Experiences with QuickCheck: Testing the Hard Stuff and Staying Sane

John Hughes

Chalmers University/Quviq AB, Göteborg, Sweden

Abstract. This is not a typical scientific paper. It does not present a new method, with careful experiments to evaluate it, and detailed references to related work. Rather, it recounts some of my experiences over the last 15 years, working with QuickCheck, and its purpose is as much to entertain as to inform.
QuickCheck is a random testing tool that Koen Claessen and I invented, which has since become the testing tool of choice in the Haskell community. In 2006 I co-founded Quviq, to develop and market an Erlang version, which we have since applied for a wide variety of customers, encountering many fascinating testing problems as a result.
This paper introduces Quviq QuickCheck, and in particular the extensions made for testing stateful code, via a toy example in C. It goes on to describe the largest QuickCheck project to date, which developed acceptance tests for AUTOSAR C code on behalf of Volvo Cars. Finally it explains a race detection method that nailed a notorious bug plaguing Klarna, northern Europe's market leader in invoicing systems for e-commerce. Together, these examples give a reasonable overview of the way QuickCheck has been used in industrial practice.

1 Introduction

Here's an interesting little experiment. Try asking a room full of software developers, "who *really, really* loves testing?" Very, very few will raise their hands. For whatever reason, most developers see testing as more of a chore than a pleasure; few go to work in the morning raring to write some test cases. And yet, testing is a vital part of software development! Why should it be so unpopular?

To understand why, imagine writing a suite of unit tests for software with, say, n different features. Probably you will write 3–4 test cases per feature. This is perfectly manageable—it's a linear amount of work. But, we all know you will not find all of your bugs that way, because some bugs can only be triggered by a pair of features interacting. Now, you *could* go on to write test cases for every pair of features—but this is a quadratic amount of work, which is much less appealing. And even if you do so, you will still not find all of your bugs—some bugs only appear when *three* features interact! Testing for all of these would involve $O(n^3)$ work, which is starting to sound very unappealing indeed—and this is before we even start to consider race conditions, which by definition involve at least two

```
#include <stdlib.h>

typedef struct queue
{ int *buf;
  int inp, outp, size;
} Queue;

Queue *new(int n)
{ int *buff =
    malloc(n*sizeof(int));
  Queue q = {buff,0,0,n};
  Queue *qptr =
    malloc(sizeof(Queue));
  *qptr = q;
  return qptr; }
```

```
void put(Queue *q, int n)
{ q -> buf[q -> inp] = n;
  q -> inp = (q -> inp + 1)
               % q -> size; }

int get(Queue *q)
{ int ans = q -> buf[q -> outp];
  q -> outp = (q -> outp + 1)
                % q -> size;
  return ans; }

int size(Queue *q)
{ return (q->inp - q->outp)
           % q -> size; }
```

Fig. 1. A queue ADT, implemented as a circular buffer in C.

features interacting, and even worse, only manifest themselves occasionally even in test cases that can provoke them in principle!

This is the fundamental problem with testing—you can never be "done". No wonder it's not so popular. What is the answer to this conundrum? Simply this:

DON'T WRITE TESTS!

But of course, we can't just deliver untested code to users. So the message is actually more nuanced: don't *write* tests—*generate* them!

This is how I have spent much of my time in recent years, working with a test case generator called QuickCheck. QuickCheck was first developed in Haskell by Koen Claessen and myself (Claessen and Hughes, 2000), and has become the testing tool of choice among Haskell developers. The idea has been widely emulated—Wikipedia now lists no fewer than 35 reimplementations of the basic idea, for languages ranging from C to Swift. Thomas Arts and I founded a company, Quviq, to market an Erlang version in 2006 (Hughes, 2007), and since then we have made many extensions, and had great fun finding bugs for Ericsson (Arts, Hughes, Johansson, and Wiger, 2006), Volvo Cars (Arts, Hughes, Norell, and Svensson, 2015), and many others. In this paper I will sketch what Quviq QuickCheck does, and tell a few stories of its applications over the years.

2 A Simple Example

Let us begin with a simple example: a queue, implemented as a circular buffer, in C. The code appears in Figure 1. It declares a struct to represent queues, containing a pointer to a buffer area holding integers, the indices in the buffer where the next value should be inserted respectively removed, and the total size of the buffer. The `new` function, which creates a queue of size `n`, allocates memory for the struct and the buffer, initializes the struct, and returns a pointer to it. `get` and `put` read and write the buffer respectively at the appropriate

index, incrementing the index afterwards modulo the size. Finally, `size` returns the number of elements currently in the queue, by taking the difference of the input and the output indices, modulo the size. The code is straightforward, and obviously correct—it's just a simple example to give us something to test.

Quviq QuickCheck provides a mechanism for testing C code directly from Erlang. In the Erlang shell, we can call

```
1> eqc_c:start(q).
```

which compiles `q.c` (containing the code from Figure 1), and makes the functions in it callable from Erlang. This is done by parsing the C code to extract the types and functions it contains, generating a server that calls the C functions on demand, generating client code in Erlang that passes a function and arguments to the server and collects the result, and then running the server in a separate OS process so that buggy C code cannot interfere with the OS process running QuickCheck. The result is a safe test environment for C functions that might behave arbitrarily badly.

