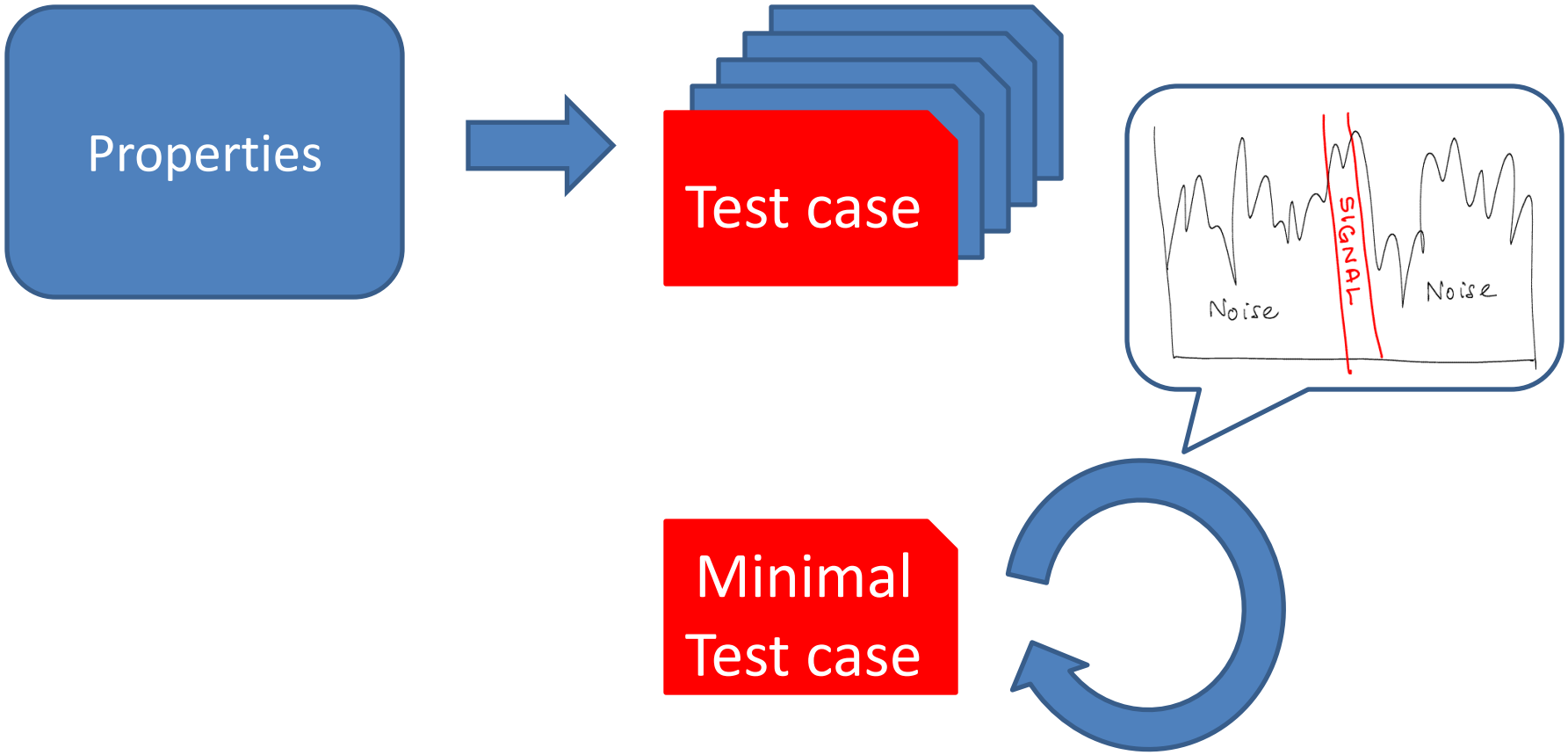
Property-based testing, race conditions, and QuickCheck

John Hughes

CHALMERS

QuviQ

QuickCheck in a Nutshell



Benefits

- Less time spent writing test code
  - One property replaces many tests
- Better testing
  - Lots of combinations you'd never test by hand
- Less time spent on diagnosis
  - Failures minimized automagically

Tests for Base 64 encoding

Expected results

Test cases

```
base64_encode(Config) when is_
    %% Two pads
    <<"QWxhZGRpbjpvcGVuIHNlc2FtZQ==">> =
        base64:encode("Aladdin:open sesame"),

    %% One pad
    <<"SGVsbG8gV29ybGQ=">> = base64:encode(<<"Hello World">>),

    %% No pad
    "QWxhZGRpbjpvcGVuIHNlc2Ft" =
        base64:encode_to_string("Aladdin:open sesam"),

    "MDEyMzQ1Njc4OSFAIzBeJiooKTs6PD4sLiBbXXt9" =
        base64:encode_to_string(
            <<"0123456789!@#0^&*();:<>,. []{}">>),
    ok.
```

Writing a Property

```
prop_base64() ->
  ?FORALL(Data,list(choose(0,255)),
          base64:encode(Data)==???).
```

Back to the tests…

```
base64_encode(Config) when is_list(Config) ->
    %% Two pads
    <<"QWxhZGRpbjpvcGVuIHNlc2FtZQ==">> =
        base64:encode("Aladdin:open sesame"),

    %% One pad
    <<"SGVsbG8gV29ybGQ=">> = base64:encode(<<"Hello World">>),

    %% No pad
    "QWxhZGRpbjpvcGVuIHNlc2Ft" =
        base64:encode_to_string("Aladdin:open sesam"),

    "MDEyMzQ1Njc4OSFAIzBeJiooKTs6PD4sLiBbXXt9" =
        base64:encode_to_string(
            <<"0123456789!@#0^&*();:<>,. []{}">>),
    ok.
```

Where did these come from?

Possibilities

- Someone converted the data
- Another base64 encoder
- The same base64 encoder!

