Testing atomicity

Finding race conditions by random testing

John Hughes

CHALMERS QuviQ

"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

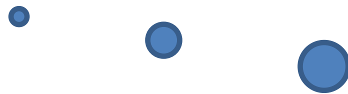
What is it?

700+
people in
6 years

Invoicing services for web shops

Distributed database:
transactions, distribution,
replication

Tuple storage



Race
conditions?

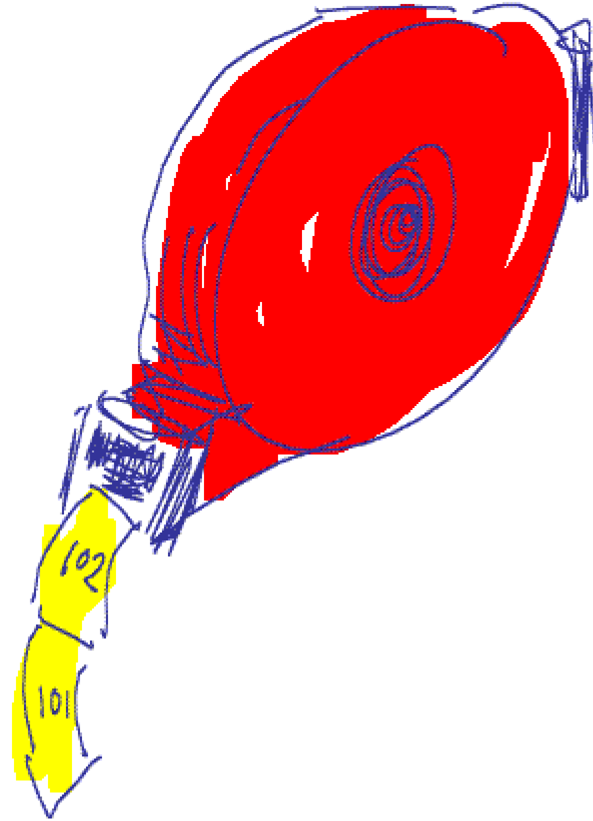
QuickCheck

- 1999—invented by Koen Claessen and me (ICFP 2000), in Haskell
- 2006—Quviq founded marketing Erlang version
- 2009—Race condition testing method (ICFP)
- Real successes and further developments

Imagine Testing This…

dispenser:take_ticket()

dispenser:reset()