We can now perform simple tests to verify that the code is working as it should:

```
2> Q = q:new(5).
{ptr,"Queue",6696712}
3> q:put(Q,1).
ok
```

```
4> q:put(Q,2).
ok
5> q:size(Q).
2
```

```
6> q:get(Q).
1
7> q:get(Q).
2
```

We create a queue with space for 5 elements, binding the pointer returned to `Q`. Then we put 1 and 2 into the queue, test `size`, and take the elements out of the queue again. All the results are as expected. We can even continue the test:

```
8> q:get(Q).
100663302
9> q:get(Q).
27452
```

```
10> q:get(Q).
6696824
11> q:get(Q).
1
```

```
12> q:get(Q).
2
```

which returns the contents of uninitialized memory, as expected, until we reach the values 1 and 2 again. We really are running C code, which is just as unsafe as ever, and the queue implementation does not—and is not intended to—perform any error checking. If you abuse it by removing elements from an empty queue, you will get exactly what you deserve.

Testing with QuickCheck

Like much of the code Quviq works with, this example is stateful. We test stateful systems by generating *sequences* of calls to the API under test, just like the test cases that developers write by hand. But we also *model* the state of the system abstractly, and define *model state transitions* for each operation in the API. Using these state transition functions, we compute the model state at every point in a test. We define *postconditions* for each API call, relating the *actual* result of the call to the *model* state—see Figure 2. A test passes if all postconditions hold.

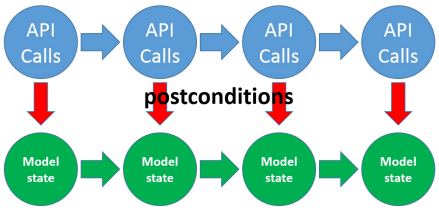

Fig. 2. Adjudging a test based on a state machine model.

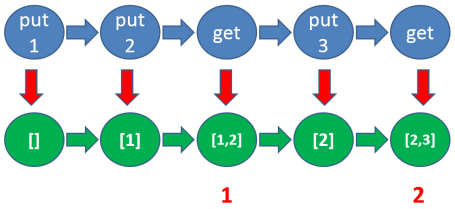

Fig. 3. A sample test case for the queue.

While the code we test can be in any programming language (such as C), the *model*—the state transition functions, postconditions, *etc.*—is defined in Erlang.

In this case, we might model the state of the queue by a *list* of the integers that should be in it; with this model, a sample test case might appear as in Figure 3. Here we start with an empty queue (modelled by the empty list `[]`), each `put` operation appends an element to the model state, each `get` operation removes an element from the state, and `get`'s postcondition checks that the actual result (in red) was the first element of the model state before the call.

We will not present all the code of the model here, but to give its flavour, we present the specification of `get`:

```
get_pre(S) -> S#state.ptr /= undefined andalso
                S#state.contents /= [].

get_next(S,_Value,_Args) ->
              S#state{contents=tl(S#state.contents)}.

get_post(S,_Args,Res) -> eq(Res,hd(S#state.contents)).
```

Here the model state `S` is actually an Erlang record (`state`) with fields `ptr` (pointer to the queue), `contents` (expected list of elements), and `size` (maximum size). `get_pre` is the precondition: `get` may only be called if the `ptr` is defined (*i.e.* `new` has been called), and the queue is non-empty. `get_next` is the state transition function: it replaces the list of elements in the model state by its tail. `get_post` is the postcondition, which checks that the actual result of `get`, `Res`, is equal to the first element of the expected contents. QuickCheck recognises these functions by virtue of their names (`get` followed by a particular suffix), and uses them to generate and run random tests. The property tested is: if all operation preconditions hold during a test run, then so do the postconditions.

Tests are run by invoking QuickCheck from the Erlang shell, giving this property as a parameter. In this case, the tests fail almost immediately:

```
18> eqc:quickcheck(q_eqc:prop_q()).
....Failed! After 5 tests.
[{set,{var,1},{call,q,new,[1]}},
 {set,{var,2},{call,q,put,[{var,1},1]}},
...10 lines of output elided...

q:new(1) -> {ptr, "Queue", 4003800}
q:put({ptr, "Queue", 4003800}, 1) -> ok
q:get({ptr, "Queue", 4003800}) -> 1
q:put({ptr, "Queue", 4003800}, -1) -> ok
q:put({ptr, "Queue", 4003800}, -1) -> ok
q:put({ptr, "Queue", 4003800}, 0) -> ok
q:get({ptr, "Queue", 4003800}) -> 0
Reason: Post-condition failed: 0 /= -1
```

After four successful tests (represented by a '`.`' in the output), QuickCheck generated a failing example, which appears next in a symbolic form—but in this paper, we focus on the pretty-printed results that follow it (beginning with the line `q:new(1) ->`...). These show the calls made and results returned—and clearly, something is wrong, because the postcondition of the last call failed. 0, the *actual* result of the call, was not equal to -1, the *expected* value.