Use the other encoder as an oracle

Use an old version (or a simpler version) as an oracle

Round-trip Properties

```
prop_encode_decode() ->
  ?FORALL(L,list(choose(0,255)),
    base64:decode(base64:encode(L))
      == list_to_binary(L)).
```

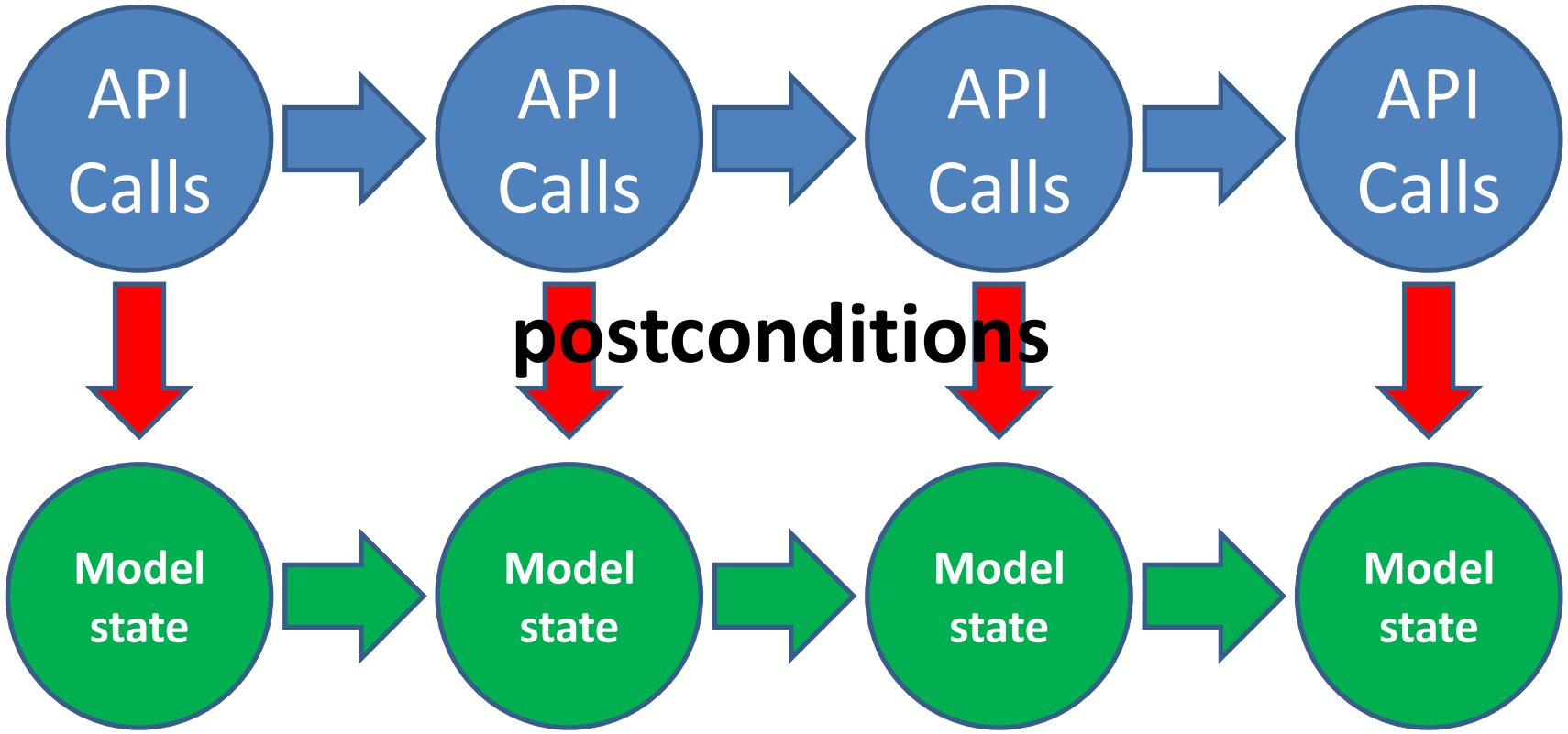
What does this test?

- **NOT** a complete test—will not find a consistent misunderstanding of base64
- **WILL** find mistakes in encoder or decoder

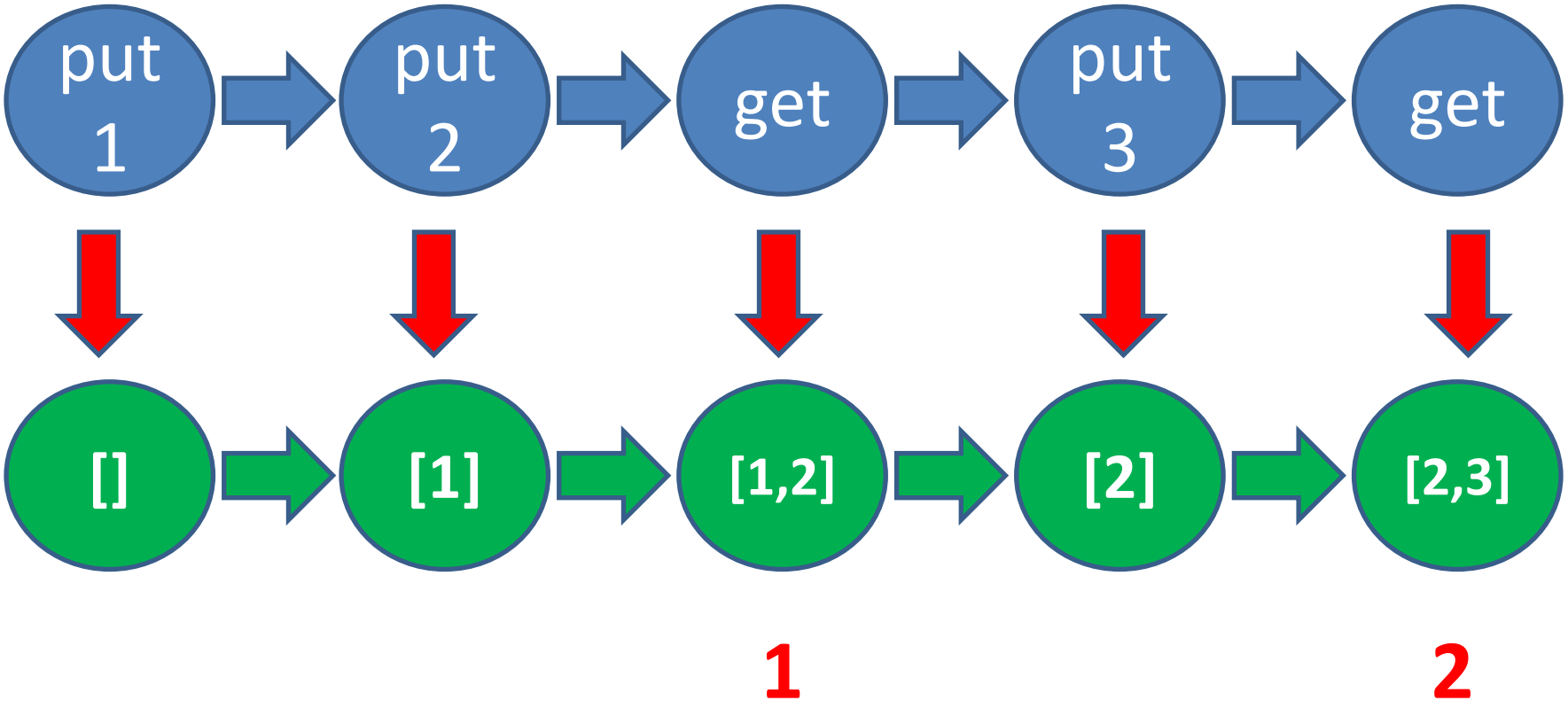
Simple properties find a lot of bugs!

