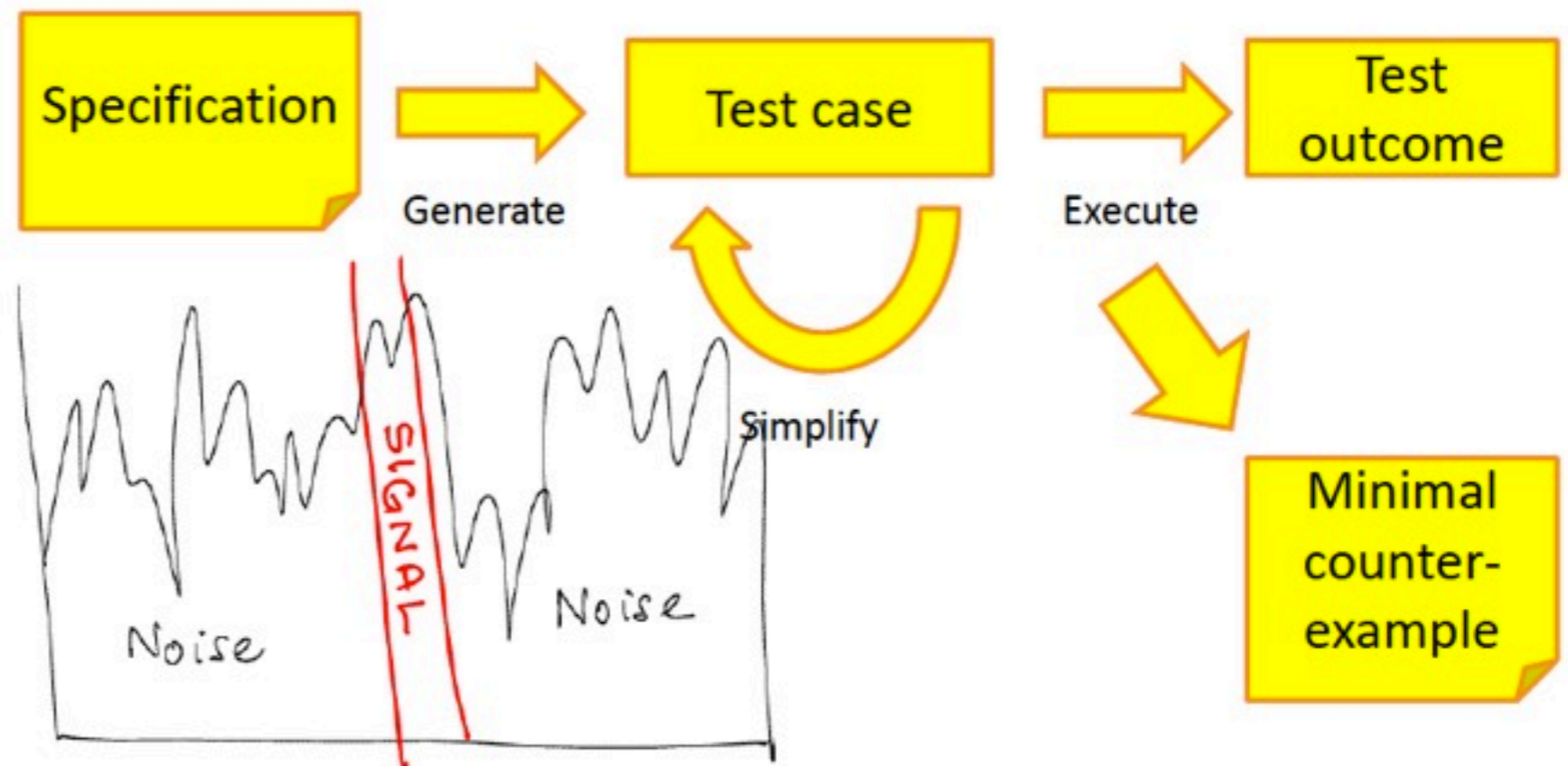
*hangs...*

-U:--- mymath\_eqc.erl All L14 (Erlang EXT)---



# QuickCheck: Random testing

- Highly imaginative generation of *legal* input data
  - ▀ We define what types of input are legal
- Tool generates test cases on the fly
- *Controlled* randomness is the key
- Shrinking



© 2011 Erlang Solutions Ltd.