Like most randomly generated failing tests, this one contains irrelevant calls as well as those few that actually provoked the failure. So QuickCheck then *shrinks* the test case to a simpler one, searching for the *smallest similar test* that also fails. In this case,the output continues:

```
Shrinking xxxx..xx.xx.xxxx.xxx(5 times)
[{set,{var,1},{call,q,new,[1]}},
 {set,{var,2},{call,q,put,[{var,1},1]}},
 {set,{var,3},{call,q,put,[{var,1},0]}},
 {set,{var,4},{call,q,get,[{var,1}]}}]

q:new(1) -> {ptr, "Queue", 4009248}
q:put({ptr, "Queue", 4009248}, 1) -> ok
q:put({ptr, "Queue", 4009248}, 0) -> ok
q:get({ptr, "Queue", 4009248}) -> 0
Reason: Post-condition failed: 0 /= 1
```

During shrinking, each '`.`' represents a smaller test that also failed—progress towards the minimal failing test—while '`x`' represents a smaller test that passed. Shrinking not only removes unnecessary *calls* from the test case, but also simplifies *arguments* where possible—in particular, -1 no longer appears in the test.

While randomly generated failing tests vary widely, the result after shrinking is very consistent—always either the test case above, or this very similar one:

```
q:new(1) -> {ptr, "Queue", 4027504}
q:put({ptr, "Queue", 4027504}, 0) -> ok
q:put({ptr, "Queue", 4027504}, 1) -> ok
```

```
q:get({ptr, "Queue", 4027504}) -> 1
Reason: Post-condition failed: 1 /= 0
```

Debugging the model

But why does the test above fail? Inspecting the results, we see that first we create a queue with space for one element, then we put a 0 into it, then we put a 1 into it—and at this point, the queue should contain $[0, 1]$—then finally we get the first element, which should be 0, but was actually 1! Clearly this is wrong!

But *why* did the C code return 1? Tracing through the example, we see that *we allocated a buffer large enough to hold one integer*, and then put two integers into it! Since we implemented a circular buffer, then the second value (1) overwrites the first (0). This is why `get` returns 1—the data has been corrupted by putting too many elements into the queue.

This really is a *minimal* test case for this error: not only *must* we perform two `put`s and a `get`, but we *must* put two different values—otherwise we overwrite a value with an equal one, and the error is not detectable. This is why the values put are 0 and 1—one of them is arbitrary, so can shrink to 0, while the other *must be different*. The simplest value different from 0 is 1, so this is the value we see. In retrospect, it was very informative that we did not see *two* zeroes—this told us immediately that if we were to put two zeroes, then the test would pass.

So where is the error? Recall that the code is not *intended* to behave sensibly when abused—and putting two elements into a queue of size one is surely abuse. Arguably, there is nothing wrong with the code—this is *a bad test*. This means that the fault is not in the implementation, but in the *model*. Inspecting the *precondition* of `put` in the model we have been using so far, we find

```
put_pre(S) -> S#state.ptr /= undefined.
```

This allows tests to call `put` at any time, as long as the queue pointer is defined. We should of course only allow calls of `put` if the queue is not already full:

```
put_pre(S) -> S#state.ptr /= undefined andalso
                length(S#state.contents) < S#state.size.
```

With this change to the precondition, we can repeat the last failed test, and see

```
q:new(1) -> {ptr, "Queue", 4027648}
q:put({ptr, "Queue", 4027648}, 0) -> ok
q:put({ptr, "Queue", 4027648}, 1) -> !!! precondition_failed
Reason: precondition_failed
```

It still fails, but at an earlier point and for a different reason—the second call to `put` can no longer be made, because its precondition is not satisfied.

Random tests now pass, and we see satisfying output from QuickCheck:

```
28> eqc:quickcheck(q_eqc:prop_q()).
.....................................................
.....................................
OK, passed 100 tests
53.5% {q,put,2}
```

```
40.2% {q,get,1}
6.3% {q,new,1}
```

We usually collect statistics as tests are run; in this case we collect the names of the functions called ({q,new,1} represents q:new/1, the new function with 1 argument in module q, and so on). Over 50% of the calls tested were to put, around 40% were to get—which is not so surprising, since given our preconditions, we cannot call get unless there is a corresponding call to put. However, these statistics reveal that we have *not* tested the size function at all! There is a simple reason for this—for simplicity, we omitted it from the model.

Debugging the code

It is easy to model the behaviour of size as well, but with this extension to the model, tests fail immediately. The shrunk example is this:

```
q:new(1) -> {ptr, "Queue", 4033488}
q:put({ptr, "Queue", 4033488}, 0) -> ok
q:size({ptr, "Queue", 4033488}) -> 0
Reason: Post-condition failed: 0 /= 1
```

We create a queue with room for one element, put an element into it, then ask how many there are—which of course should be 1, but size returns 0! This time the model is not at fault—the C code is simply returning the wrong answer.