Let’s test some C!

Modelling in Erlang



Example



Code Fragments

```
next_state_data(_,_,S,_,{call,_,put,[_,X]}) ->
   S#state{elements=S#state.elements++[X]};

next_state_data(_,_,S,_,{call,_,get,_}) ->
   S#state{elements=tl(S#state.elements)};


postcondition(_,_,S,{call,_,get,_},Res) ->
   Res == hd(S#state.elements);

postcondition(_,_,S,{call,_,size,_},Res) ->
   Res == length(S#state.elements);
```

A QuickCheck Property

```
prop_q() ->
    ?FORALL(Cmds,commands(?MODULE),
            begin
              {H,S,Res} = run_commands(?MODULE,Cmds),
              Res == ok)
            end).
```

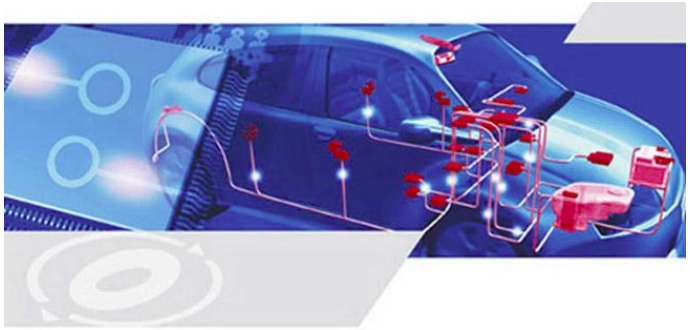
Let’s run some tests…

Lessons

- One property can find *many* bugs
- Shrinking makes diagnosis *very* simple

Doing it for real

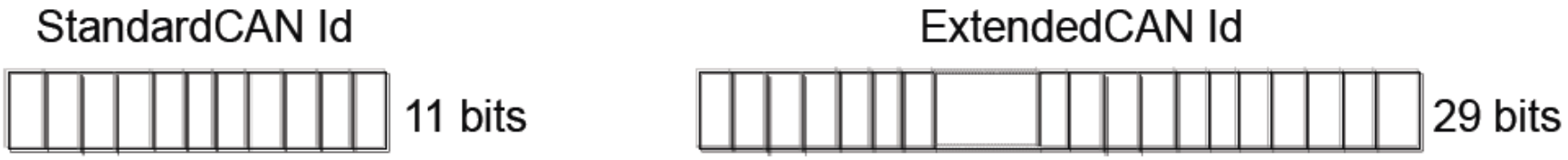
- AUTOSAR
  - Embedded software in cars
  - ~50 standard components



- Test code from three suppliers against QuickCheck models
- **95 issues found already!**

COM component: 500 pages of documents, 250 pages of C code… 25 pages of QuickCheck specification

Example: Mixed features

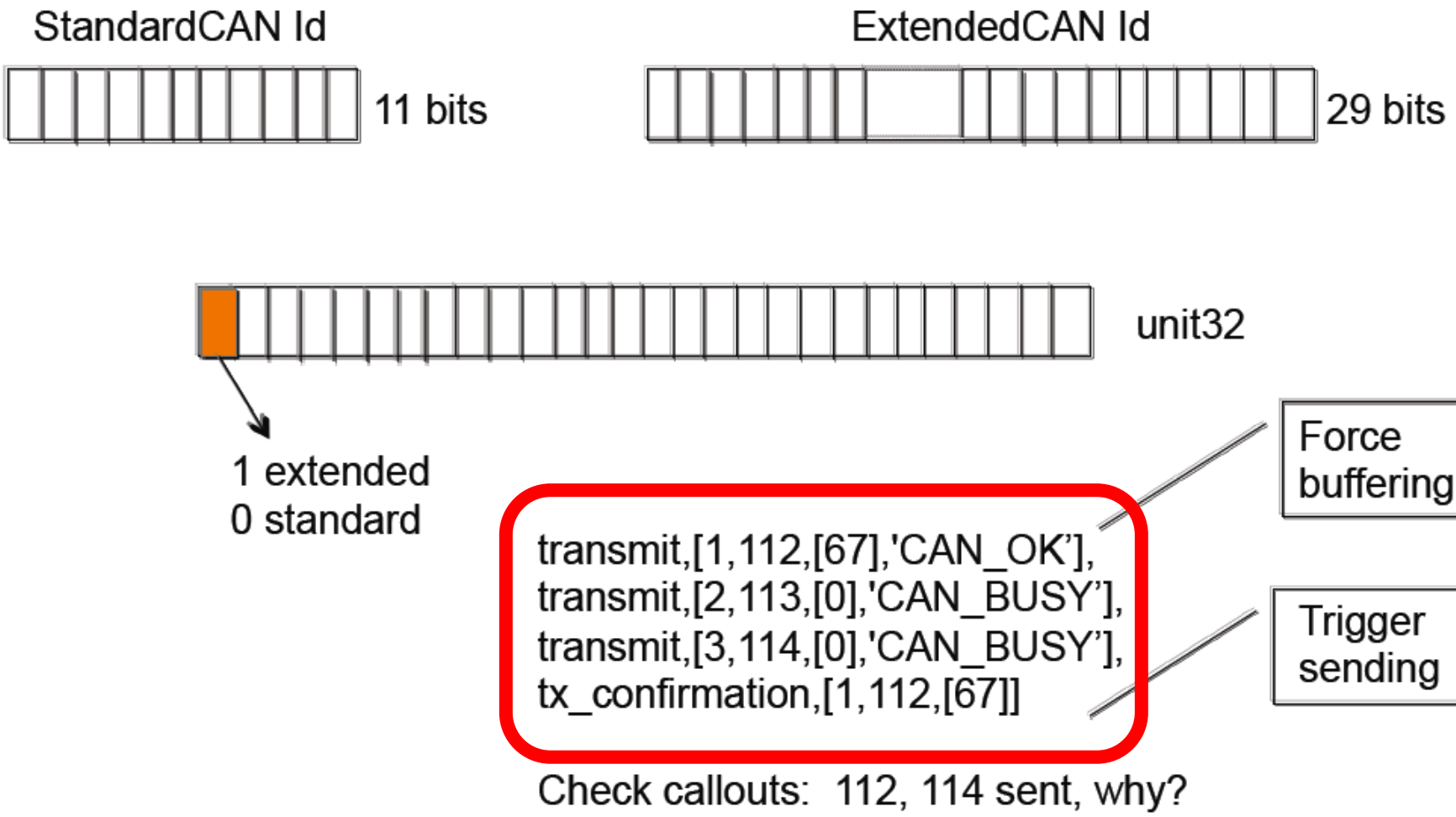


Priority: lowest number has highest priority

Example:
Extended Id 113 has higher
priority than standard Id 114

Buffered higher priority
messages should be sent first

Example: Mixed features



"We know there is a lurking bug somewhere in the dets code. We have got 'bad object' and 'premature eof' every other month the last year. We have not been able to track the bug down since the dets files is repaired automatically next time it is opened.“

Tobbe Törnqvist, Klarna, 2007

What is it?

