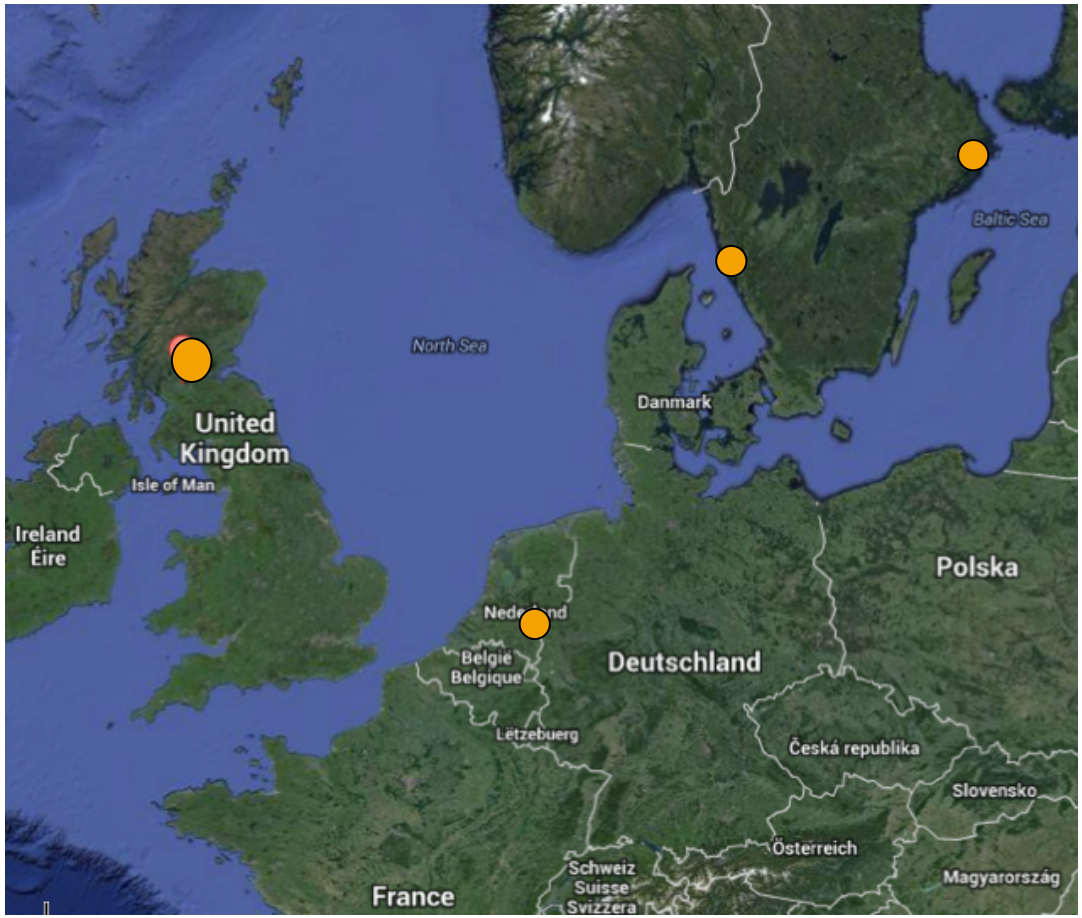
QuviQ

Property-Based Testing

Quviq AB
Thomas Arts

Functional Programming late 80ies

Haskell
(Academia)

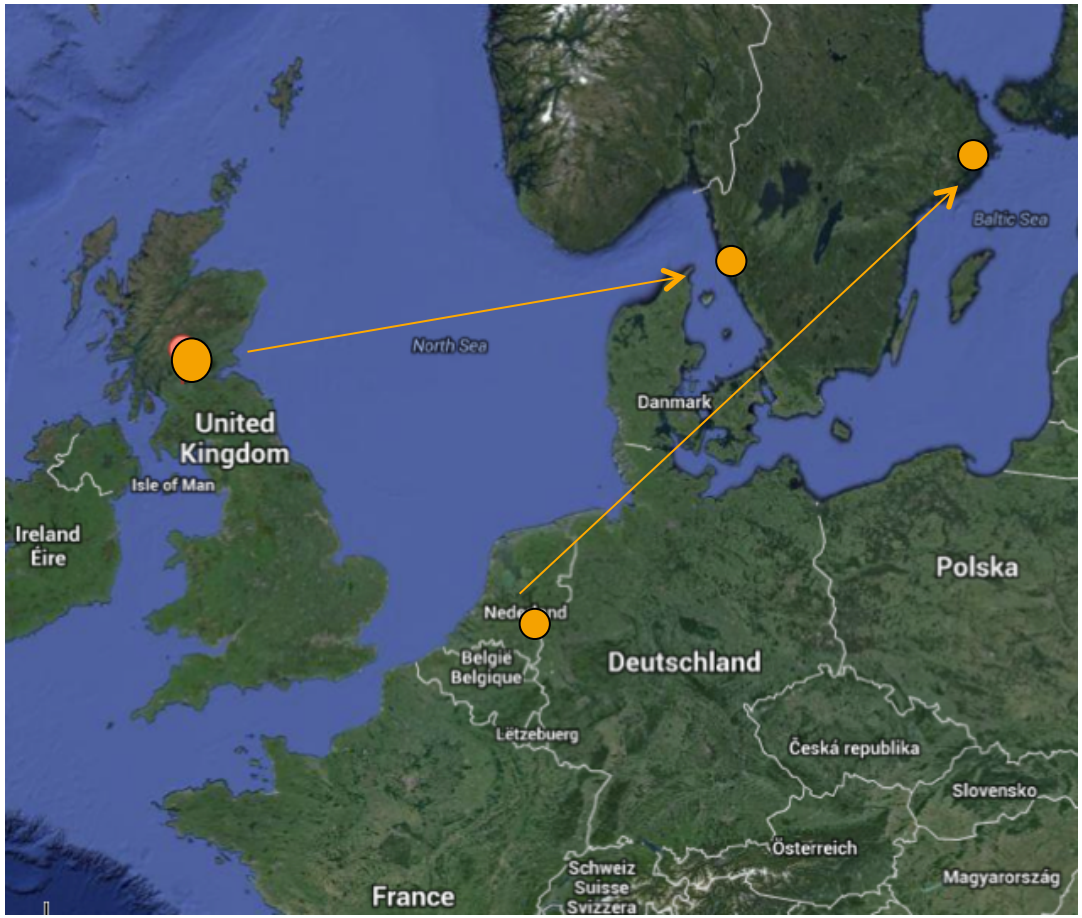


Erlang
(Ericsson)

90ies: Sweden strong in Functional Programming

Haskell
(Academia)

John Hughes
professor in
Göteborg



Erlang
(Ericsson)

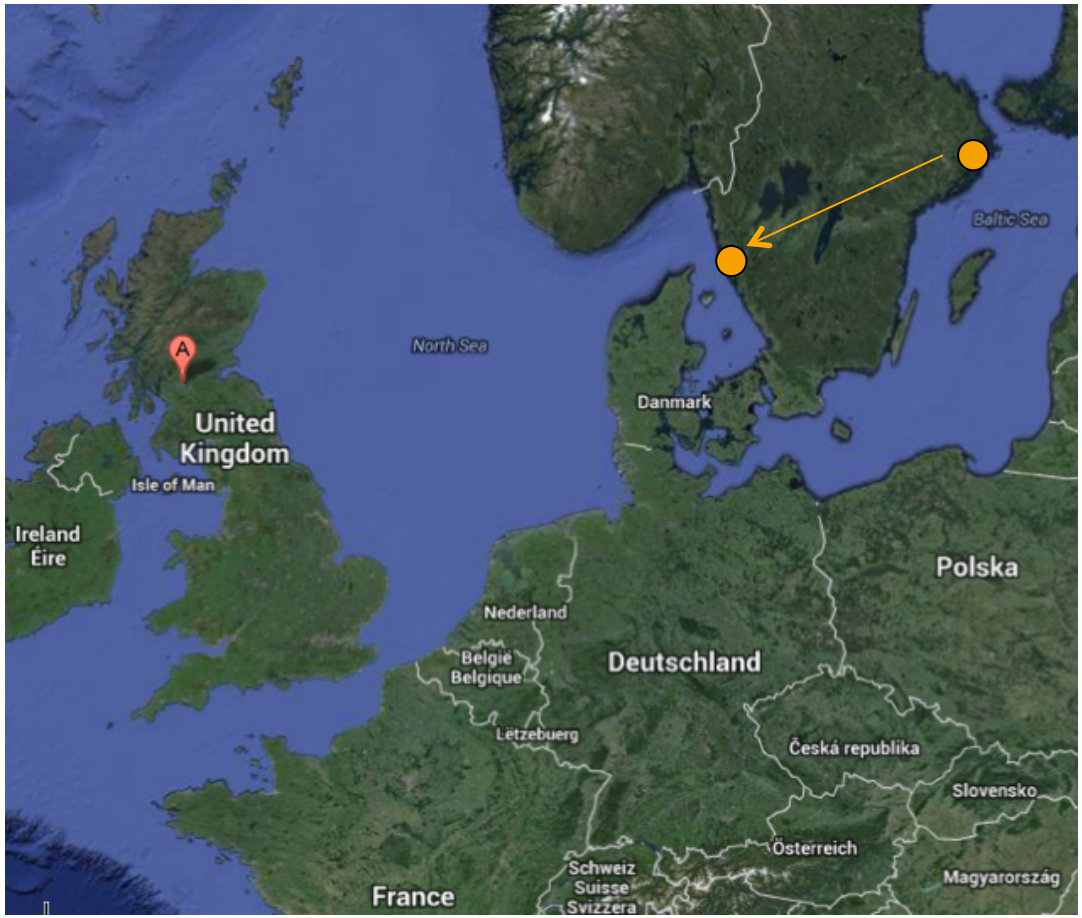
Thomas Arts
Ericsson
Research lab

2000 Property-based testing invented

Haskell
(Academia)

John Hughes
professor in
Göteborg

QuickCheck:
Generating
test cases



Erlang
(Ericsson)