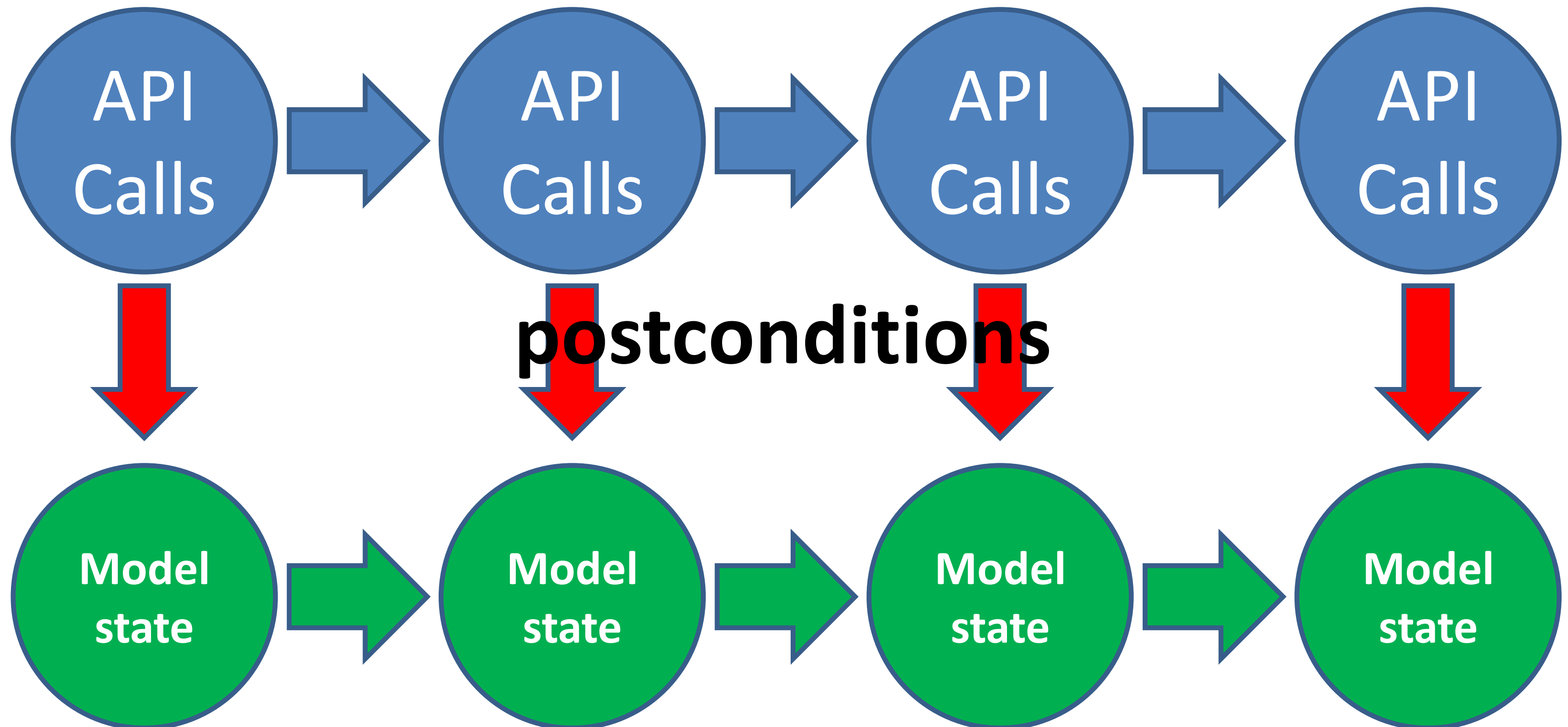
A Unit Test in Erlang

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

Side-effects require a *sequence* of calls to test