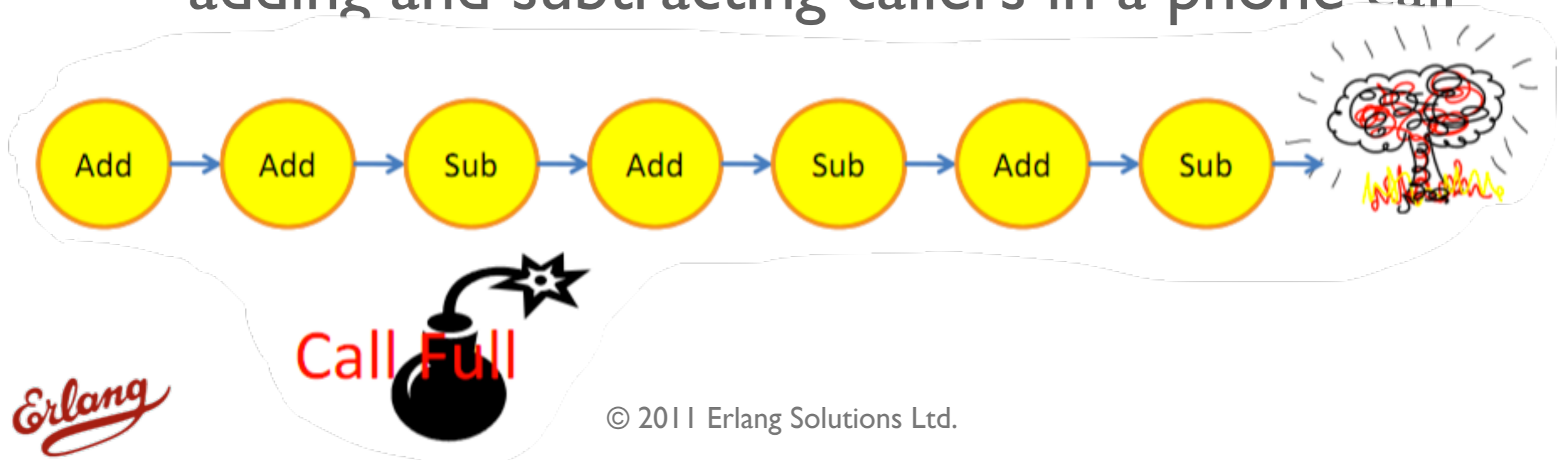
# QuickCheck pilot: Media Proxy

- Interoperability test—endless complexity
- Normal approach: connect two products, see them fail, adjust, repeat
- Our contract spec: 100 pages long
- Our interop test spec: 700 pages long...
  
- Problem: Requires stateful testing, which QuickCheck couldn't do



# QuickCheck pilot: Media Proxy

- 6 work days on a complex, “well-tested” product
- Found some really interesting bugs
- Our 100-page spec + our 700-page test spec  
~> 500 LOC Erlang (ok, slight exaggeration...)
- Favourite bug of all time
  - adding and subtracting callers in a phone call



© 2011 Erlang Solutions Ltd.

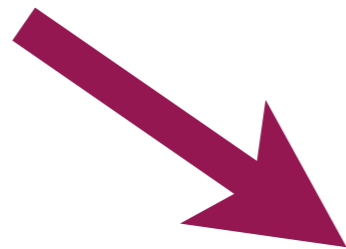
# Pilot—How was it done?

- Generators

MediaDescriptor ::= SEQUENCE

```
{ termStateDescr TerminationStateDescriptor OPTIONAL,  
  streams CHOICE  
  { oneStream StreamParms,  
    multiStream SEQUENCE OF StreamDescriptor  
  } OPTIONAL,  
  ...
```

...



```
mediadescriptor(Streams) when Streams /= [] ->  
  {mediaDescriptor,  
   #'MediaDescriptor'{  
     streams =  
       case Streams of  
         [{Id,Mode}] ->  
           oneof([oneStream,streamParms(Mode)],  
                {multiStream,[stream(Id,Mode)]});  
         _ -> {multiStream,  
              [stream(I,M) || {I,M} <- Streams]}  
       end}}.
```



# Pilot—Generate only valid input

- Example: The “add” command is valid if
  - The context (call) is empty
  - The context contains only a single termination

```
valid_cmd(S,{set,V,_,send_add,[Cxt,Streams,Req]}) ->  
lists:member(Cxt,[?megaco_choose_context_id |  
singletoncontexts(S)])
```

- Hmm, we could also generate *invalid* input...
  - Control the proportions of valid vs invalid



# Pilot—Post-conditions

- Given our (logical) state, and the command issued
  - verify the result received
- If we sent an invalid command, the result should be a rejection

# Pilot—Random evil...

- Last-minute experiment with controlled generation of illegal values
- Must think about what is a *meaningful* illegal value
- No time to follow up, but...
  - Initial results were astonishing
  - Within minutes, QuickCheck had broken our error handling in a number of ways
  - Would likely ruin any ad-hoc error handling approach



# Model-based Testing

- QuickCheck spin-offs exist in several other languages (Haskell, Scala, Java, Clojure, ...)
- Other tools exist, e.g. C#-based NModel (<http://nmodel.codeplex.com>, .NET only..)

```
using NModel;
using NModel.Attributes;
using NModel.Execution;

namespace PowerSwitch
{
    enum Power { On, Off };
    public static class Contract
    {
        static Power power = Power.Off;

        [Action]
        static void PowerOn() { power = Power.On; }
        static bool PowerOnEnabled() { return power == Power.Off; }

        [Action]
        static void PowerOff() { power = Power.Off; }
        static bool PowerOffEnabled() { return power == Power.On; }

        public static ModelProgram Create() { return LibraryModelProgram.Create(typeof(Contract)); }
    }
}
```

```
FSM(Off, AcceptingStates(),
    Transitions(t(Off, PowerOn(), On), t(On, PowerOff(), Off)))
```



# QuickCheck in Anger

- Gemini Mobile
- Combining
  - <http://github.com/norton/ubf>
    - a protocol contract checker
  - <http://github.com/meck>
    - a mocking library for Erlang
  - QuickCheck



# Modeling data races

- Problem: “Webmail for millions”
- Mail server compacting the inbox in the background
- During compaction, messages may appear twice in the database
- Filtering must verify that users are not getting duplicates



# Modeling data races

- Logs indicated that duplicates did happen
  - Spent one week meditating over log data
- Stubbed out the database with Meck
- Simulated database representations of valid message histories with QuickCheck
- Verified that message retrieval worked as expected (total effort: 1 day)
- *“In summary, a total of 4-5 old defects and 2 new defects introduced by the fixes were found by quickcheck.”* (Joseph Wayne Norton, Gemini Mobile)



# Random RESTfulness

- Used UBF to write a contract for a REST service (C++-based server)
- UBF JSON encoder
- Erlang UBF client, driven by QuickCheck
- QuickCheck generating random requests which comply with the UBF spec
- UBF contract checker verifies that the reply complies with the contract



# Lessons learned

- Side-effect free code easy to test
- Separate effects from data manipulation
- First prototype should yield an abstract test spec as input to the development project
- Random tests are cheap
  - don't fall in love with the tests