Thomas Arts
Ericsson
Research lab

experience in
formal verification

professor in
Göteborg

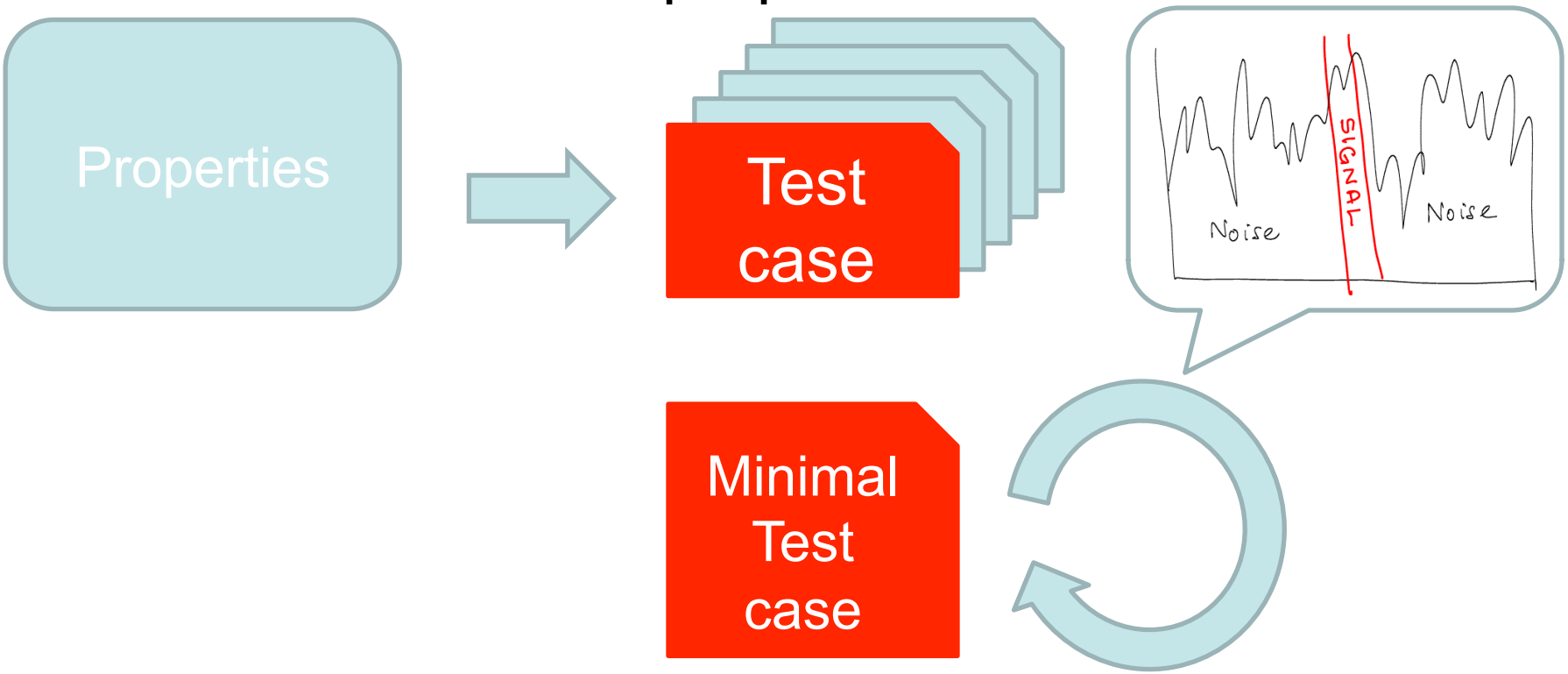
We identify software *properties*

- For all possible sequence of allowed database queries, business rule R is never violated
- Under all circumstances, highest priority signals are submitted first
- Each raised alarm is cleared after soft restart
- the legacy software behaves exactly the same as the new product

How does it work?

Requirements provide us software properties

QuickCheck is a tool that automatically generates test cases from these properties.



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

Based on a property, we test a program

An example: SMS encode/decode

GSM standard

Technical Specification **03.38**

Version 5.3.0, July 1996

Packing of 7 bit characters

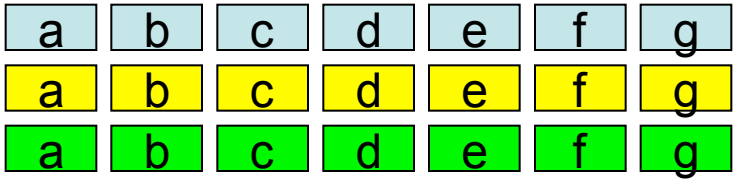
6.1.2.1.1 (page 13-14)

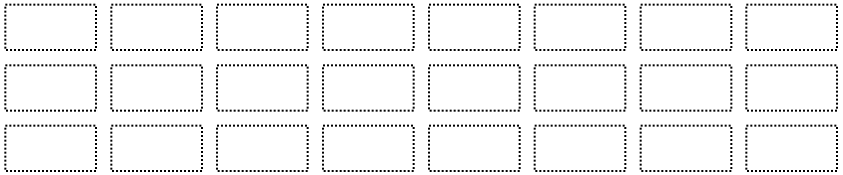
Example

Interpretation

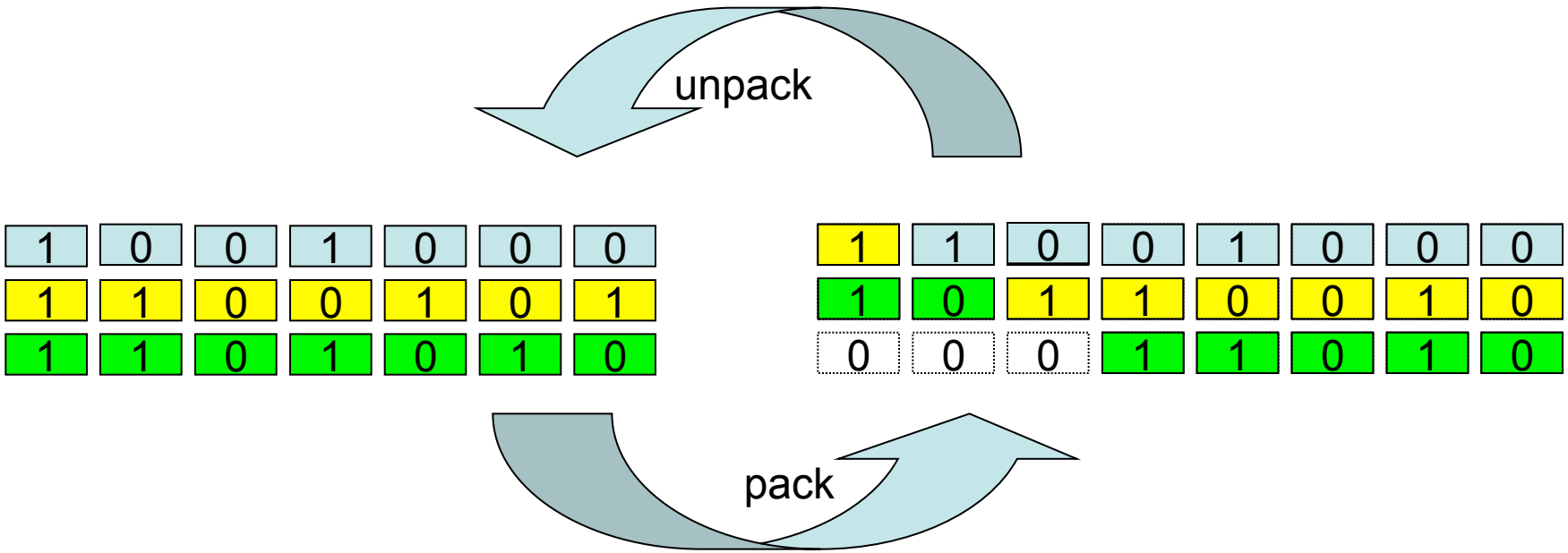
3 characters containing 7 bits

packets 8 bits





Example



“Hej” = [72,101,106]

“È²^Z” = [200,178,26]

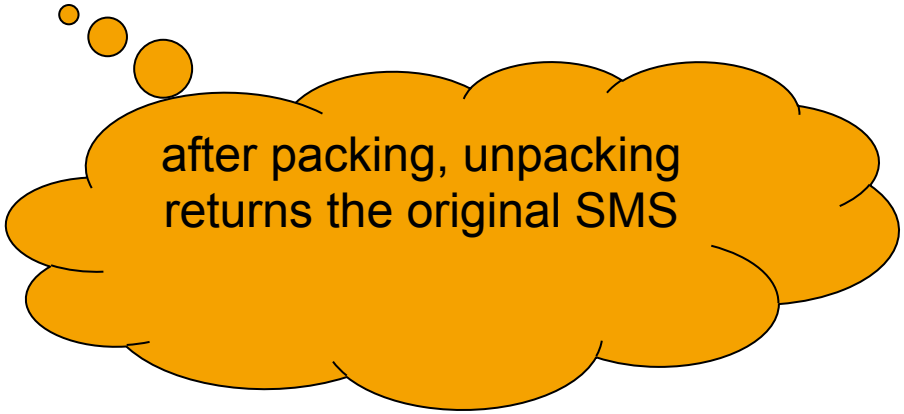
Two properties

```
prop_compress() ->
  ?FORALL(Len, choose(8,160),
    ?FORALL(Msg, vector(Len,choose(0,127)),
                size(sms:pack(Msg)) < length(Msg))).
```

An SMS of 8 characters or longer, is shorter than 8 bytes after packing

Two properties

```
prop_identity() ->
    ?FORALL(Msg, list(choose(0,127)),
                sms:unpack(sms:pack(Msg)) == Msg).
```