Using this example we can trace through the code in Figure 1 to find the problem. new(1) initializes q->size to 1. size returns

```
(q->inp - q->outp) % q->size
```

But q->size is 1, and any value modulo 1 is zero! It follows that size *always* returns 0 when the queue size is 1, no matter how many times we call put. Likewise, put and get increment their respective indices *modulo* 1, so these indices are also always zero. Putting an element into the queue does not change its state—a full queue looks exactly the same as an empty one! This is the real problem, and it is not limited to queues of size one. If we create a queue of size n, and put n items into it, then the same thing happens—the input index wraps around back to zero, and a full queue looks the same as an empty one. Our *representation* of queues is inadequate—it cannot distinguish full from empty.

How to fix this? There are several possibilities—we could, for example, add a boolean to the Queue struct to distinguish full queues, but this would lead to special cases throughout the code. Much nicer is to modify the new function so that, when asked to create a queue with space for n elements, we actually create one with space for $n + 1$ elements!

```
Queue *new(int n)
{ int *buff = malloc((n+1)*sizeof(int));
  Queue q = {buff,0,0,n+1};
  Queue *qptr = malloc(sizeof(Queue));
  *qptr = q;
  return qptr;
}
```

Let this be our secret. Now, the only way for a caller to witness the 'bug' is by putting $n+1$ elements into a queue with room for n—violating the precondition of `put`, which means *the problem is their fault!* We successfully avoid blame, and can go home happy.

Sure enough, with this change then repeating the last test succeeds:

```
32> eqc:check(q_eqc:prop_q()).
OK, passed the test.
```

However, running random tests provokes another failure, which always shrinks to the same test case:

```
q:new(1) -> {ptr, "Queue", 5844232}
q:put({ptr, "Queue", 5844232}, 0) -> ok
q:get({ptr, "Queue", 5844232}) -> 0
q:put({ptr, "Queue", 5844232}, 0) -> ok
q:size({ptr, "Queue", 5844232}) -> -1
Reason: Post-condition failed: -1 /= 1
```

We create a queue of size one (that's really two!), and put a zero into it—now it's full. We get the zero—and now it's empty, so we can put another zero in. Now the queue should contain one zero... but when we ask the `size`, then -1 is returned! This is clearly a bug in the C code—the size should never be negative.

Tracing this example through the code in Figure 1, we see that we really created a queue with a `q->size` of 2, then called `put` twice, so the input index `q->inp` wraps around back to zero. We called `get` once, so `q->outp` is 1, and in `size`, `(q->inp - q->outp) % q->size` computes `(-1) % 2`—but $(-1) \bmod 2$ is $+1$, as we all learned in school. Or is it? Trying it in Erlang:

```
34> (-1) rem 2.
-1
```

In Erlang, at least, `rem` is *not* the modulo operator—it computes the remainder after integer division, *rounding towards zero*. Since `(-1) div 2` is zero, then `(-1) rem 2` must be -1. The same is true in C.

With this in mind, we see that `size` returns a negative result whenever `q->inp` has wrapped around, but `q->outp` has not—whenever

```
  q->inp - q->outp
```

is negative. To fix the bug, we must ensure that this expression *is* never negative, and a simple way to do so is to apply `abs` to it. We modify `size` as follows,

```
int size(Queue *q)
{ return abs(q->inp - q->outp) % q -> size;
}
```

and repeat the last test:

```
36> eqc:check(q_eqc:prop_q()).
OK, passed the test.
```

Of course it passes, as we would expect.

Now, notice what we did. We found a subtle bug in our implementation. We created a test case that provoked the bug. We fixed the code, and our test now passes. All our tests are green! So... we're done, right?

In practice, many developers would be satisfied at this point. But let us just run a few more random tests... we find they fail immediately!

```
q:new(2) -> {ptr, "Queue", 6774864}
q:put({ptr, "Queue", 6774864}, 0) -> ok
q:put({ptr, "Queue", 6774864}, 0) -> ok
q:get({ptr, "Queue", 6774864}) -> 0
q:put({ptr, "Queue", 6774864}, 0) -> ok
q:size({ptr, "Queue", 6774864}) -> 1
Reason: Post-condition failed: 1 /= 2
```

What's going on here? First, we create a queue of size two. Progress! The code now works—for queues of size one. We put *three* items into the queue, and remove one. Now there should be two items in the queue, but `size` returns 1.

This is actually very similar to the example we just fixed. The queue is size two (really three!), so calling `put` three times makes the input index wrap around back to zero. We called `get` once, so the output index is one. The correct answer for `size` is thus $(-1) \bmod 3$, which is 2, but we computed `abs(-1) % 2`, which is `1`. Taking the absolute value was the wrong thing to do; it worked for the first test case we considered, but not in general.

A correct implementation must ensure that the operand of `%` is non-negative, *without* changing its value modulo `q->size`. We can do this by *adding* `q->size` to it, instead of taking the absolute value:

```
int size(Queue *q)
{ return (q->inp - q->outp + q->size) % q -> size;
}
```

When we make this change, then this last failing test—and all the others we can generate—pass. The code is correct at last.