500+ people in 5 years

Invoicing services for web shops

Distributed database: transactions, distribution, replication

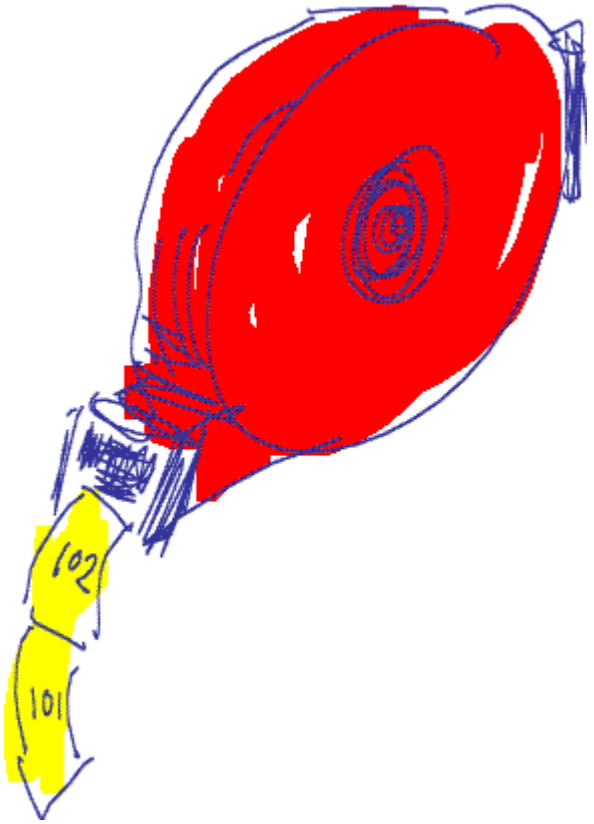
Tuple storage

Race conditions?

Imagine Testing This…

dispenser:take_ticket()

dispenser:reset()



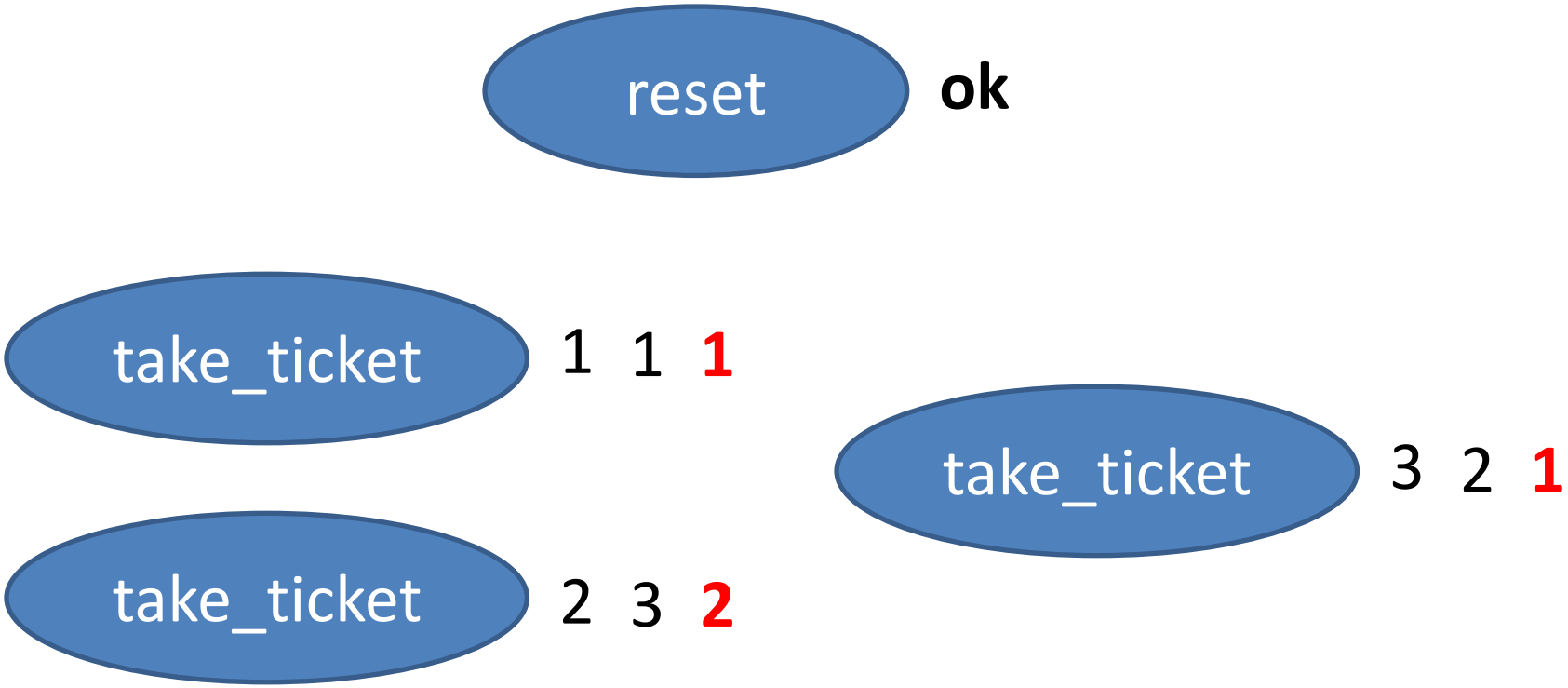
A Unit Test in Erlang

```
test_dispenser() ->
      ok =  reset(),
      1  =  take_ticket(),
      2  =  take_ticket(),
      3  =  take_ticket(),
      ok =  reset(),
      1  =  take_ticket().
```