Example

DEMO

Benefits

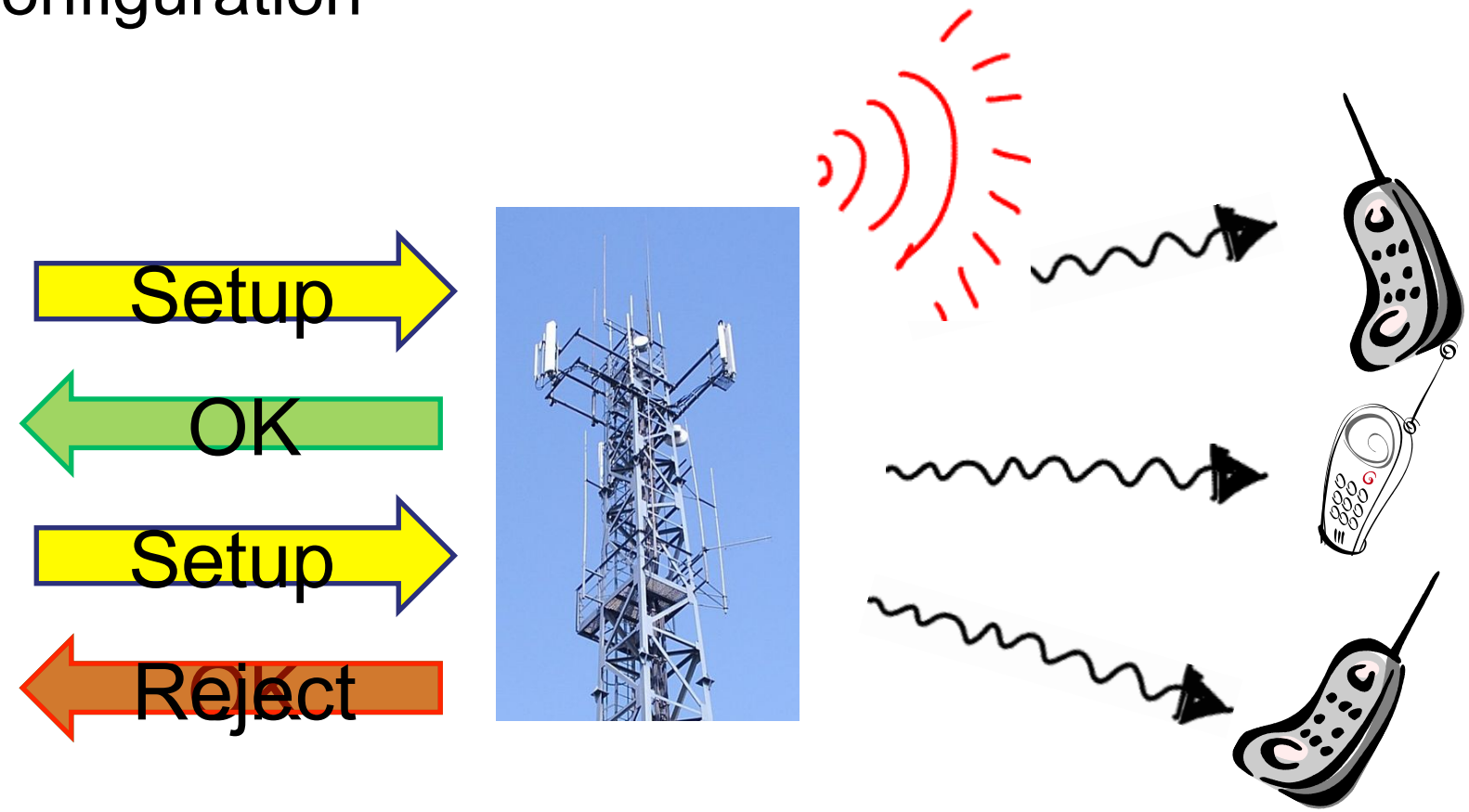
- Less time spent writing test code
  - One property replaces many tests
- Better testing
  - Lots of combinations you'd never test by hand
- Less time spent on diagnosis
  - Failures minimized automagically

In real software, properties are more complex

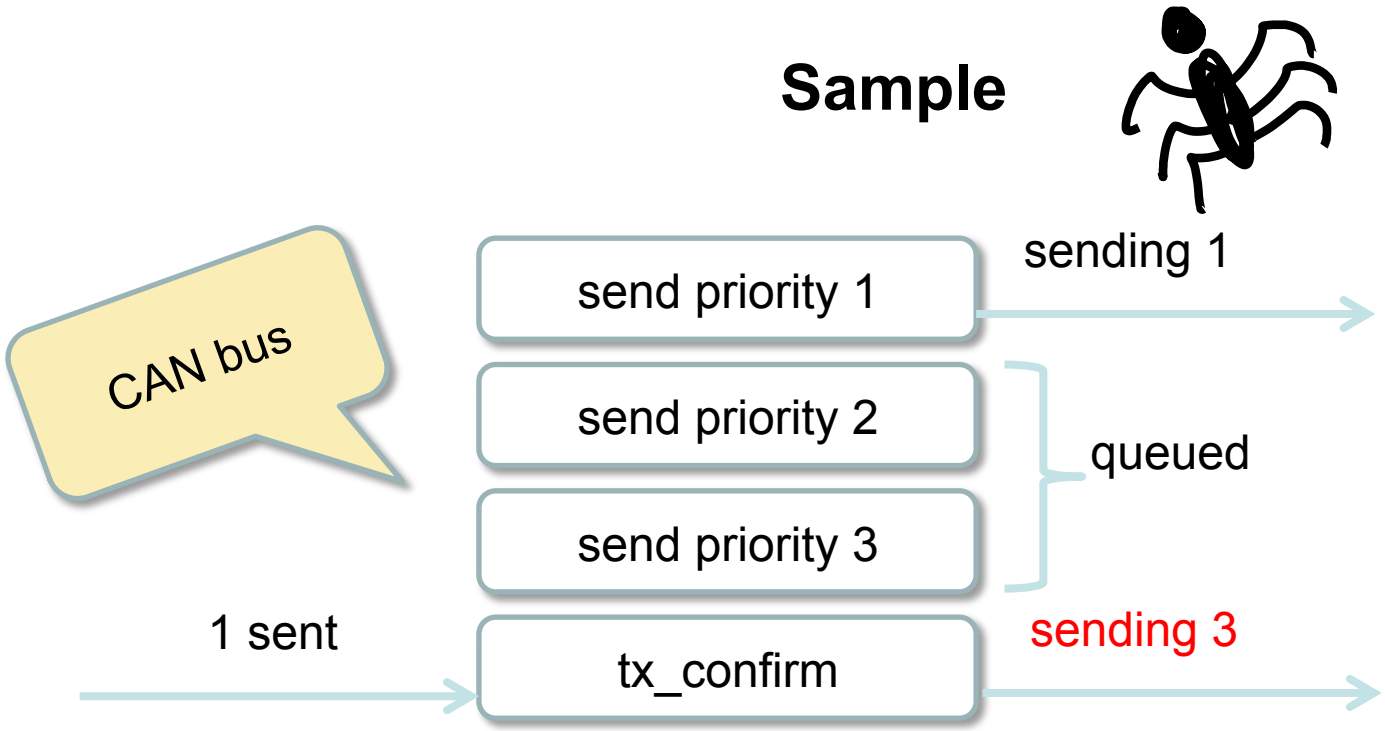
Real software has state... properties need to be stateful and *generate a sequence of commands*

Shrinking to minimal failing test case saves a lot of time in analysis

Testing cell and channel setup/teardown and configuration



Property: highest priority message should be send first



Cause: failure to mask a bit off an extended CAN-identifier

two examples

Financial sector: finding race conditions

Automotive: highly configurable software

What is it?

500+ people in 5 years

Application	klarna Invoicing services for web shops
Mnesia	Distributed database: transactions, distribution, replication
Dets	Tuple storage
File system	

Race conditions?

Bug in Klarna's financial database

- Appearing in production once a month
- subtle, not found by six weeks of investigation with conventional methods
- Thought to appear in databases of ~ 1GB.

Finding race-conditions

Only special circumstances trigger a specific race

Observation in fault-tolerant software difficult

Traditional approach: try to find race-condition related errors in *integration testing*… try to simulate extreme cases and see what happens.

If you observe the error, you don't know what really caused it!

If you observe the error, you don't know what really caused it!

Analysis is expensive.

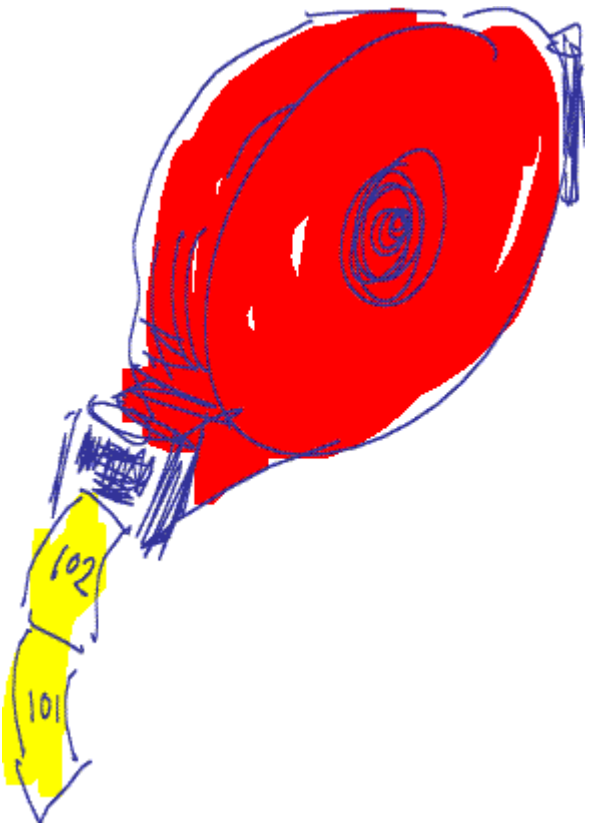
… we see it after 25000 function calls, 491 communicating processes, and with 1GB data loaded, …sometimes…

Analysis for traditional unit testing is cheaper… but **we cannot write unit tests for concurrency**!