Lessons from the queue example

What can we learn from this? Three lessons that apply in general:

- The *same* property can find many *different* bugs. This is the whole point of test case generation: one model can provoke a wide variety of bugs.
- Errors are often in the model, rather than the code. Calling something a specification does not make it right! QuickCheck finds *discrepancies* between the model and the code; it is up to the developer to decide which is correct.
- *Minimal failing test cases* make debugging easy. They are often much smaller than the first failing one found. We think of shrinking as "extracting the signal from the noise". Random tests contain a great deal of junk—that is their purpose! Junk provokes unexpected behaviour and tests scenarios that the developer would never think of. But tests usually *fail* because of just a few features of the test case. Debugging a test that is 90% irrelevant is a

nightmare; presenting the developer with a test where *every part* is known to be relevant to the failure, simplifies the debugging task enormously.

Property-based testing

I referred several times above to the "property" that is being tested, namely that "*if all operation preconditions hold during a test run, then so do the postconditions*". Properties are fundamental to QuickCheck: indeed, QuickCheck *always* tests a property, by generating suitable test cases and checking that the property holds in each case. Properties are often expressed in terms of a stateful model—as in this section—but this is not always the case, and even for the same model, the properties tested may differ. Since properties are fundamental here, we call the general approach "*property-based testing*", rather than "model-based testing" which is a better-established term.

3 The Volvo Project

So far, we only considered a toy example—the reader could be forgiven for wondering if the method really scales. After all, in this example, the specification was actually larger than the code! QuickCheck would not be useful if this were true in general. So, let us summarize the results of the largest QuickCheck project so far—acceptance testing of AUTOSAR Basic Software for Volvo Cars (Arts, Hughes, Norell, and Svensson, 2015).

Modern cars contain 50–100 processors, and many manufacturers base their software on the AUTOSAR standard. Among other things, this defines the 'Basic Software', which runs on *every* processor in the car. It consists of an Ethernet stack, a CAN-bus stack (the CAN bus, or "Controller Area Network", is heavily used in vehicles), a couple of other protocol stacks, a routing component, and a diagnostics cluster (which records fault codes as you drive). The basic software is made up of many modules, whose behaviour is *specified* in detail, but there are a number of suppliers offering competing implementations. Usually a car integrates basic software from several different suppliers—which means that if they do *not* follow the standard, then there is a risk of system integration problems. It is not even easy to tell, if two processors cannot talk to each other, *which* of the suppliers is at fault!

Volvo Cars wanted to *test* their suppliers code for conformance with the standard, and funded Quviq to develop QuickCheck models for this purpose.

Most errors we found are confidential, but we can describe one involving the CAN bus stack, which we tested using a mocked hardware bus driver. Now, every message sent on the CAN bus has a *CAN identifier*, or message type, which also serves as the *priority* of the message—the smaller the CAN identifier, the higher the priority. QuickCheck found a failing test with the following form:

- Send a message with CAN identifier 1 (and check that the hardware driver is invoked to put it onto the bus).
- Send a message with CAN identifier 2 (which should *not* be passed to the driver yet, because the bus is busy).

– Send a message with CAN identifier 3 (which should also not be sent yet).
– Confirm transmission of message 1 by the bus (and check that the driver is now invoked to send message 2).

Of course, the bug was that the stack sent the message with identifier 3 instead.

This *is* the smallest test case that can provoke the bug. We must first send a message, making the bus busy, and then send two more with different priorities, which are both queued. Then we must release the bus, and observe that the wrong message is taken from the queue. This is just what the test does.

The source of the bug was this: the original CAN standard allowed only 11 bits for the message identifier, but today 2048 different message types is no longer enough, so the current standard *also* allows an 'extended CAN id' of 29 bits. Both kinds have the same meaning—in particular, the priority does not depend on the encoding—but when the stack sends a message, it must know which format to use. This particular stack stored both kinds of identifier in the same unsigned integer, but used bit 31 to indicate an extended identifier. In this test, message number 2 used an extended CAN identifier, while message number 3 used the original format. Of course, when comparing stored message identifiers in order to decide which message to send, it is essential to mask off the top bit... which the developer forgot to do in this case. So the second message was treated as priority $2^{31} + 2$, and was not chosen for sending next.

The actual fault is a tricky mistake in low-level C code—nevertheless, *it can be provoked with a short sequence of API calls*, and *we can find this sequence by generating a random one and shrinking it.*

The fault is also potentially serious. Those priorities are there for a reason. Almost everything in the car can talk on the CAN bus—the brakes, the stereo.... Here's a tip: don't adjust the volume during emergency braking!

For this project we read around 3,000 pages of PDFs (the standards documents). We formalized the specification in 20,000 lines of QuickCheck code. We used it to test a million lines of C code in total, from 6 different suppliers, finding more than 200 problems—of which well over 100 were ambiguities or inconsistencies in the standard itself! In the few cases where we could compare our test code to traditional test suites (in TTCN3), our code was an order of magnitude smaller—but tests more.

So, does the method scale? Yes, it scales.

4 Debugging a database

The last story (Hughes and Bolinder, 2011) begins with a message to the Erlang mailing list, from Tobbe Törnqvist at Klarna.