Expected results

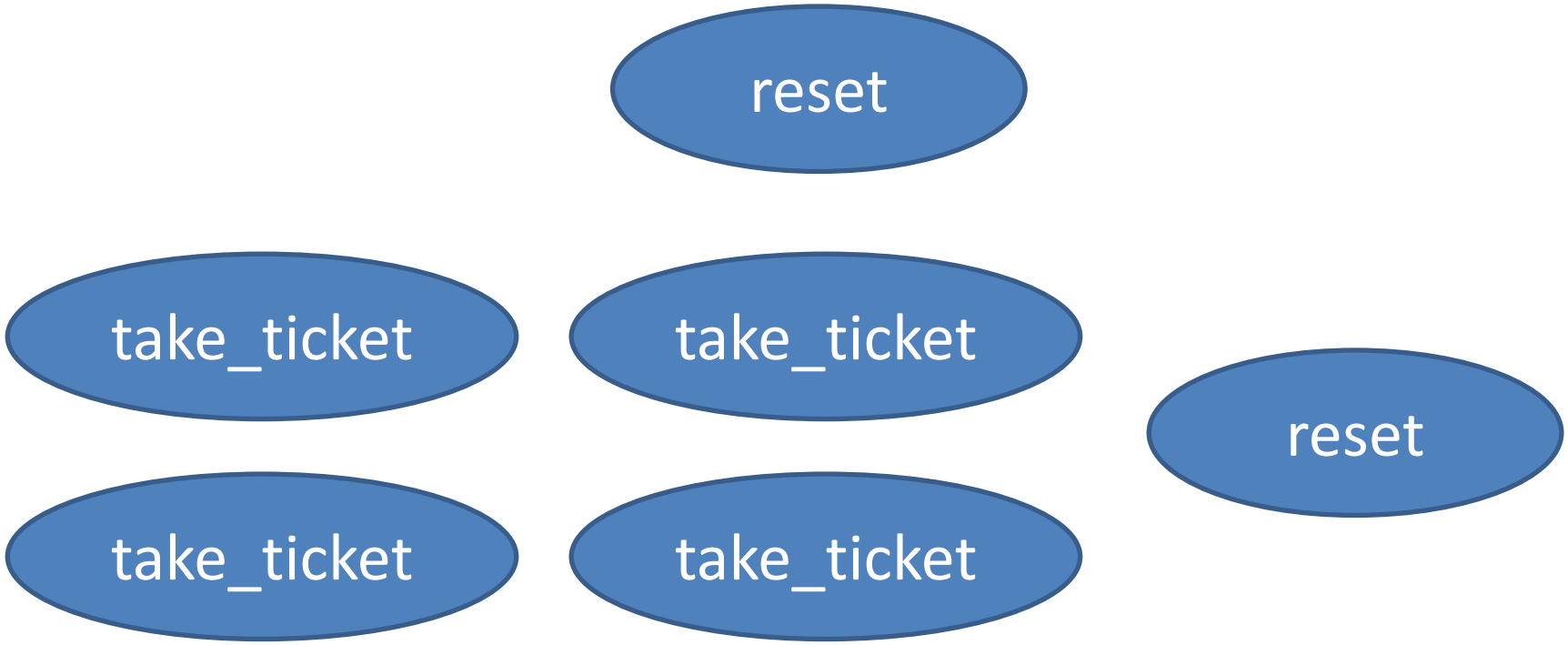
BUT…

A Parallel Unit Test



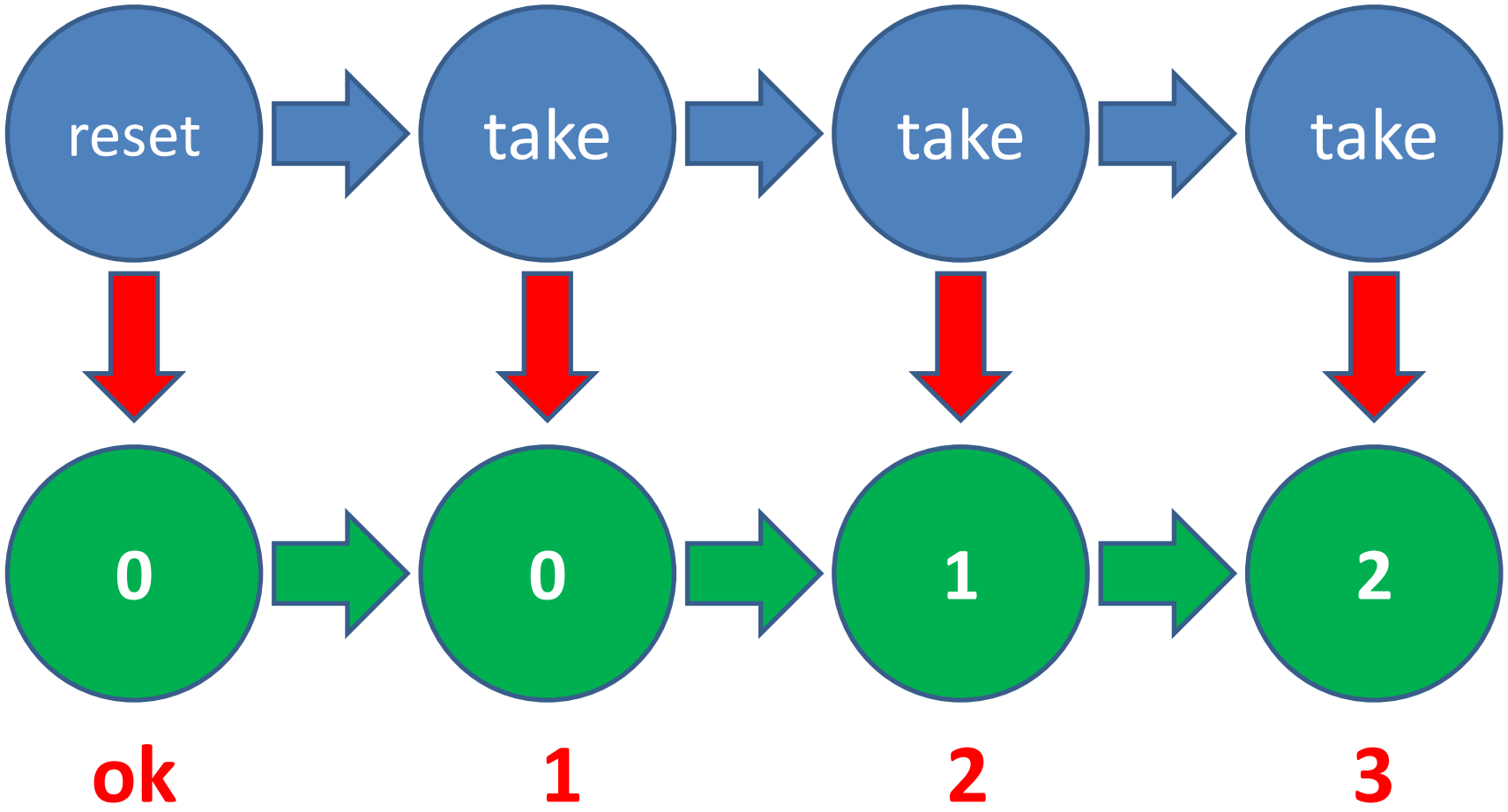
- Three possible correct outcomes!