Imagine Testing This…

dispenser:take_ticket()

dispenser:reset()



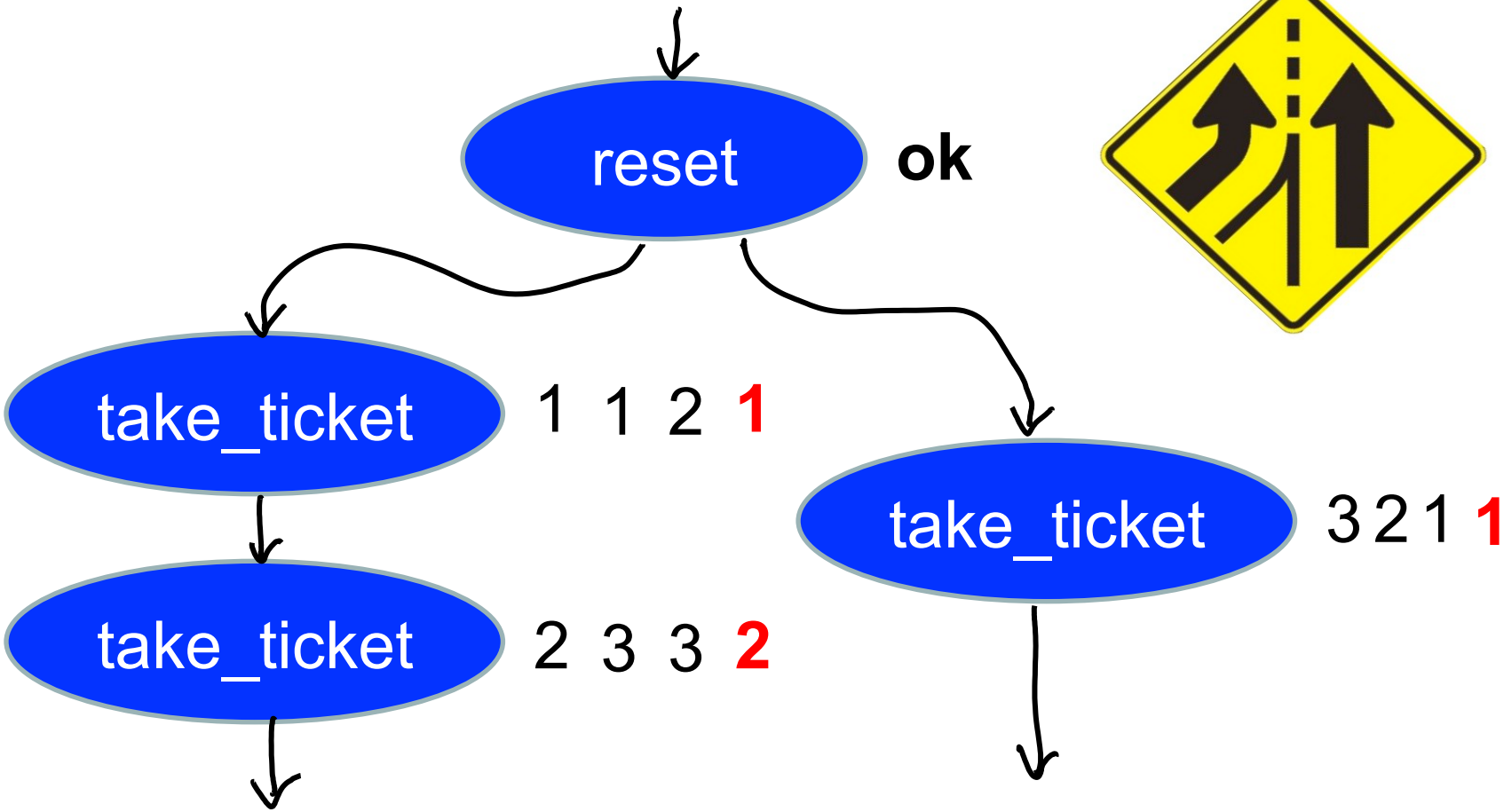
A Unit Test in Erlang

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

Expected results

BUT …

A Parallel Unit Test



- Three possible correct outcomes!

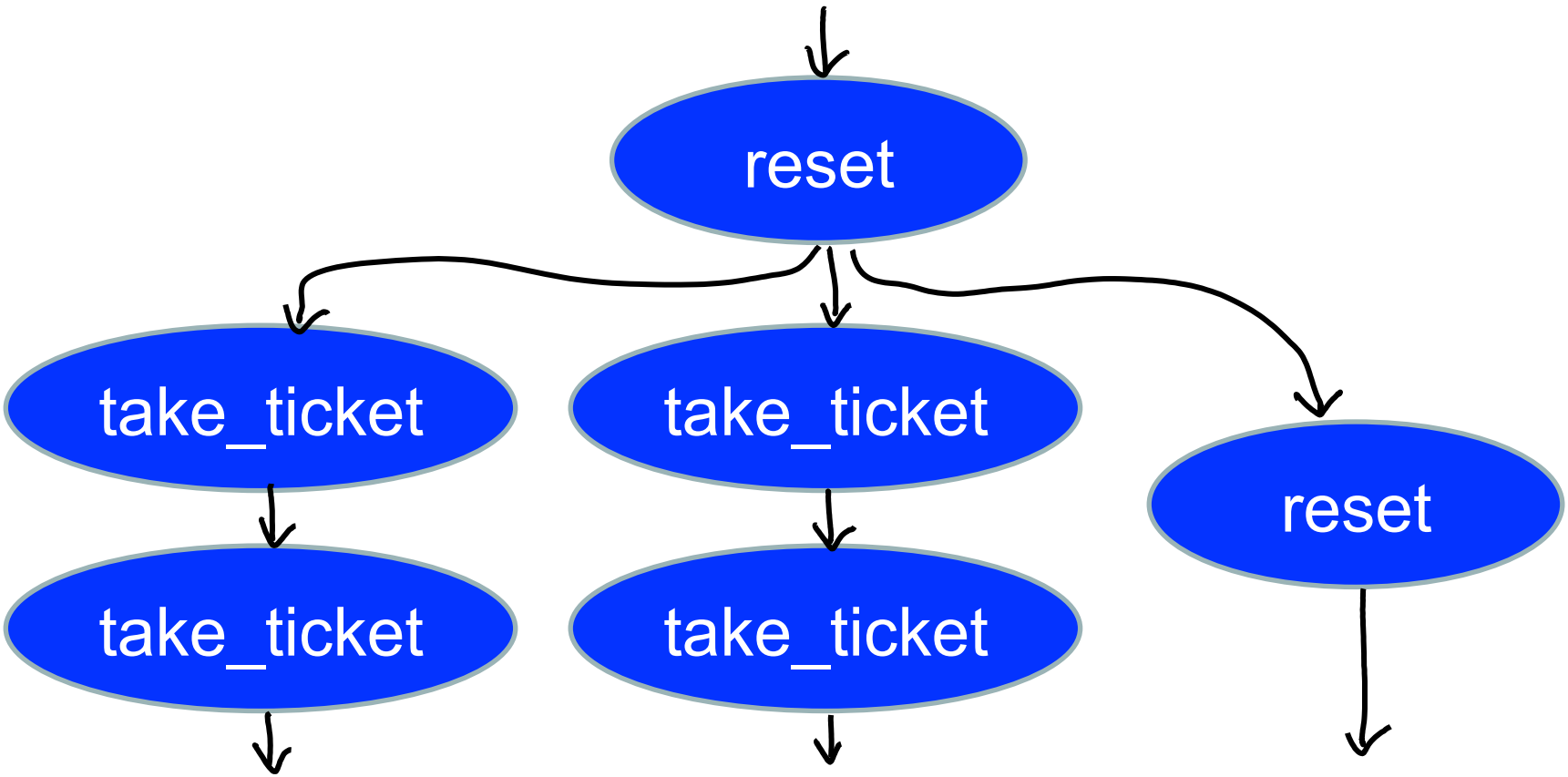
A Unit Test in Erlang

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

Expected results

BUT …

Another Parallel Test



30 possible correct outcomes!

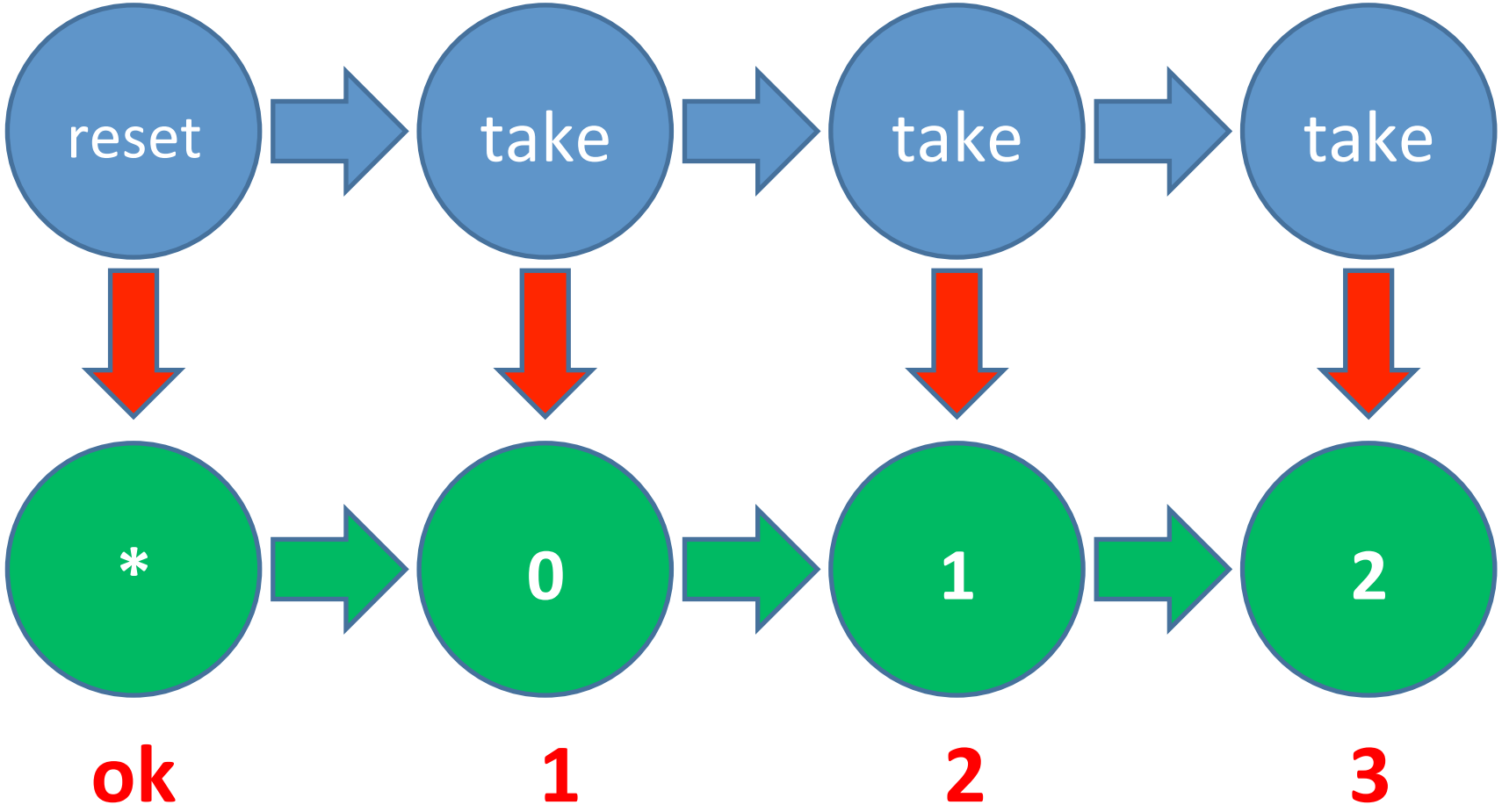
Writing unit tests for concurrent events:

Headache!

