QuickCheck Testing for Fun and Profit

John Hughes

Chalmers University of Technology,
S-41296 Gothenburg,
Sweden

1 Introduction

One of the nice things about purely functional languages is that functions often satisfy simple properties, and enjoy simple algebraic relationships. Indeed, if the functions of an API satisfy elegant laws, that in itself is a sign of a good design—the laws not only indicate conceptual simplicity, but are useful in practice for simplifying programs that use the API, by equational reasoning or otherwise. It is a comfort to us all, for example, to know that in Haskell the following law holds:

```
reverse (xs++ys) == reverse xs++reverse ys
```

where `reverse` is the list reversal function, and `++` is list append.

It is productive to formulate such laws about one's code, but there is always the risk of formulating them incorrectly. A stated law which is untrue is worse than no law at all! Ideally, of course, one should prove them, but at the very least, one should try out the law in a few cases—just to avoid stupid mistakes. We can ease that task a little bit by defining a function to test the law, given values for its free variables:

```
prop_revApp xs ys =
  reverse (xs++ys) == reverse xs++reverse ys
```

Now we can test the law just by applying `prop_revApp` to suitable pairs of lists.

Inventing such pairs of lists, and running the tests, is tedious, however. Wouldn't it be fun to have a tool that would perform that task for us? Then we could simply write laws in our programs and automatically check that they are reasonable hypotheses, at least. In 1999, Koen Claessen and I built just such a tool for Haskell, called "QuickCheck" [4,5,7,6]. Given the definition above, we need only pass `prop_revApp` to `quickCheck` to test the property in 100 random cases:

```
> quickCheck prop_revApp
Falsifiable, after 2 tests:
[1,-1]
[0]
```

Doing so exposes at once that the property is not true! The values printed are a counter-example to the claim, `[1,-1]` being the value of `xs`, and `[0]` the value of `ys`. Indeed, inspecting the property more closely, we see that `xs` and `ys` are the wrong way round in the right hand side of the law. After correcting the mistake, quickChecking the property succeeds:

```
> quickCheck prop_revApp
OK, passed 100 tests.
```

While there is no *guarantee* that the property now holds, we can be very much more confident that we did not make a stupid mistake. . . particularly after running another few thousand tests, which is the work of a few more seconds.

We wrote QuickCheck for fun, but it has turned out to be much more useful and important than we imagined at the time. This paper will describe some of the uses to which it has since been put.

2 A Simple Example: Skew Heaps

To illustrate the use of QuickCheck in program development, we shall implement *skew heaps* (a representation of priority queues), following Chris Okasaki [15]. A heap is a binary tree with labels in the nodes,

```
data Tree a = Null | Fork a (Tree a) (Tree a)
  deriving (Eq, Show)
empty = Null
```

such that the value in each node is less than any value in its subtrees:

```
invariant Null = True
invariant (Fork x l r) = smaller x l && smaller x r
smaller x Null = True
smaller x (Fork y l r) = x <= y && invariant (Fork y l r)
```

Thanks to the invariant, we can extract the minimum element (i.e. the first element in the queue) very cheaply:

```
minElem (Fork x _ _) = x
```

To make other operations on the heap cheap, we aim to keep it roughly balanced—then the cost of traversing a branch will be logarithmic in the number of elements. This is achieved in a skew heap by inserting elements into the two subtrees alternately. No extra information is needed in nodes to keep track of where to insert next: we *always* insert into the left subtree, but swap the subtrees after each insertion—*skewing* the heap—so that the next insertion chooses the other subtree.

```
insert x Null = Fork x Null Null
insert x (Fork y l r) = Fork (min x y) r (insert (max x y) l)
```

We expect that the two subtrees of a node should be "roughly balanced", but what does this mean precisely? A moment's thought suggests that the left and right subtrees should contain precisely the same number of elements after an *odd* number of insertions, but the right subtree may be one element larger than the left one after an *even* number of insertions. We conjecture that skew heaps are balanced in the following sense:

```
balanced Null = True
balanced (Fork _ l r) = (d==0 || d==1) && balanced l && balanced r
  where d = weight r - weight l

weight Null = 0
weight (Fork _ l r) = 1 + weight l + weight r
```

Now we can use QuickCheck to test our conjecture. To do so we need to generate random skew heaps. Since the only function so far that constructs skew heaps is `insert`, we can construct any reachable skew heap by choosing a random list of elements, and inserting them into the empty heap:

```
make :: [Integer] -> Tree Integer
make ns = foldl (\h n -> insert n h) empty ns
```

We can now formulate the two properties we are interested in as follows:

```
prop_invariant ns = invariant (make ns)
prop_balanced ns = balanced (make ns)
```