Another Parallel Test

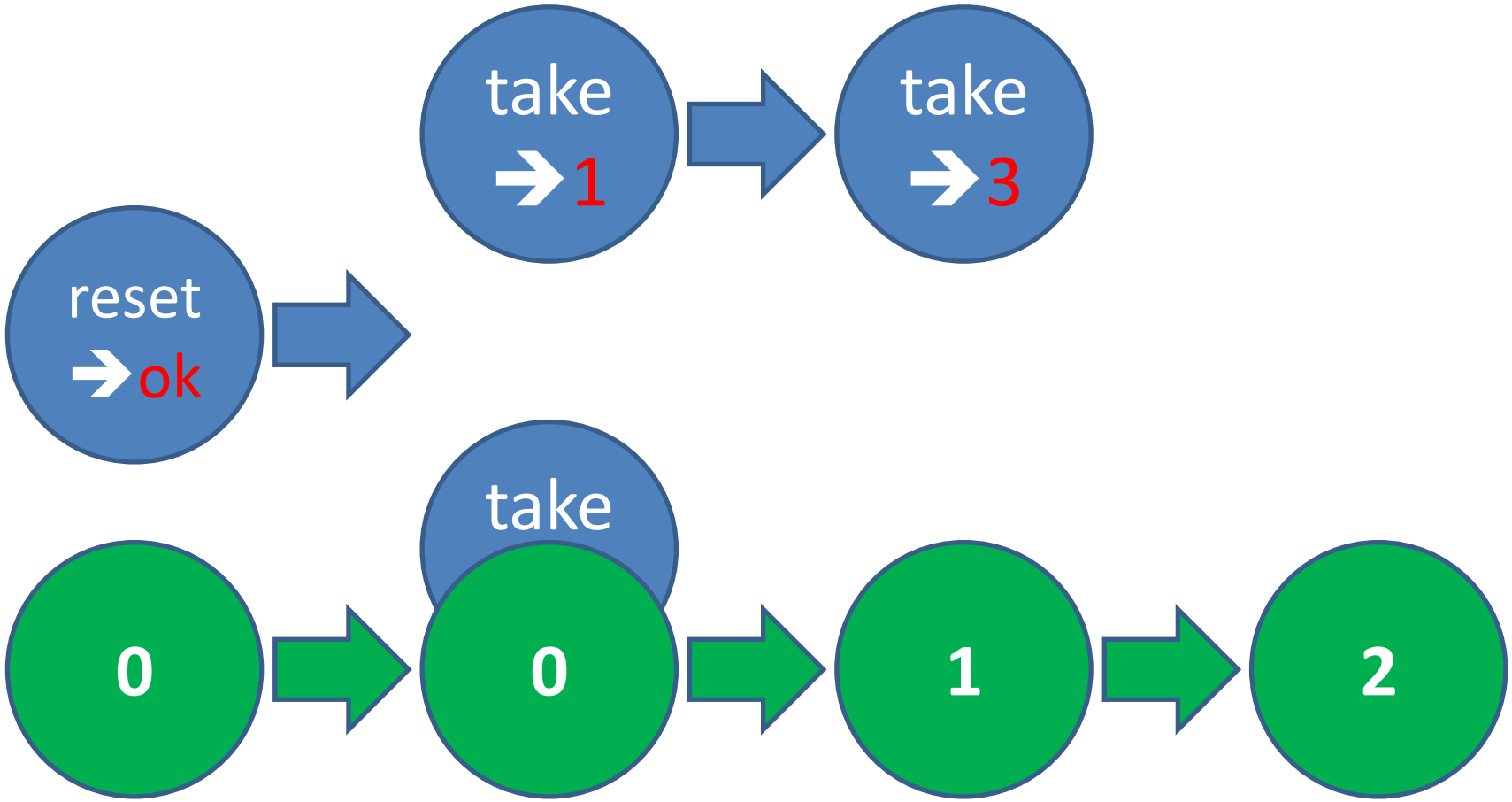


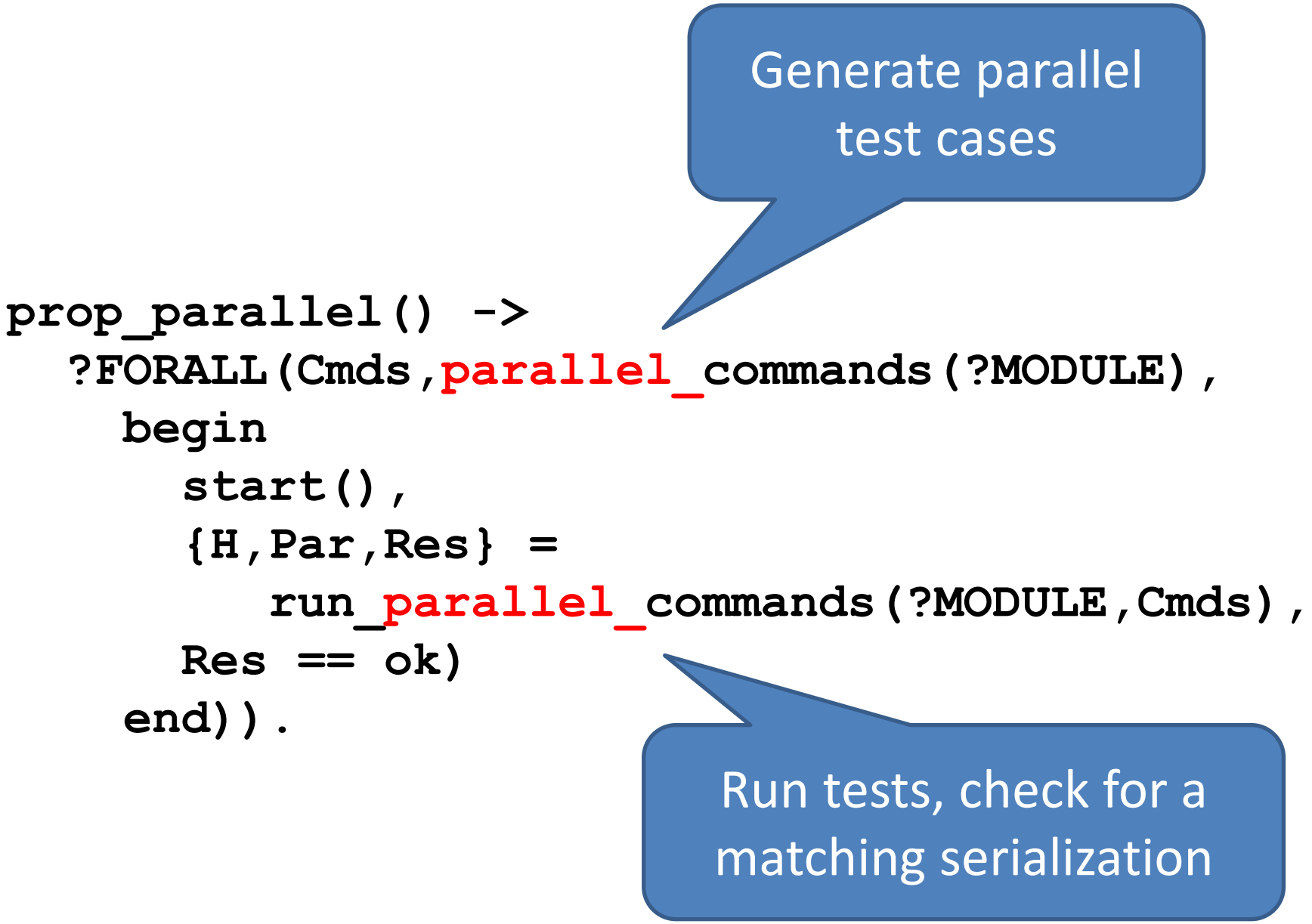
- 42 possible correct outcomes!

Modelling the dispenser



Parallel Test Cases




Generate parallel test cases
prop_parallel() ->
?FORALL(Cmds,parallel_commands(?MODULE),
begin
start(),
{H,Par,Res} =
run_parallel_commands(?MODULE,Cmds),
Res == ok)