> *"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."*

The context: Klarna was a Swedish start-up offering invoicing services to web shops[1]. They built their product in Erlang, and stored their data in Mnesia, the database packaged with the Erlang distribution. Mnesia provides transactions, distribution, replication, and so on, but needs a back end to actually store the data. The back end for storage on disk is *dets*, which stores tuples in files. This problem piqued my interest—it was quite notorious in the Erlang community, and sounded as though it might involve a race condition.

To explain how QuickCheck tests for race conditions, let us consider a much simpler example: a ticket dispenser such as one finds at delicatessen counters, which issues each customer with a ticket indicating their place in the queue. We might model such a dispenser in Erlang by two functions, `take_ticket()` and—because the roll of tickets needs to be replaced sometimes—`reset()`. A simple unit test for these functions might look as follows:

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

Here we use Erlang pattern matching to compare the result of each call with its *expected value* (`ok` being an atom, a constant, in Erlang—conventionally used as the result of a function with no interesting result). This test case is typical of test cases written in industry: we make a sequence of calls, and compare the results to expected values.

It is easy to create a QuickCheck state machine for testing the dispenser also. We can model its state as a single integer, which is set to 0 by `reset()`, incremented by `take_ticket()`, and used in the postcondition of `take_ticket` to check the result.

However, running *sequential* tests of the ticket dispenser is not enough: the whole point of the system is to regulate the flow of many concurrent customers to one delicatessen counter. If we do not also run parallel tests of the API, then we are failing to test an essential part of its behaviour.

Now, consider one simple parallel test:

`reset()`	
`take_ticket()`	`take_ticket()`
`take_ticket()`	

which represents first resetting the dispenser, then *two* customers taking tickets in parallel—the left customer taking two tickets, the right customer taking one. *What are the expected results?* Well, the left customer might get tickets #1 and #2, and the right customer ticket #3. Or, the left customer might get tickets #1 and #3, while the right customer gets ticket #2! But if *both* customers get ticket number #1, then the test has failed.

[1] They have since grown to well over 1,000 people, serving 8 countries and counting.

This poses a problem for the traditional way of writing test cases, because *we cannot predict the expected results*. In this case, we could in principle compute all three possibilities (in which the right customer gets ticket number #1, #2 or #3), and compare the actual results against all three—but this approach does not scale at all. Consider this only slightly larger test:

`reset()`		
`take_ticket()`	`take_ticket()`	`reset()`
`take_ticket()`	`take_ticket()`	

This test has *thirty* possible correct outcomes! No developer in their right mind would try to enumerate all of these—and anyone who tried would surely get at least one wrong. The *only* practical way to decide if a test such as this has passed or failed, is via a *property* that distinguishes correct from wrong results. Parallel testing is a 'killer app' for property-based testing.

The way that QuickCheck decides whether a test like this has passed is by searching for *any interleaving* of the calls which makes their results match the model. If there is such an interleaving, then the test passes—but if there is *no* such interleaving, then it has definitely failed. Here we are using the *same* model as we used for sequential testing, to test a *different property* of the implementation—namely, serializability of the operations.

Testing a buggy version of the dispenser, QuickCheck immediately reports:

```
Parallel:
1. dispenser:take_ticket() --> 1
2. dispenser:take_ticket() --> 1
Result: no_possible_interleaving
```

That is, two parallel calls of `take_ticket` can both return 1—but no *sequence* of two calls can return these results. In this case, there is a blatant bug in the code: `take_ticket()` reads and then writes a global variable containing the next ticket number, with no synchronization at all—so no wonder there is a race condition. But the point is that QuickCheck *finds* the bug very fast, and *shrinks* it to this minimal example very reliably indeed.

Returning to dets, it stores tuples of the form $\{\texttt{Key},\texttt{Val}_1,\texttt{Val}_2\ldots\}$ in a file, and its API is unsurprising. For example,

- `insert(Table,List)` inserts a list of tuples into a table,
- `delete(Table,Key)` deletes all tuples with the given key from the table, and
- `insert_new(Table,List)` is like `insert`, unless one of the keys to be inserted is already in the table, in which case it is a no-op.

These operations are all guaranteed to be atomic.

A state machine modelling dets is easy to construct—it is almost enough just to model the table by a list of the tuples that should be in it. My model of the core of the dets API is less than 100 lines of code—which compares well to the implementation, over 6,000 lines of code spread over four modules, maintaining hash tables on the disk, supporting an old format for the files, and so on. Thus, although dets is quite complex, its intended behaviour is simple.

I first tested the model thoroughly by running tens of thousands of sequential tests. This ensures that the model and the code are consistent—and it did turn up a few surprises. There is no point running parallel tests of a system, if inconsistencies can already be found using sequential tests. (I assumed that dets behaves correctly in the sequential case—it is mature and well-tested code, after all—so any inconsistencies indicate a misunderstanding of the intended behaviour, *i.e.* a bug in the model).

Once the model was correct, I began to run parallel tests. This only required writing a few lines of code, because we reuse the *same* model for sequential and parallel testing. Almost immediately, a real bug appeared:

```
Prefix:
   open_file(dets_table,[{type,bag}]) --> dets_table
Parallel:
1. insert(dets_table,[]) --> ok
2. insert_new(dets_table,[]) --> ok
Result: no_possible_interleaving
```

In the sequential prefix of the test, we open the dets file, then in parallel call `insert` and `insert_new`. Both insert an *empty* list of tuples—so they are both no-ops—and they both return `ok`! At first sight this looks reasonable—but the documentation for `insert_new` says it should return a boolean! Indeed, in tens of thousands of sequential tests, `insert_new` returned `true` or `false` every time—now, suddenly, it returned `ok`. Strange!