Expected results

State Machine Specifications



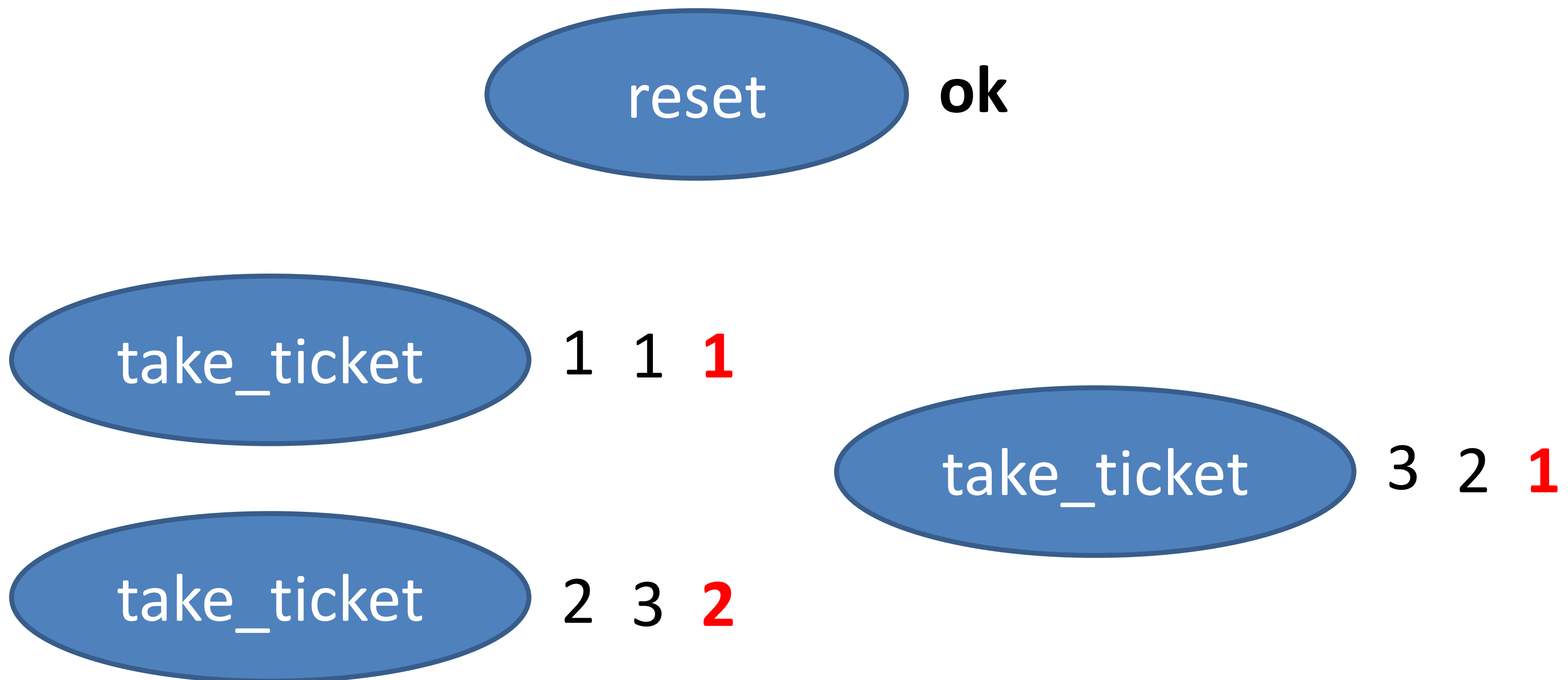
Modelling the dispenser

reset take take take

0 0 1 2

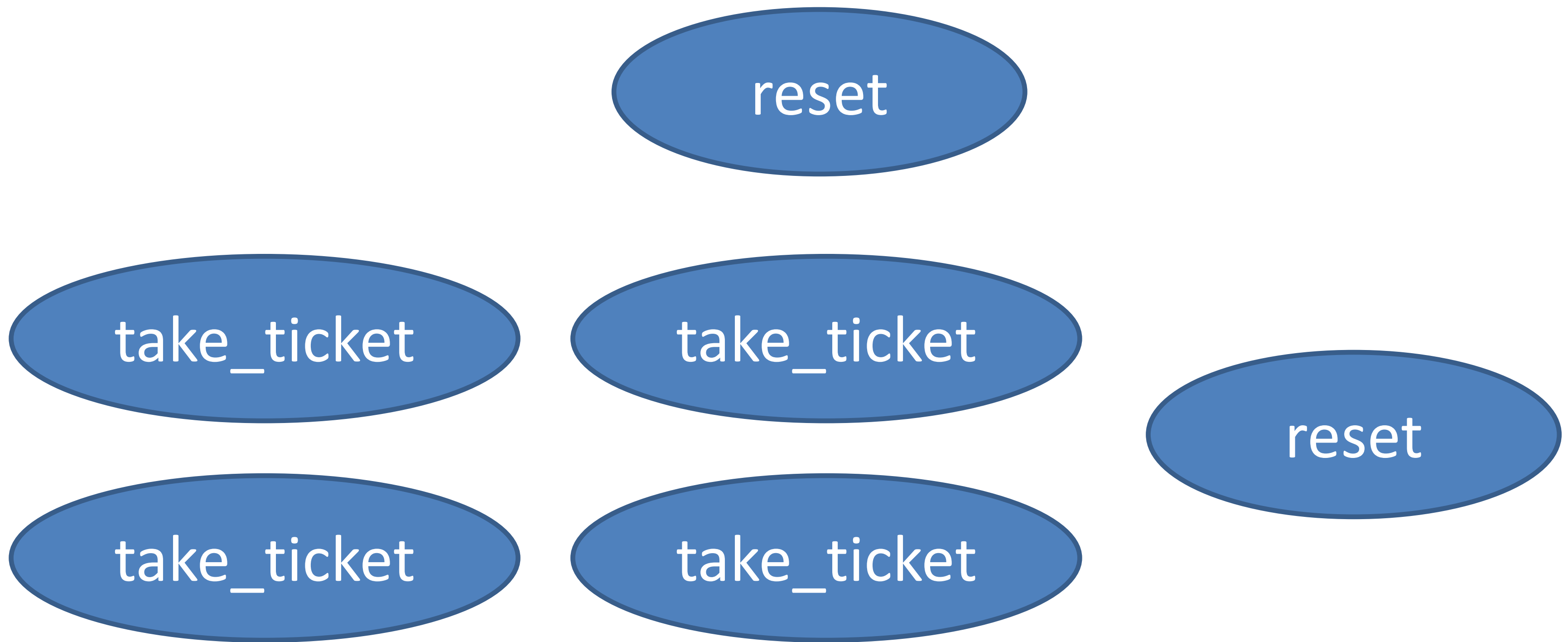