We gave `make` a specific type to control the generation of test data: QuickCheck generates property arguments based on the type expected, and constraining the type of `make` is a convenient way to constrain the argument types of both properties at the same time. (If we forget this, then QuickCheck cannot tell what kind of test data to generate, and an "ambiguous overloading" error is reported). Now we can invoke QuickCheck to confirm our conjecture:

```
Skew> quickCheck prop_invariant
OK, passed 100 tests.
Skew> quickCheck prop_balanced
OK, passed 100 tests.
```

We also need an operation to *delete the minimum element* from a heap. Although finding the element is easy (it is always at the root), deleting it is not, because we have to *merge* the two subtrees into one single heap.

```
deleteMin (Fork x l r) = merge l r
```

(In fact, `merge` is usually presented as part of the interface of skew heaps, even if its utility for priority queues is less obvious). If either argument is `Null`, then merge is easy to define, but how should we merge two non-empty heaps? Clearly, the root of the merged heap must contain the lesser of the root elements of `l` and `r`, but that leaves us with *three* heaps to fit into the two subtrees of the new `Fork`—`l`, `r` and `h` below—so two must be merged recursively... *but which two?*

```
merge l Null = l
merge Null r = r
merge l r | minElem l <= minElem r = join l r
          | otherwise              = join r l

join (Fork x l r) h = Fork x ...
```

The trick is to realize that the two subtrees of a node are not created equal: we ensured during insertion that the left subtree is never larger than the right one. So any recursion should be on the *left* subtree, guaranteeing that the size of the recursive argument at least halves at each call, and that the total number of calls is logarithmic in the size of the heaps. Thus we should merge `l` with `h` above, not `r`, and because merging increases the size of the heap, skew the subtrees again, so that the next merge will choose `r` instead.

```
join (Fork x l r) h = Fork x r (merge l h)
```

Is this really right? Let us test our properties again! Of course, now skew heaps can be constructed by a combination of insertions and deletions, so our method of generating random reachable heaps is no longer complete. Now we must generate heaps from a random sequence of insertions *and deletions*:

```
data Op = Insert Integer | DeleteMin
  deriving Show

make ops = foldl op Null ops
  where op h (Insert n)    = insert n h
        op Null DeleteMin = Null
        op h DeleteMin    = deleteMin h
```

One difficulty is that a *random* sequence of insertions and deletions may attempt to delete an element from an empty heap, provoking an error. There are various ways to avoid this: we could arrange not to generate such sequences in the first place, we could generate arbitrary sequences but discard the erroneous ones, or we can simply ignore any deletions that are applied to an empty heap. In the code above we chose the last alternative, because it is the simplest to implement.

Note that `make` now has a different type—it expects a list of `Op`s as its argument—and thus so do our two properties. To test them, QuickCheck needs to be able to generate values of the `Op` type, and to make that possible, we must specify a *generator* for this type.

QuickCheck generators are an abstract data type, with a rich collection of operations for constructing them. Indeed, provision of *first-class generators* is one of the main innovations in QuickCheck. We use the Haskell class system to associate generators with types, by defining instances of

```
class Arbitrary a where
  arbitrary :: Gen a
```

The `Gen` type is also a *monad*, making available the monad operations

```
return :: a -> Gen a
```

to construct a constant generator, and

```
(>>=) :: Gen a -> (a -> Gen b) -> Gen b
```

to sequence two generators—although we usually use the latter via Haskell's syntactic sugar, the `do`-notation.

So, we specify how `Op` values should be generated as follows:

```
instance Arbitrary Op where
  arbitrary =
    frequency [(2,do n <- arbitrary; return (Insert n)),
               (1,return DeleteMin)]
```

The `frequency` function combines weighted alternatives—here we generate an insertion twice as often as a deletion, since otherwise the resulting heaps would often be very small. In the first alternative, we choose an arbitrary `Integer` and generate an `Insert` containing it; in the second alternative we generate a `DeleteMin` directly.

Now we can check that any sequence of insertions and deletions preserves the heap invariant

```
Skew> quickCheck prop_invariant
OK, passed 100 tests.
```

and that skew heaps remain balanced:

```
Skew> quickCheck prop_balanced
Falsifiable, after 37 tests:
[DeleteMin,Insert (-9),Insert (-18),Insert (-14),Insert 5,
Insert (-13),Insert (-8),Insert 13,DeleteMin,DeleteMin]
```

Oh dear! Clearly, deletion does *not* preserve the balance condition. But maybe the balance condition is too strong? All we really needed above was that the *left subtree is no larger than the right*—so let's call a node "good" if that is the case.

```
good (Fork _ l r) = weight l <= weight r
```

Now, if all the nodes in a heap are good, then `insert` and `merge` will still run in logarithmic time. We can define and test the property that all nodes are good:

```
Skew> quickCheck prop_AllGood
Falsifiable, after 55 tests:
[Insert (-7),DeleteMin,Insert (-16),Insert (-14),DeleteMin,
DeleteMin,DeleteMin,Insert (-21),Insert (-8),Insert 3,
Insert (-1),Insert 1,DeleteMin,DeleteMin,Insert (-12),
Insert 17,Insert 13]
```

Oh dear dear! Evidently, skew heaps contain a mixture of good and bad nodes.