Another run of QuickCheck reported a second bug:

```
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...
```

In this case, both `insert` and `insert_new` try to insert the same tuple, which means that `insert_new` could succeed or fail... but it should *not* time out, and we should *not* see the message

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

At this point I disabled the testing of `insert_new`, which seemed not work in parallel, and discovered a third bug:

```
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
```

Here we first open the file, and then *in parallel* open it again, and insert a tuple, then fetch the entire contents of the table. The test fails because process

2 sees an empty table, *even though it just inserted a tuple*! It may seem a little suspicious to open the table twice, but it is not at all—Erlang is designed for highly concurrent applications, so we *expect* many processes to be using the table at the same time, and all of them need to make sure it is open.

I reported these bugs to Hans Bolinder at Ericsson, who was responsible for the dets code, and next day he sent me a thank you and a fixed version. However, he thought these were probably not the bugs that were plaguing Klarna, because the symptoms were different. At Klarna, the file was being corrupted. Hans gave me one line of code that could check for corruption, and I added it to the test to check that, after executing a parallel test, the file was not corrupt.

This time it took around 10 minutes of testing to find a bug:

```
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
```

First we open, close, and open the file again, and then we do *three* things in parallel: a `lookup` of a key that is not found, and two insertions of the same tuple. All results are consistent with the model—but the corruption check encountered a 'premature eof'.

This is really the *smallest test case* that can provoke this error. It was initially hard to believe that the open–close–open sequence was really necessary. I manually removed the first open and close, and ran the smaller test case tens of thousands of times. It passed, every single time. Today I know why: the first call to open *creates* the file, while the second call opens an *existing* file. It's slightly different, and dets enters a slightly different state—and that state is key to provoking the bug. Three parallel operations are needed because the first makes the dets server busy, causing the next two to be queued up, and then (wrongly) dispatched in parallel by the server, once the first call is complete.

The final bug was found when preparing a talk on this experience soon afterwards, whlie rerunning the tests in order to copy the output onto slides. Again, after around ten minutes of testing, the following bug was found:

```
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
```

We open the file and insert a tuple, then, in parallel, *reopen* the file, and lookup a key not in the table, then delete the key that is. All the results are consistent

with the model—but the corruption check encountered a 'bad object'. Recall the mailing list message:

> *"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."*

So it seemed these might indeed be the Klarna bugs.

In each case, Hans Bolinder returned a fixed version of the code within a day of receiving the bug report—and at Klarna, where the problem had meanwhile begun to appear once a week, there has been only *one* 'bad object' error since the new code went into production... when a reading a file last written *before* the new code was installed.

Prior to this work Hans Bolinder had spent six weeks at Klarna hunting for the bug, and the best theory was that it struck when the file was around 1GB, and might be something to do with rehashing. Now we know the bugs could be provoked with a database with at most one record, using only 5–6 calls to the API. In each case, given the minimal failing test, it was less than a day's work to find and fix the bug. This really demonstrates the enormous value of shrinking—and shows just how hopeless it is to try to debug this kind of problem from failures that happen in production, where not only the five or six relevant things have occurred, but also millions of irrelevant others.

5 In conclusion

This paper recounts some of *our* experiences of using an advanced testing tool in real applications—its purpose is as much to entertain as to inform. Of course, there is a wealth of other work in the area, that this paper makes no attempt to cover—RANDOOP (Pacheco, Lahiri, Ernst, and Ball, 2007), DART (Godefroid, Klarlund, and Sen, 2005), and their successors would be good starting points for further reading.

I have also just touched the surface of the work that we have done. There was far more to the Volvo project than could be described here. We needed to develop a property-based approach to mocking (Svenningsson, Svensson, Smallbone, Arts, Norell, and Hughes, 2014). We needed to develop ways to *cluster* models of individual components into an integrated model of a subsystem, because AUTOSAR subsystems are typically *implemented* monolithically, but *specified* as a collection of interacting components. We found that QuickCheck's shrinking tended to isolate the *same* bug in each run—which is not ideal when generating a test report on a component with *several different* bugs—so we developed methods to direct testing away from already-known problems towards areas where unknown bugs might still lurk, and report *all* the bugs at once. We were asked "does your model cover this test case?", so we developed tools to answer such questions—and when the AUTOSAR consortium released six "official" test cases for the CAN bus stack, we used them to show that three of these test cases were *not* covered by the model, because they were not consistent with the standard (Gerdes, Hughes, Smallbone, and Wang, 2015). We were asked: "which *requirements* have you tested?", so we developed ways to collect requirements

coverage during testing—leading us to question what testing a requirement even *means* (Arts and Hughes, 2016).