ok 1 2 3

A Parallel Unit Test



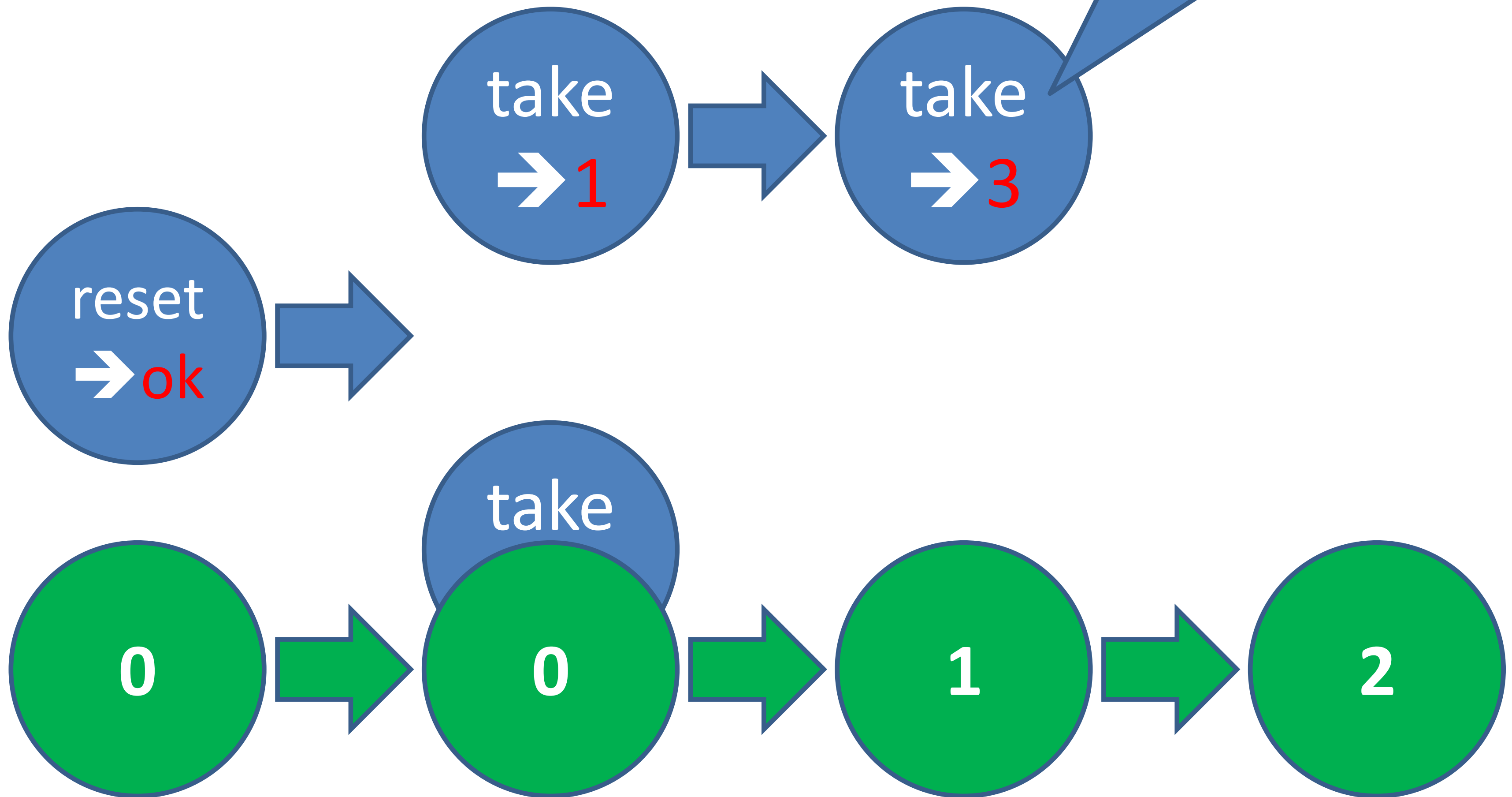
- Three possible correct outcomes!

Another Parallel Test



- 42 possible correct outcomes!