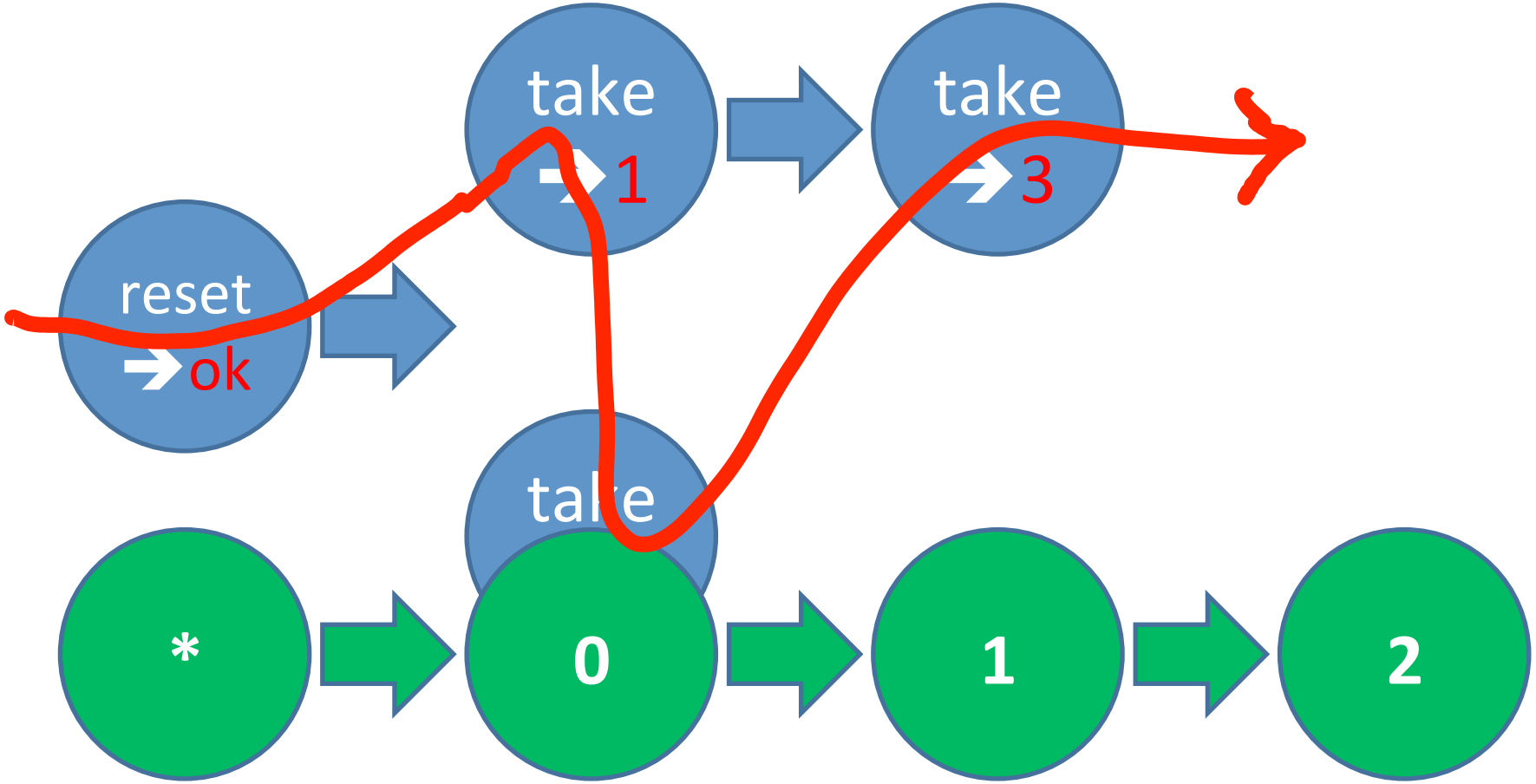
Thus, people don't!

QuickCheck does it for you!

Modelling the dispenser



Parallel Test Cases



QuickCheck properties:

Property specifies behaviour of any command sequence

QuickCheck

- runs the sequences with different threads
- collect the results
- checks whether this can be explained from sequential behaviour

Bug in Klarna's financial database

Before

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

After

- Database with *1* record!
- 5 calls to reproduce
- 2 days model
- < 1 day to fix

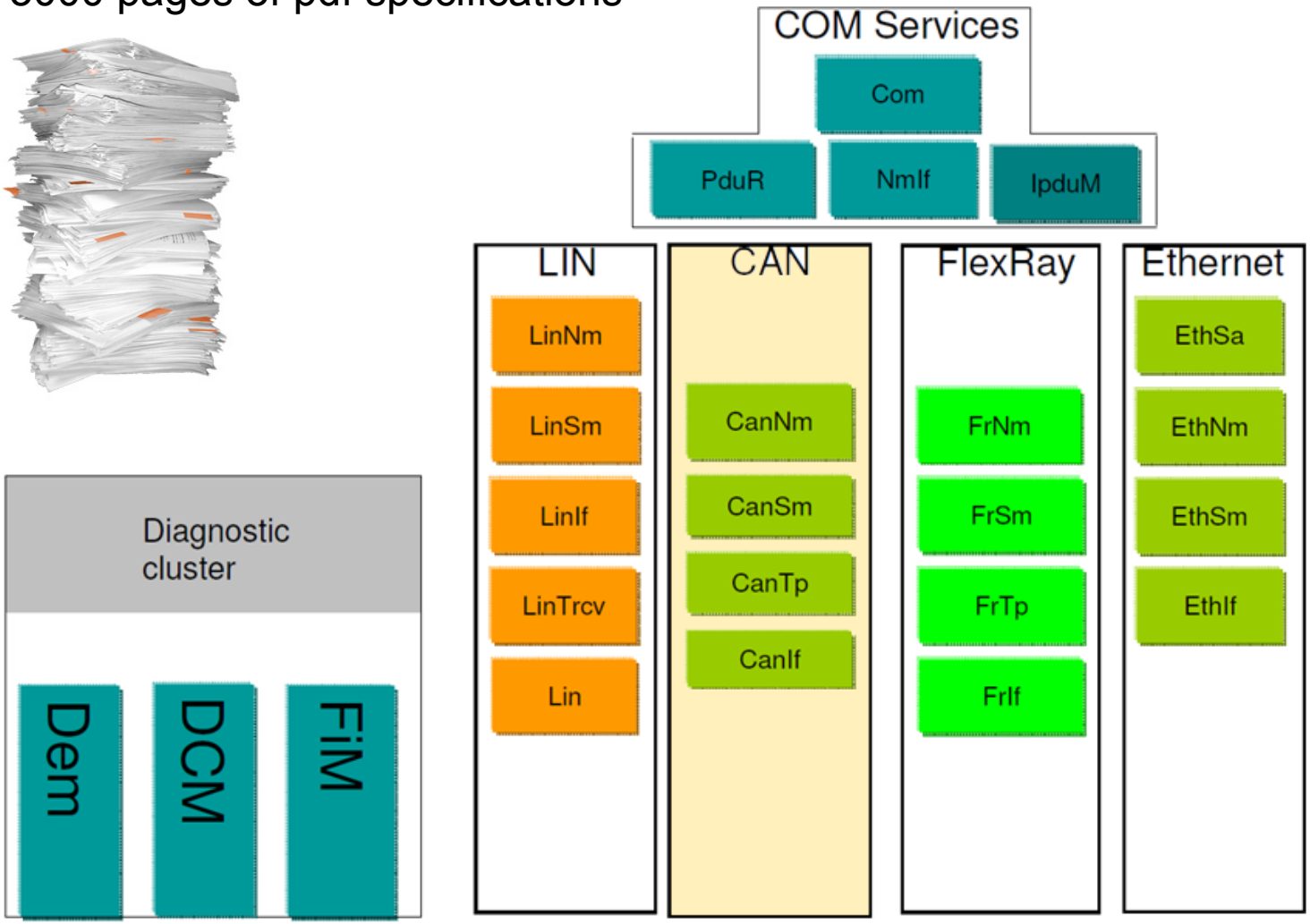
COTS components in automotive domain

AUTOSAR is a standard for software architecture in the automotive industry.

Volvo wants to be able to replace components from one vendor by components of another.