In other areas, we helped Basho test their no-SQL database, Riak, for the key property of *eventual consistency*—and found a bug (now fixed, of course) that was present, not only in Riak, but in the original Amazon paper (DeCandia, Hastorun, Jampani, Kakulapati, Lakshman, Pilchin, Sivasubramanian, Vosshall, and Vogels, 2007) that kicked off the no-SQL trend. Recently we tested Dropbox's file synchronization service, with some surprising results there too (Hughes, Pierce, Arts, and Norell, 2016). Both these applications were based in part on the state machine framework presented here, but added additional *properties* to express subtle correctness conditions.

Perhaps the most general lesson from all this experience is that formulating specifications is *hard*—and many developers struggle with it. It may often be easier to give *examples* of correct behaviour, than to define *in general* what correctness means. This should not come as a surprise, since mathematicians have used examples for centuries—but it does suggest that starting from a specification may not always be appropriate. Sometimes I am asked: "wouldn't it be great if we could just synthesize code from a specification?" It would not: we would just replace buggy code with buggy specifications. A key insight from using QuickCheck is that we find bugs by comparing *two independent descriptions* of the desired behaviour—the implementation, and the specification—and it is the inconsistencies between the two that reveal errors.

The *need for a specification* is the real weakness of property-based testing—not that we use testing, rather than static analysis or proof, to relate specifications and implementations. Testing works well enough! The real question is: how can we make specifications more useable, and easier to construct—for real systems with all their complex behaviours? There is much work to be done here—perhaps future tools will even help developers construct specifications from examples.

In any event, there are many fascinating problems waiting to be addressed. If this paper helps persuade the reader that testing is not a chore that must be done, but a fascinating research area in its own right, then it will have fulfilled its purpose.

And don't forget,

DON'T WRITE TESTS, GENERATE THEM!

Acknowledgements

The work described here was partially supported by the EU FP7 project "PROWESS", and by the Swedish Foundation for Strategic Research grant "RAWFP".

References

Arts, T., Hughes, J., Norell, U., Svensson, H.: Testing AUTOSAR software with QuickCheck. In: Software Testing, Verification and Validation Workshops

(ICSTW), 2015 IEEE Eighth International Conference on. pp. 1–4 (April 2015)

Arts, T., Hughes, J.: How well are your requirements tested? In: Briand, L., Khurshid, S. (eds.) International Conference on Software Testing, Verification and Validation (ICST). IEEE (April 2016), to appear

Arts, T., Hughes, J., Johansson, J., Wiger, U.: Testing Telecoms Software with Quviq QuickCheck. In: Proceedings of the 2006 ACM SIGPLAN Workshop on Erlang. pp. 2–10. ERLANG '06, ACM, New York, NY, USA (2006)

Claessen, K., Hughes, J.: QuickCheck: A Lightweight Tool for Random Testing of Haskell Programs. In: Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming. pp. 268–279. ICFP '00, ACM, New York, NY, USA (2000)

DeCandia, G., Hastorun, D., Jampani, M., Kakulapati, G., Lakshman, A., Pilchin, A., Sivasubramanian, S., Vosshall, P., Vogels, W.: Dynamo: Amazon's Highly Available Key-value Store. In: Proceedings of Twenty-first ACM SIGOPS Symposium on Operating Systems Principles. pp. 205–220. SOSP '07, ACM, New York, NY, USA (2007)

Gerdes, A., Hughes, J., Smallbone, N., Wang, M.: Linking Unit Tests and Properties. In: Proceedings of the 14th ACM SIGPLAN Workshop on Erlang. pp. 19–26. Erlang 2015, ACM, New York, NY, USA (2015)

Godefroid, P., Klarlund, N., Sen, K.: DART: Directed Automated Random Testing. In: Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation. pp. 213–223. PLDI '05, ACM, New York, NY, USA (2005)

Hughes, J.: QuickCheck Testing for Fun and Profit. In: Proceedings of the 9th International Conference on Practical Aspects of Declarative Languages. pp. 1–32. PADL'07, Springer-Verlag, Berlin, Heidelberg (2007)

Hughes, J., Pierce, B., Arts, T., Norell, U.: Mysteries of Dropbox: Property-Based Testing of a Distributed Synchronization Service. In: Briand, L., Khurshid, S. (eds.) International Conference on Software Testing, Verification and Validation (ICST). IEEE (April 2016), to appear

Hughes, J.M., Bolinder, H.: Testing a Database for Race Conditions with QuickCheck. In: Proceedings of the 10th ACM SIGPLAN Workshop on Erlang. pp. 72–77. Erlang '11, ACM, New York, NY, USA (2011)

Pacheco, C., Lahiri, S.K., Ernst, M.D., Ball, T.: Feedback-Directed Random Test Generation. In: Proceedings of the 29th International Conference on Software Engineering. pp. 75–84. ICSE '07, IEEE Computer Society, Washington, DC, USA (2007)

Svenningsson, J., Svensson, H., Smallbone, N., Arts, T., Norell, U., Hughes, J.: An Expressive Semantics of Mocking. In: Proceedings of the 17th International Conference on Fundamental Approaches to Software Engineering - Volume 8411. pp. 385–399. Springer-Verlag New York, Inc., New York, NY, USA (2014)