end)).
Run tests, check for a matching serialization

Let’s run some tests

```
take_ticket() ->
    N = read(),
    write(N+1),
    N+1.
```

Prefix:

Parallel:

1. take_ticket() --> 1

2. take_ticket() --> 1

Result: no_possible_interleaving

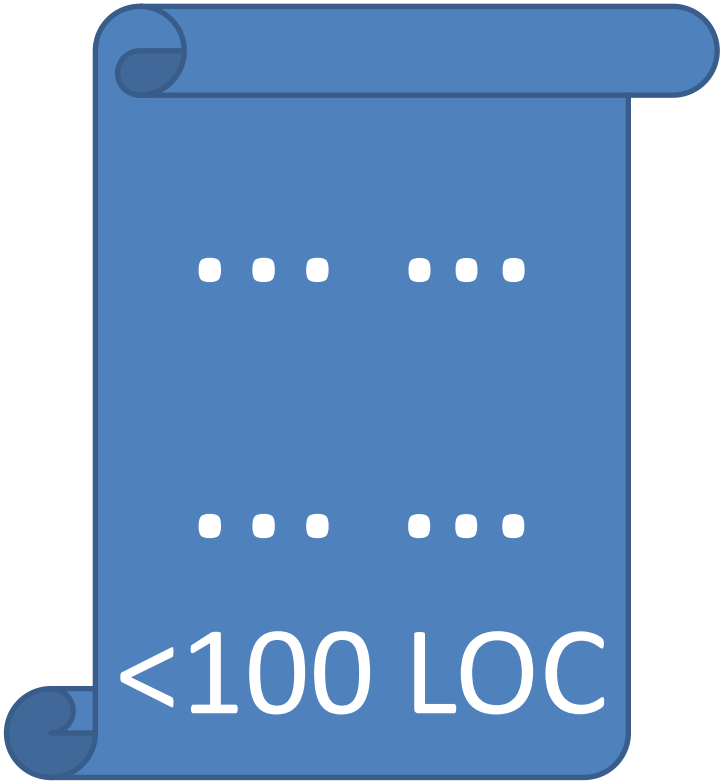
dets

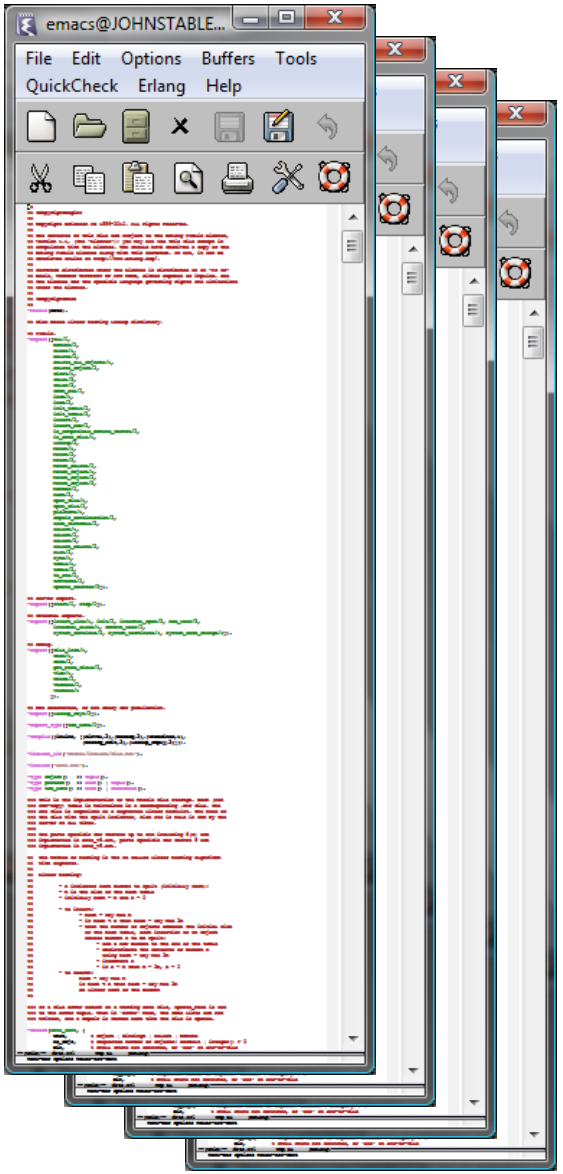
- Tuple store:

  {Key, Value1, Value2…}

- Operations:
  - insert(Table,ListOfTuples)
  - delete(Table,Key)
  - insert_new(Table,ListOfTuples)
  - …
- Model:
  - List of tuples (almost)

QuickCheck Specification





> 6,000 LOC

DEMO

Bug #1

insert_new(Name, Objects) -> Bool

Types:
Name = name()
Objects = object() | [object()]
Bool = bool()

```
Prefix:
    open_file(dets

Parallel:
1. insert(dets_ta

2. insert_new(dets_table,[]) --> ok

Result: no_possible_interleaving
```

Bug #2

```
Prefix:
   open_file(dets_table,[{type,set}]) --> dets_table

Parallel:
1. insert(dets_table,{0,0}) --> ok

2. insert_new(dets_table,{0,0}) --> …time out…
```

=ERROR REPORT==== 4-Oct-2010::17:08:21 ===
** dets: Bug was found when accessing table dets_table

Bug #3

```
Prefix:
   open_file(dets_table,[{type,set}]) --> dets_table

Parallel:
1. open_file(dets_table,[{type,set}]) --> dets_table

2. insert(dets_table,{0,0}) --> ok
   get_contents(dets_table) --> []

Result: no_possible_interleaving
```

Is the file corrupt?

Bug #4

```
Prefix:
   open_file(dets_table,[{type,bag}]) --> dets_table
   close(dets_table) --> ok
   open_file(dets_table,[{type,bag}]) --> dets_table

Parallel:
1. lookup(dets_table,0) --> []

2. insert(dets_table,{0,0}) --> ok

3. insert(dets_table,{0,0}) --> ok

Result: ok
```

Bug #5

```
Prefix:
   open_file(dets_table,[{type,set}]) --> dets_table
   insert(dets_table,[{1,0}]) --> ok

Parallel:
1. lookup(dets_table,0) --> []
   delete(dets_table,1) --> ok

2. open_file(dets_table,[{type,set}]) --> dets_table

Result: ok
false
```

"We know there is a lurking bug somewhere in the dets code. We have got 'bad object' and 'premature eof' every other month the last year."

Tobbe Törnqvist, Klarna, 2007

Each bug fixed the day after reporting the failing case

Before

- Files over 1GB?
- Rehashing?
- > 6 weeks of effort!

After

- Database with *one* record!
- 5 calls to reproduce
- < 1 day to fix

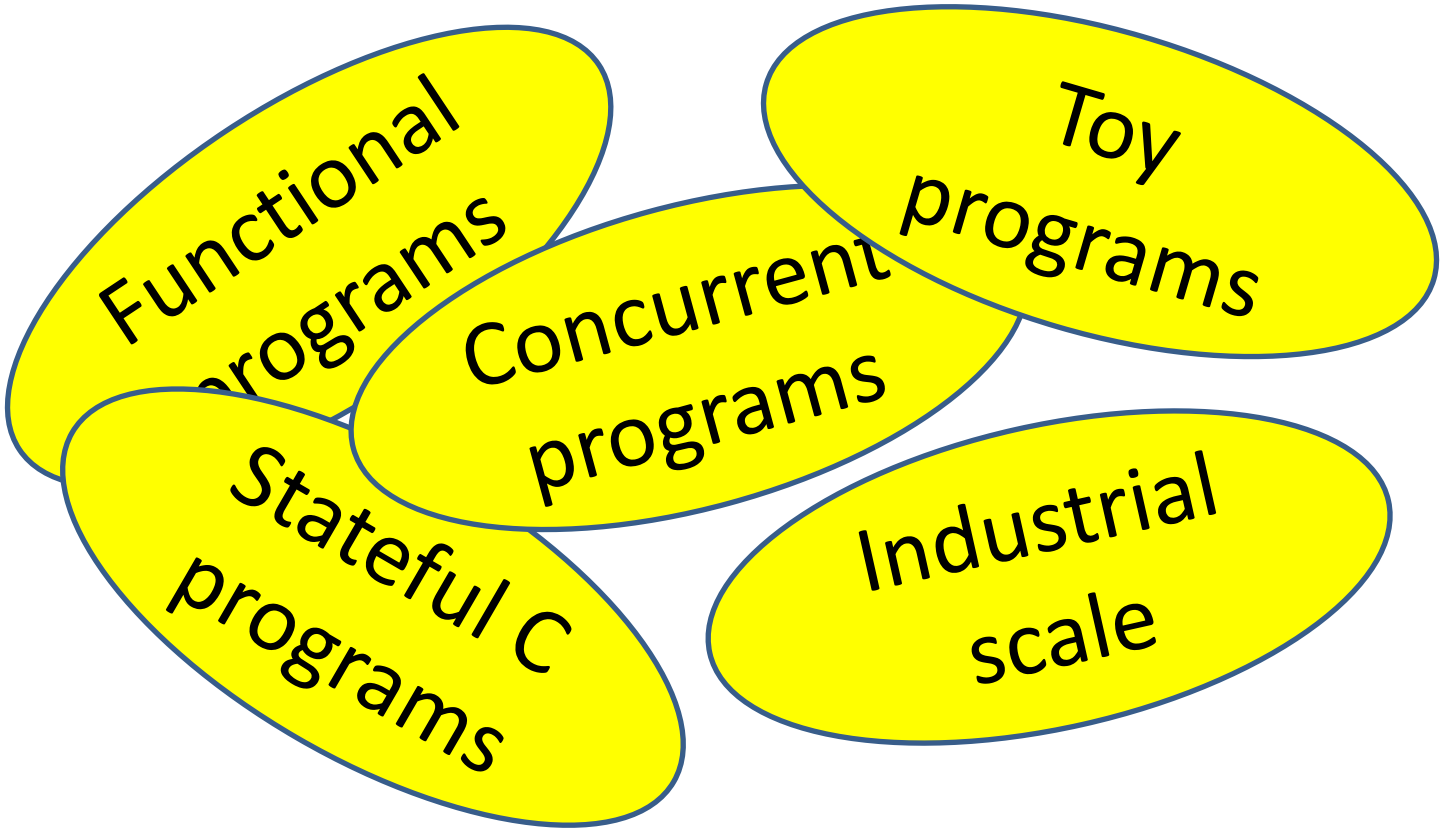
Hand-written test suites test *one* feature at a time

Generated tests can test *many* features, in unexpected combinations

- Particularly good for finding feature interactions—such as race conditions

- 100% code coverage is only the beginning…

Property-based testing



- Finds bugs in everything it's applied to!

Better Testing—Less Work