AUTOSAR standard

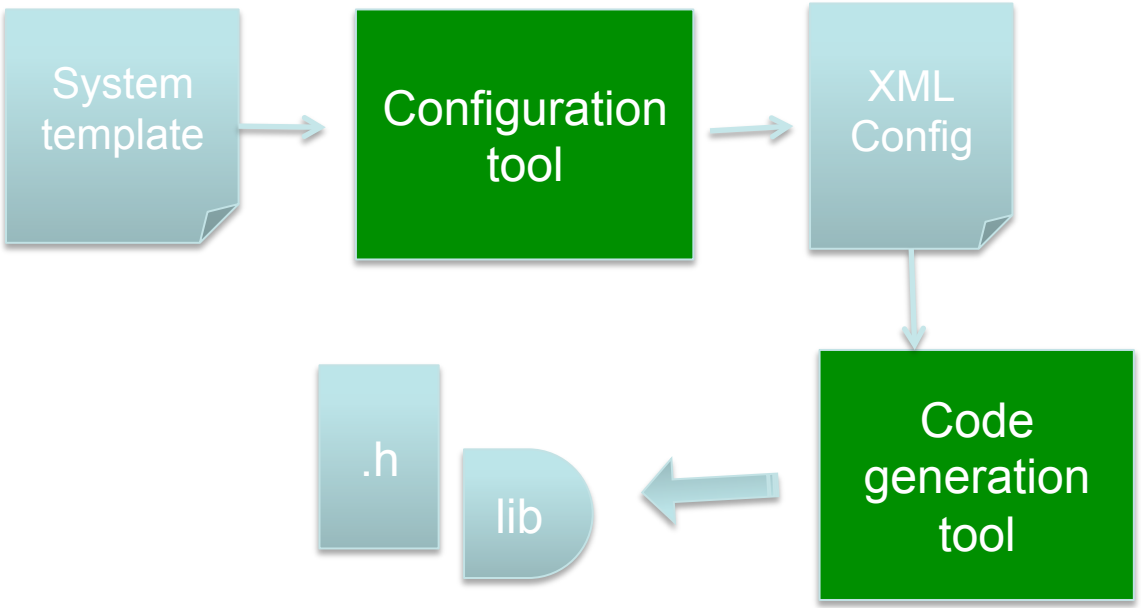
3000 pages of pdf specifications



What is difficult in writing test cases?

- Everything is configurable. Thousands of parameters can be specified
- AUTOSAR is modular. Tests are designed against a specification, but there is no specification for combinations of modules

Configurations are vendor specific



A test is:
A configuration and a set of API calls with their expected results.

Tests

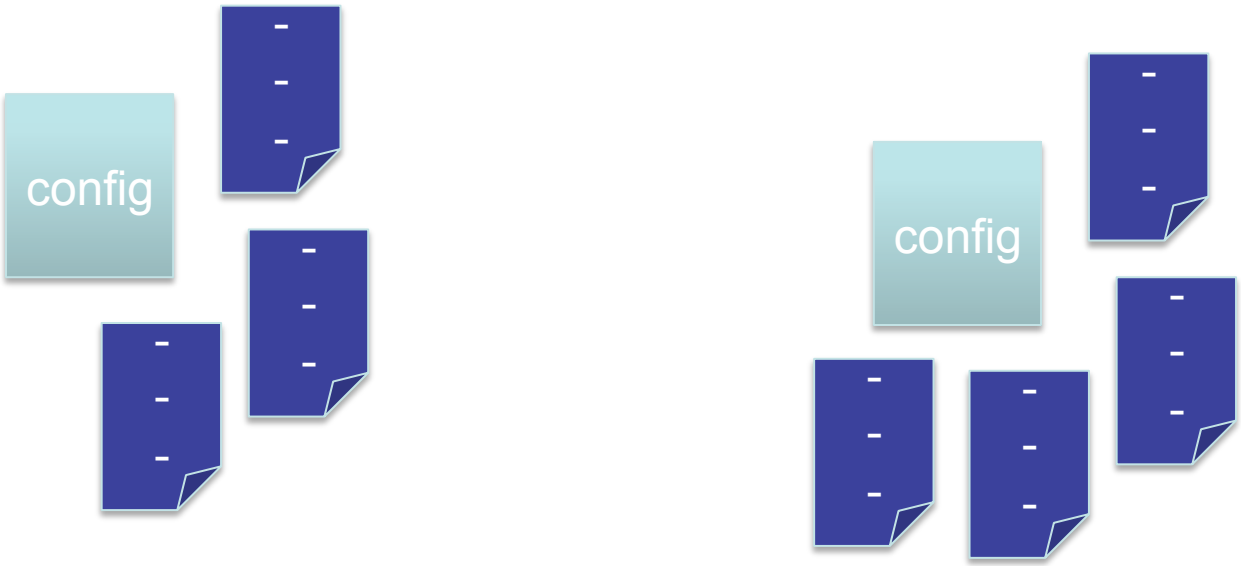
A test is:
A configuration and a set of API calls with their expected results.

For testing configurations are kept small

A number of API call sequences per configuration

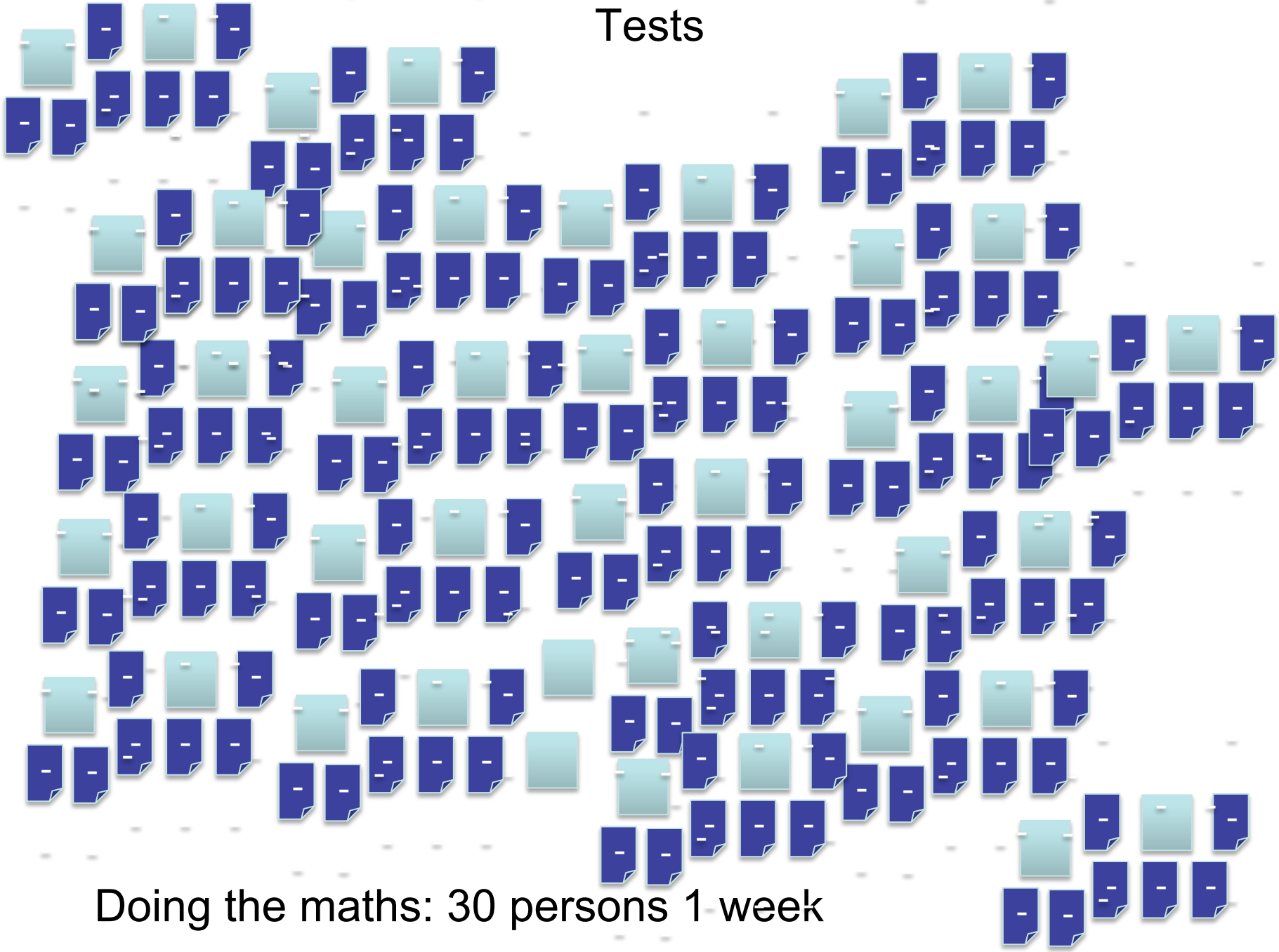
Vendor may need to change configuration a bit before code can be generated and tests can be run.