Deciding a Parallel



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

Prefix:

Parallel:

1. dispenser:take_ticket() --> 1

2. dispenser:take_ticket() --> 1

Result: no_possible_interleaving

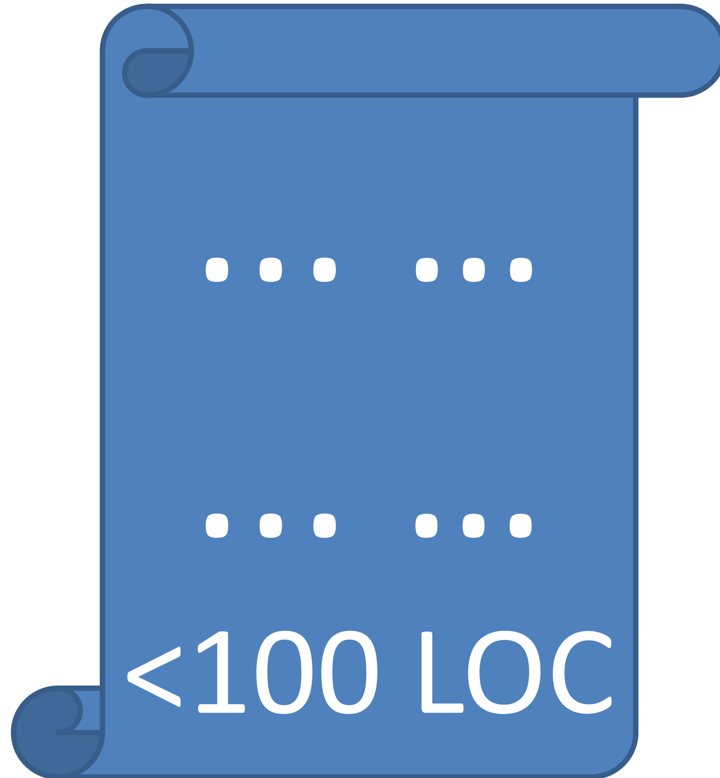
dets

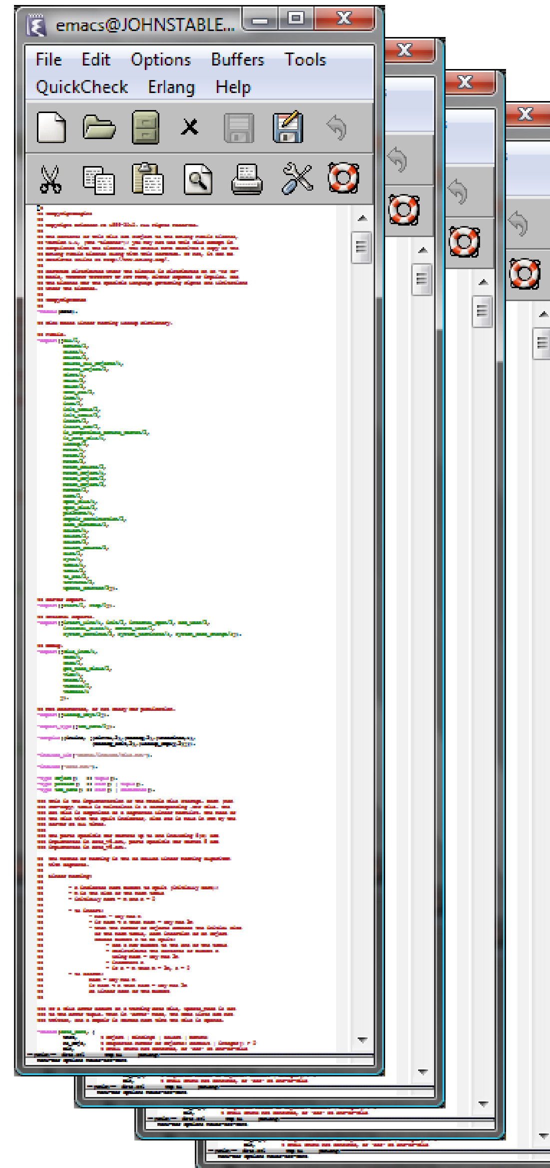
- Tuple store:

  {Key, Value1, Value2…}

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

QuickCheck Specification





> 6,000 LOC

Bug #1

```
Prefix:
   open_file(dets

Parallel:
1. insert(dets_ta

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

Result: no_possible_interleaving
```

insert_new(Name, Objects) -> Bool

Types:

Name = name()

Objects = object() | [object()]

Bool = bool()

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
```

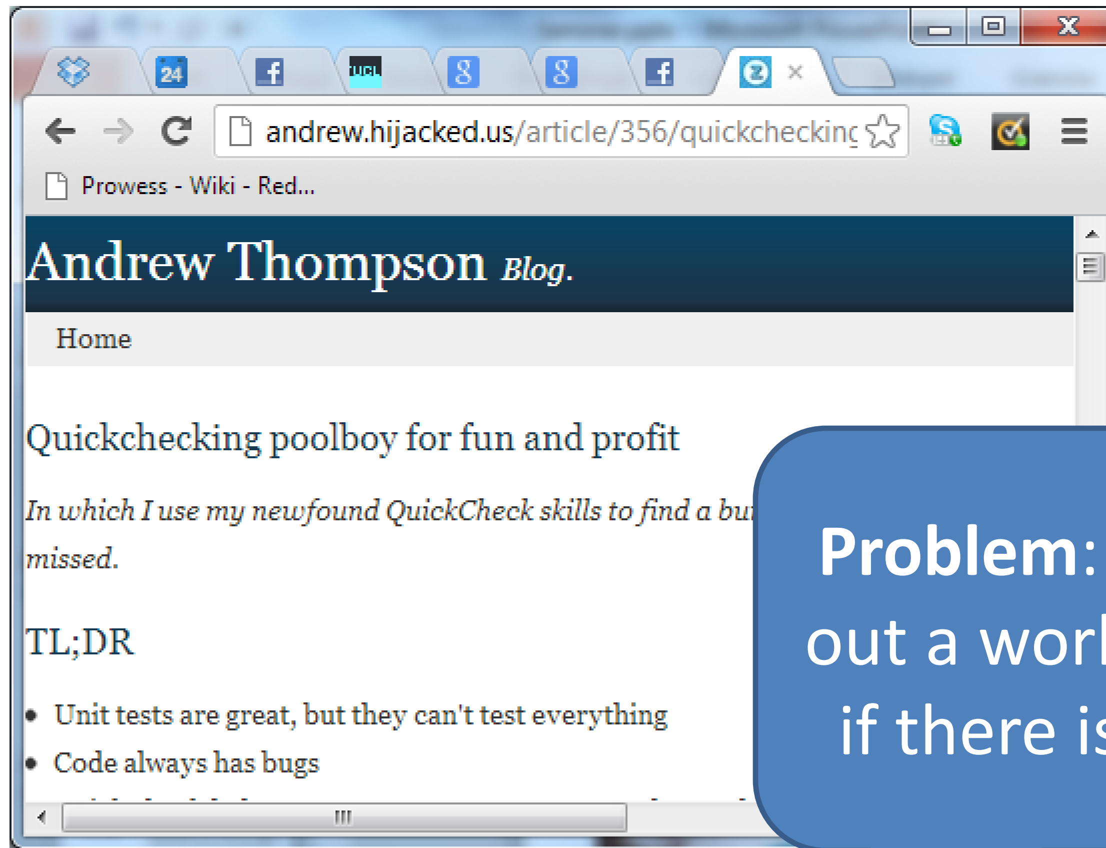
"We know there is a lurking bug somewhere in the dets code. We have got 'bad object' and 'premature eof' every other month the last year."

Tobbe Törnqvist, Klarna, 2007

Each bug fixed the day after reporting the failing case

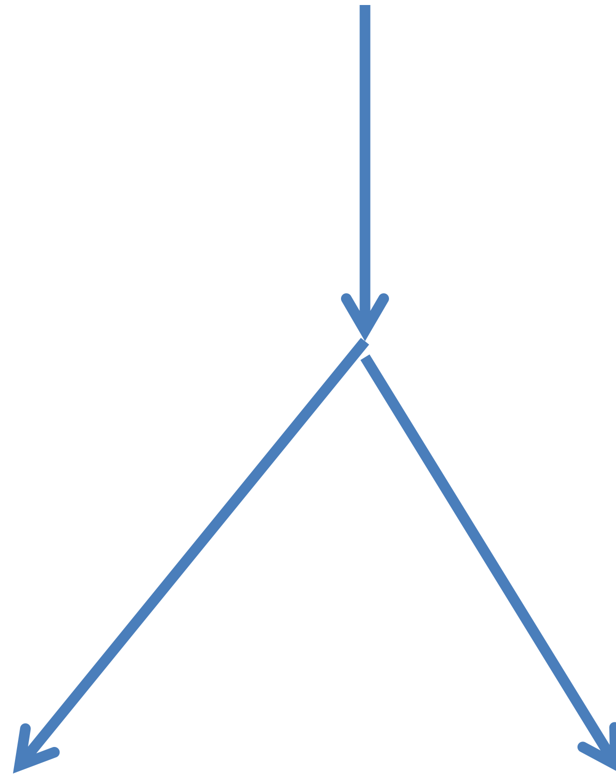
Testing a Worker Pool

- Check out a worker
- Check in a worker
- Handle workers crashing