Tests



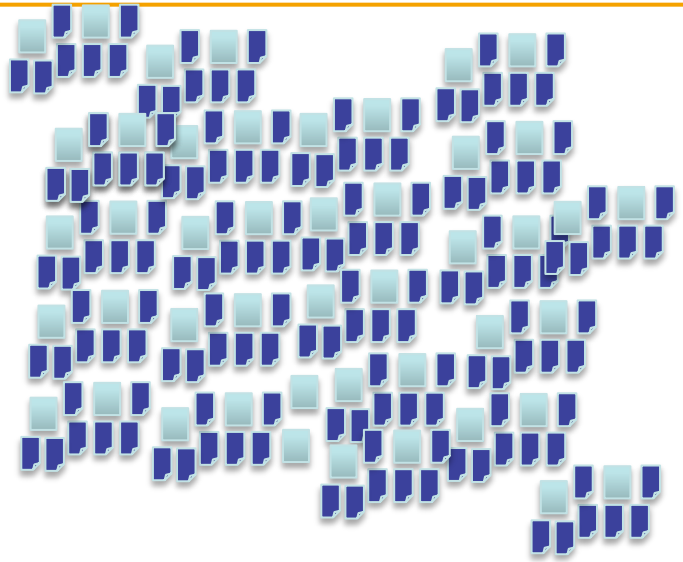
Doing the maths: 1 person 1 week

Tests



Doing the maths: 30 persons 1 week

Tests



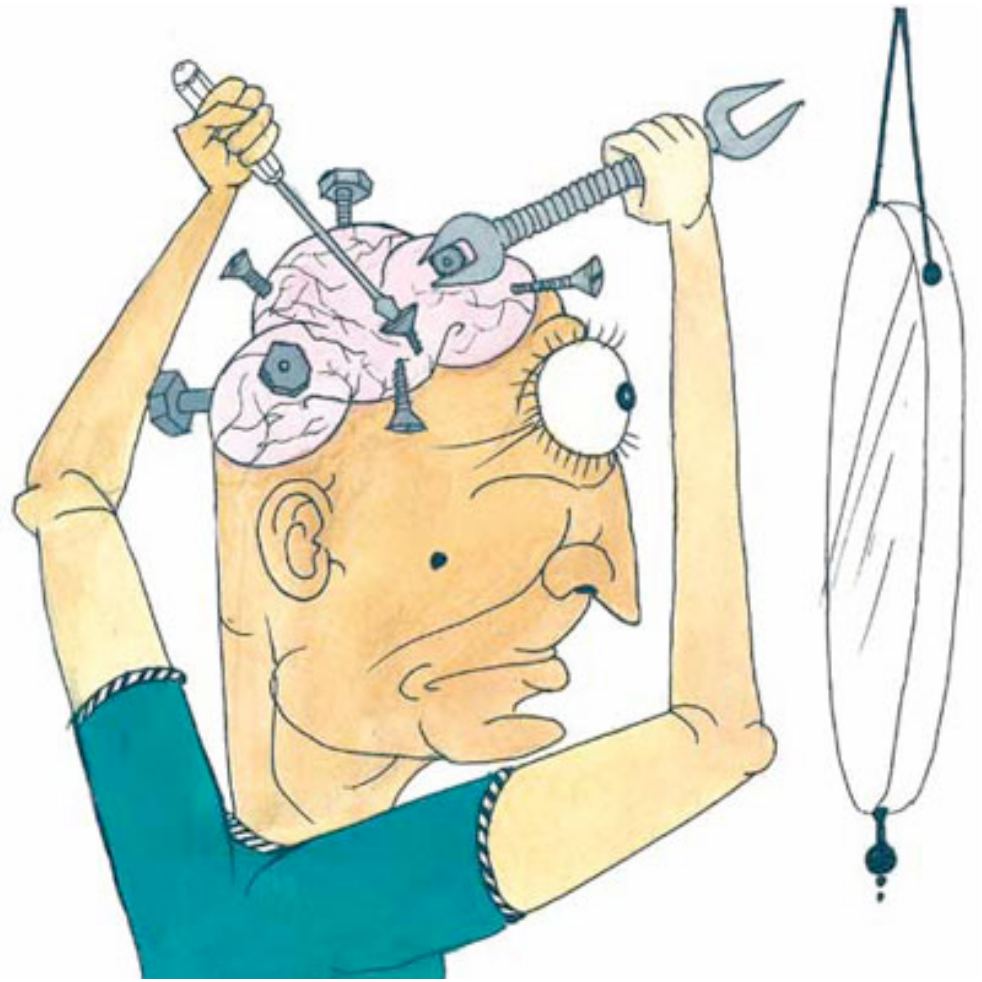
X 50

Needed to cover the specifications and configuration features

Doing the maths:
30 person years, 2-5 tests per week per developer, 10000 tests…. 3000 configurations

Executing those tests is a nightmare, since one needs to adopt the configurations and generate code

Property-based testing

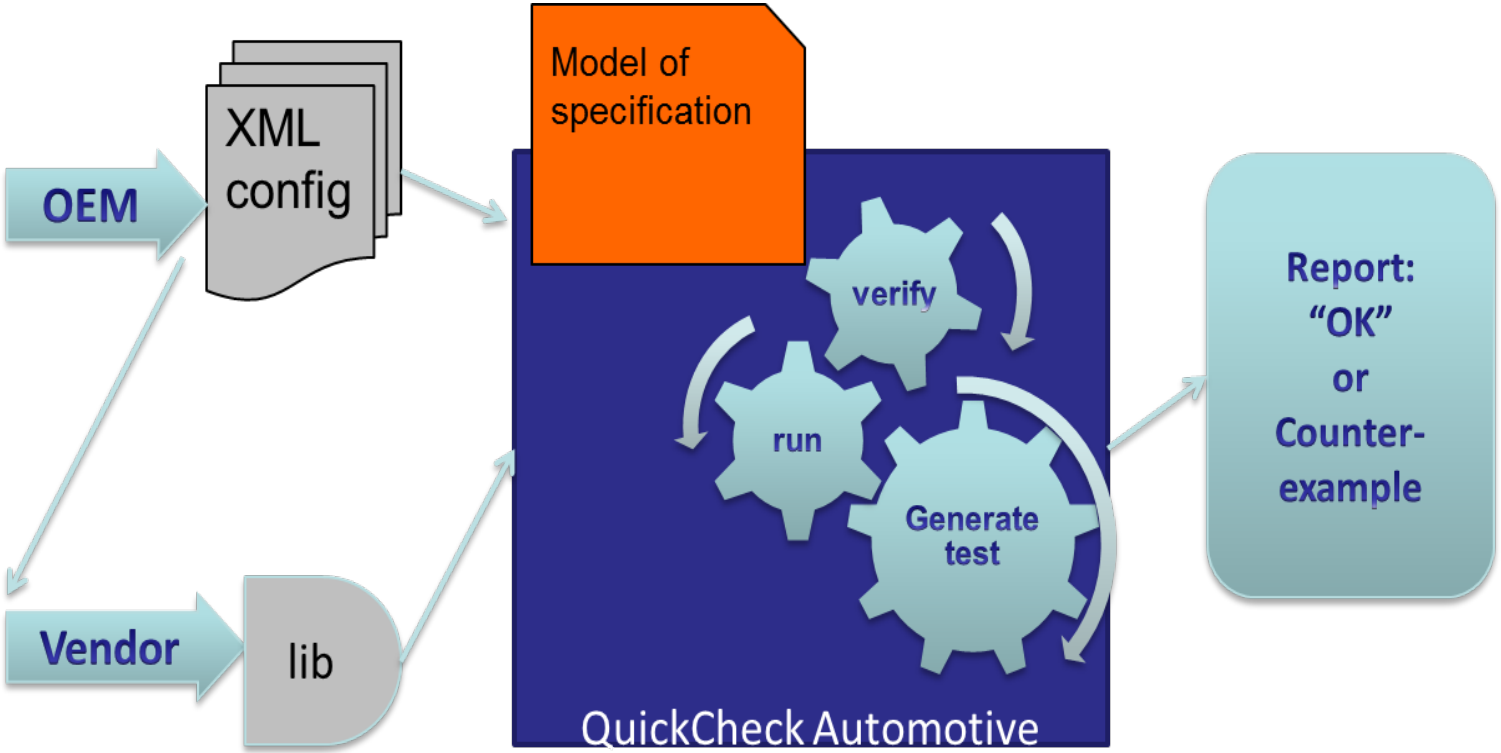


change your mind
about testing

Model-based testing

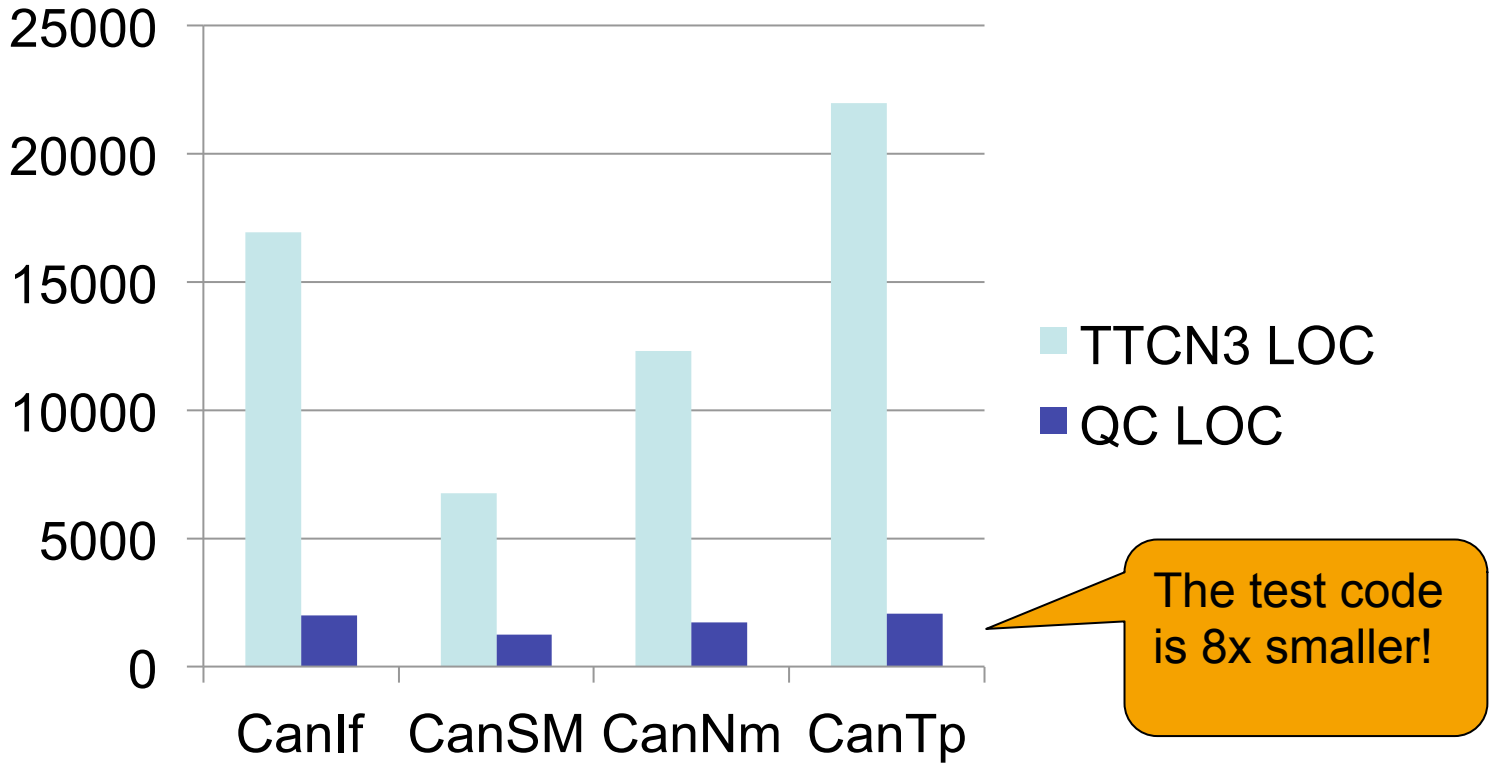
- Compositional models
- One large configuration supporting all test cases
- A huge number of test cases automatically generated
- All features/requirements tested at the same time
  - Many API's at once, all assertions always around and random sequences no-one would consider testing