- Handle clients crashing while holding a worker



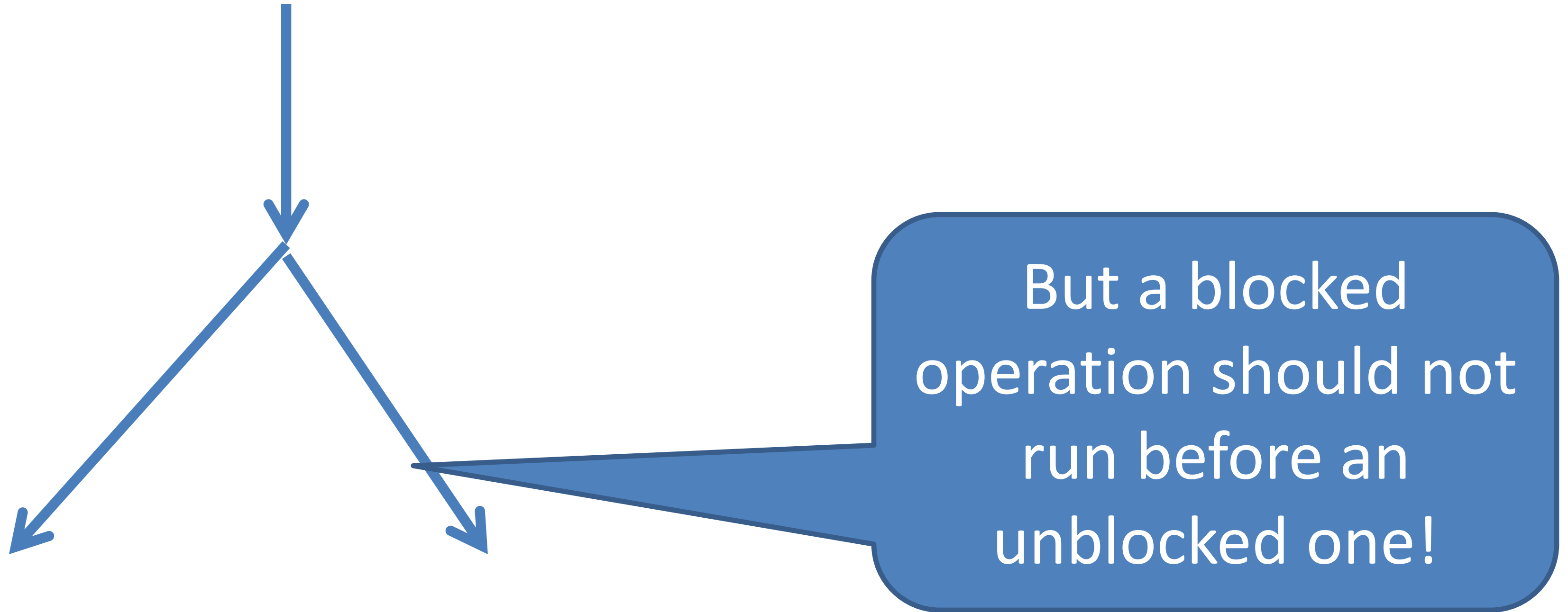
- Loads and loads of bugs found
- 80 unit tests passed throughout!
- Parallel testing found no race conditions

Blocking operations



Test deadlocks? In practice, lock *times out*

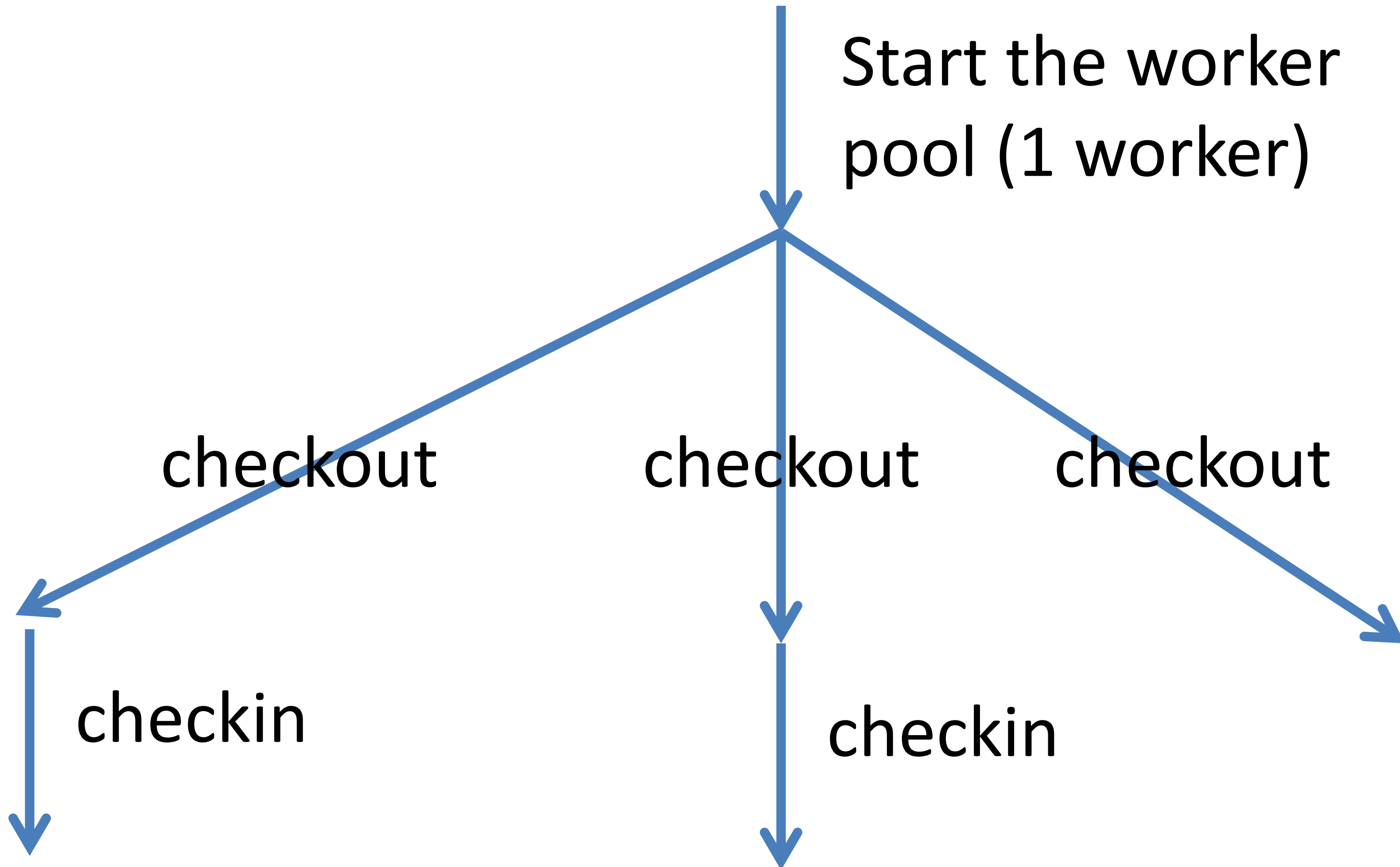
Should this test pass?



Serializability with Blocking

- Specify when an atomic operation should block
- When exploring interleavings, *never choose a blocked operation when there is an unblocked alternative*
- We *rule out* some interleavings, potentially making test fail that would otherwise have passed

A race condition in Poolboy?



Conclusion

- Serializability is a
  - *simple condition*
  - that is *surprisingly effective*
  - at revealing bugs in real industrial code

Not quite done…

Provoking races

- We’ve used:
  - Repeated execution on a multicore processor
  - Random scheduling
  - ”Procrastination”… repeating a test, but reordering message deliveries to the same process
  - Model checking—all possible schedules