Highly configurable

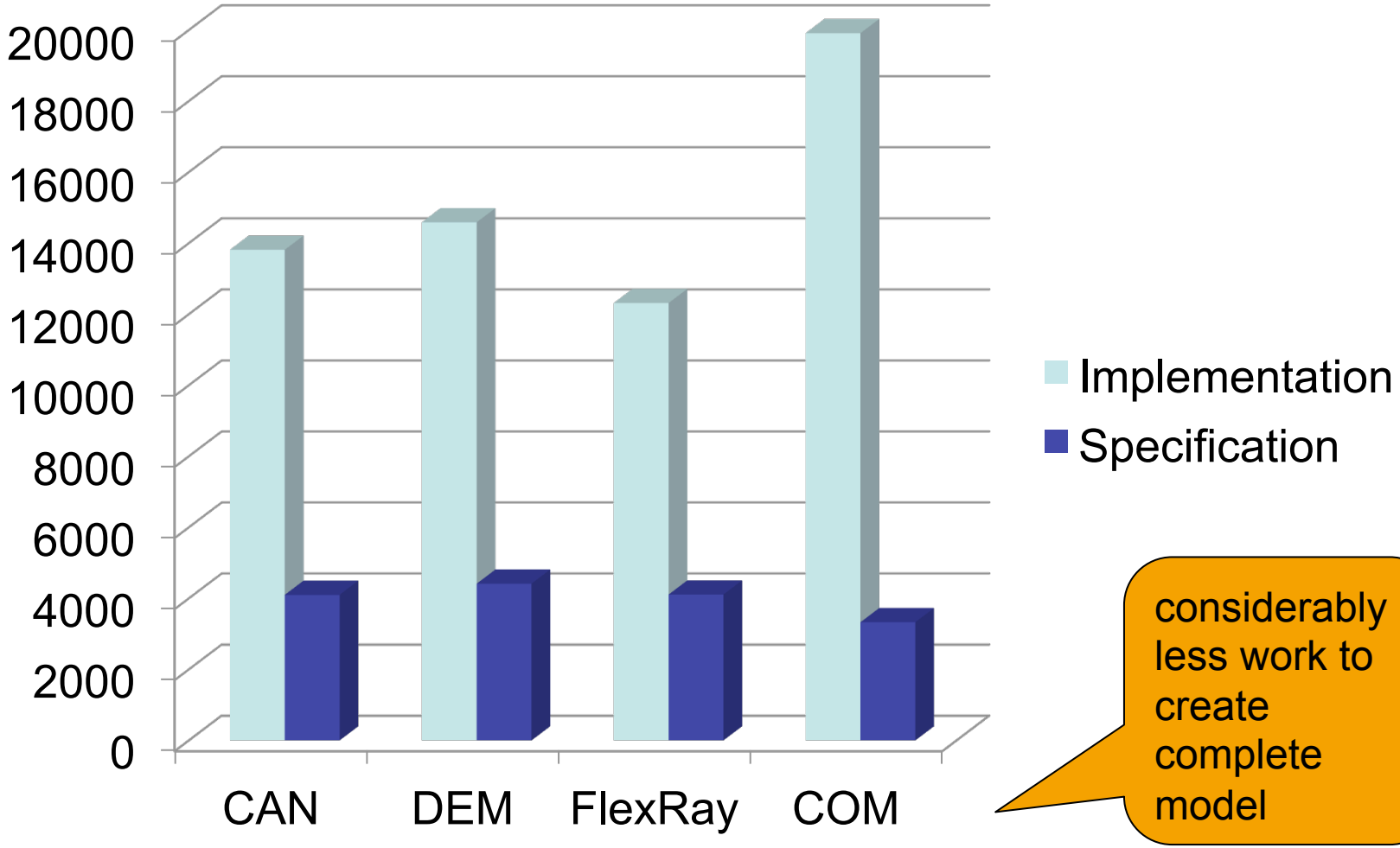


TTCN-3 test suite for CAN modules:

245 test cases, 58KLOC



The Problem of Scale



Advantages

- Over 100 errors found in 4 weeks of testing production ready code
- Erroneous dependencies between features found
  - Mix of many features tested in same tests
- Failures found in “obvious fault-free implementation”
  - Everything is tested, even parts otherwise excluded by manual tests
- General higher coverage
  - Many more tests executed
  - All assertions always considered
- Common human mistakes detected
  - Common human errors for both developer and test designer are found by model

Jan 6 – Jan 31

Confirmed errors in production code

- Com: over 20
- Can: over 30, mostly in CanSM and CanTp
- Lin: 5 errors, Requesting a schedule cannot be tested due to incompatibility.

In some cases, model more precise than vendors want to be.

Starting a TP Gateway

```
PduR_Init( ...);
PduR_FrTpStartOfReception( 'PduR_Path15_S', 30, ...);
```

Vendor1: `{ 0,'BUFREQ_E_OVFL'}`

Vendor2: `{29,'BUFREQ_OK'}`

Vendor3: `{ 0,'BUFREQ_E_OVFL'}`

Vendor4: `{29,'BUFREQ_OK'}`

Scope reduction: do not test Tp Gateway with messages larger than buffer size

Starting a TP Gateway

```
PduR_Init(...);
PduR_FrTpStartOfReception( 'PduR_Path15_S', 8, ...);
```

Vendor1: `{29,'BUFREQ_OK'}`

Vendor2: `{29,'BUFREQ_OK'}`

Vendor3: `{ 8,'BUFREQ_OK'}`

Vendor4: `{ 8,'BUFREQ_OK'}`

Model Variant: return either max buffer size (i.e. 29) or maximum requested size (i.e. 8).

Starting a TP Gateway

```
PduR_Init(...);
PduR_FrTpStartOfReception( 'PduR_Path15_S', 0, ...);
```

Vendor1: `{29,'BUFREQ_OK'}`

Vendor2: `{29,'BUFREQ_OK'}`

Vendor3: `{ 0,'BUFREQ_OK'}`

Vendor4: `{ 0,'BUFREQ_OK'}`

Violation of PDUR507:

The service shall provide the currently available maximum buffer size when invoked with TpSdulength equal to 0.

Comparing implementations

- reveals ambiguities in specifications
- reveals freedom in specifications

A test that succeeds for one vendor may fail for all others.

Static tests are not re-usable!

Comparing implementations

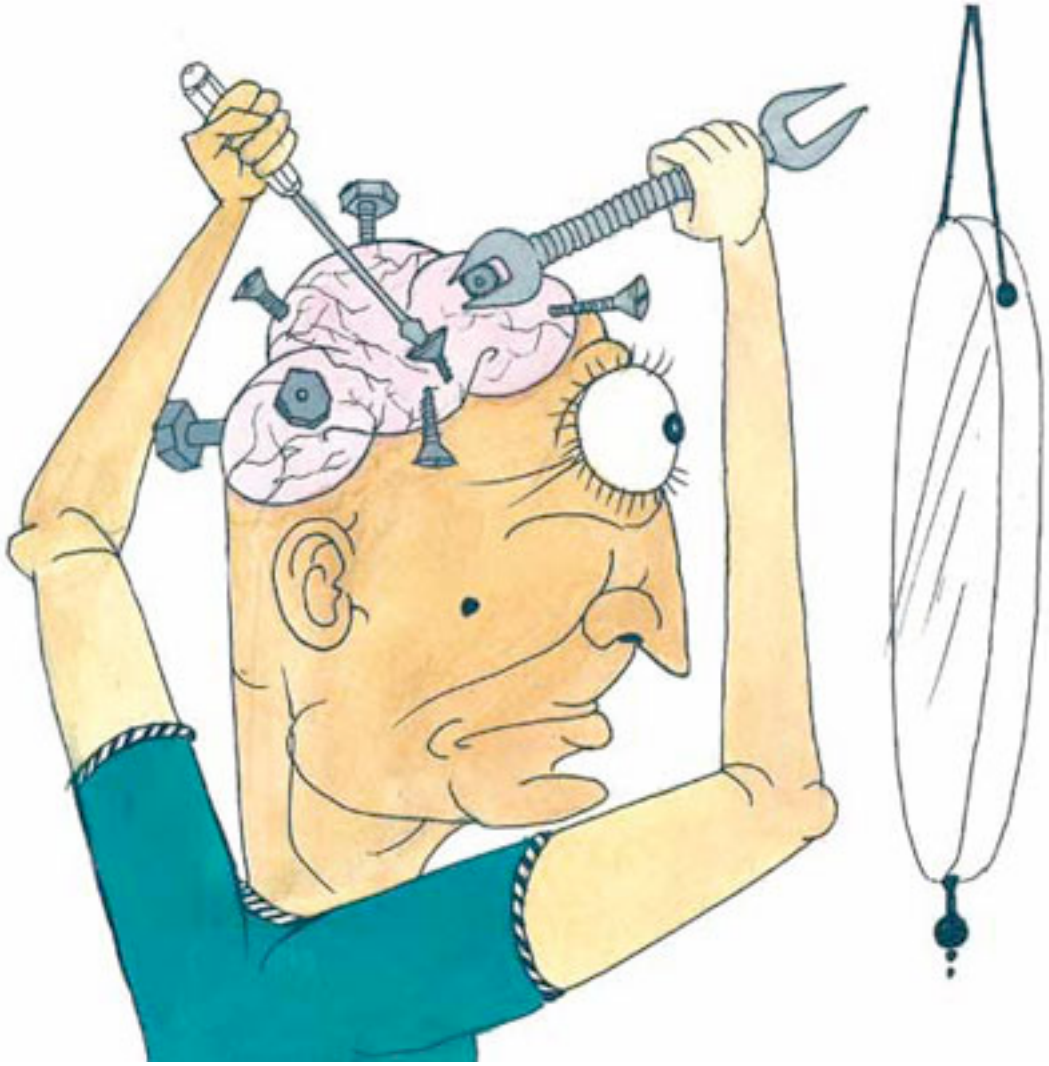
- reveals ambiguities in specifications
- reveals freedom in specifications

Dealing with "freedom" by defining model variations.

Models **configurable** and

instantiated by variants

Property-based testing



change your mind about testing