Consulting Okasaki, we find the key insight behind the efficiency of skew heaps: *although bad nodes are more costly to process, they are cheaper to construct!* Whenever we construct a bad node with a large left subtree, then *at the same time* we recurse to create an unusually *small* right subtree—so this recursion is cheaper than expected. What we lose on the swings, we regain on the roundabouts, making for logarithmic *amortized* complexity.

To formalise this argument, Okasaki introduces the notion of "credits"—each bad node carries one credit, which must be supplied when it is created, and can be consumed when it is processed.

```
credits Null = 0
credits h@(Fork _ l r) =
  credits l + credits r + if good h then 0 else 1
```

Since we cannot directly observe the cost of insertion and deletion, we define a function `cost_insert h` that returns the number of recursive calls of `insert` made when inserting into `h`, and `cost_deleteMin h`, which returns the number of calls of `join` made when deleting from `h` (definitions omitted). Now, we claim that *on average* each insertion or deletion in a heap of `n` nodes traverses only `log2 n` nodes, and creates equally many new, possibly bad nodes, so `2*log2 n` credits should suffice for each call. (The first `log2 n` credits pay for the recursion in this call, and the second `log2 n` credits pay for bad nodes in the result).

If we now specify

```
prop_cost_insert n ops =
  cost_insert h <= 2*log2 (weight h) + 1
  where h = make ops
```

then QuickCheck finds a counterexample[1], because this property only holds on average, but when we take credits into account

```
prop_cost_insert n ops =
  cost_insert h + credits (insert n h)
  <=
  2*log2 (weight h) + 1 + credits h
  where h = make ops
```

then the property passes hundreds of thousands of tests. Likewise, the property

```
prop_cost_deleteMin ops =
  h/=Null ==>
    cost_deleteMin h + credits (deleteMin h)
    <=
    2*log2 (weight h) + credits h
 where h = make ops
```

[1] Only one test case in around 3,000 is a counterexample. This is because the method we use to generate heaps produces rather few bad nodes. Counterexamples can be found more quickly by generating heaps directly, rather than via `insert` and `deleteMin`, so that the proportion of bad nodes can be increased.

succeeds (where we have used QuickCheck's *implication* operator `==>` to state a precondition that must hold in every test case, to avoid the error that would result by calling `deleteMin` on the empty heap).

Each of these properties states that the credits allocated for the operation, together with the accumulated credits in the heap, suffice both to pay for the operation itself, and for the credits retained in its result. So any sequence of insertions and deletions, starting with the empty heap, will incur only logarithmic cost per operation.

Why bother to test these properties, when Okasaki has already proved them? Well, the proof is informal, and proofs can be wrong. Okasaki's statements are in terms of "big O" notation, rather than the precise formulations above—the "`+ 1`" in `prop_cost_insert` came as a surprise, for example. Finally, we might have transcribed Okasaki's code incorrectly—or deliberately altered it. Actually, Okasaki uses a different definition of `insert`:

```
insert x h = merge (Fork x Null Null) h
```

This simplifies the proof, because now both insertion and deletion are defined in terms of `merge`, so only `merge` need be considered in the proof. But this definition of `insert` does not preserve balance, even when there are no deletions, which leads me to prefer my own definition above. Also, a specialised insertion function is likely to be more efficient than one using `merge`. But is it safe to replace the definition of `insert` with an optimised one with a different result? Okasaki's proof no longer applies directly, but the property above shows that it is.

We can take this example further. So far, we have tested the heap invariant and complexity properties. But apart from these, do `insert` and `delete` actually implement priority queues? To answer that, we need a *specification* that they should fulfill. One good way to specify them is via an abstract *model* of priority queues—such as ordered lists. Insertion is then modelled by the standard function to insert into an ordered list, and deletion is modelled by the function `tail`. To formalise this, we define a function mapping each skew heap to its model:

```
model :: Tree Integer -> [Integer]
model h = sort (flatten h)

flatten Null = []
flatten (Fork a l r) = a : flatten l ++ flatten r
```

Now, given a function `f` on ordered lists, and a function `g` on heaps, we can define a property stating that `f` correctly models `g` on a heap `h`, as follows:

```
(f `models` g) h =
  f (model h) == model (g h)
```

and formulate the correctness of insertion and deletion like this:

```
prop_insert n ops = ((List.insert n) `models` insert n) h
  where h = make ops
prop_deleteMin ops = size h>0 ==> (tail `models` deleteMin) h
  where h = make ops
```

Testing these properties succeeds, and after running many thousands of tests we can be reasonably confident that the stated properties do actually hold.

What this example shows us is that *QuickCheck changes the way we test code.* Instead of focussing on the choice of test cases—trying to guess which cases may reveal errors—we leave that instead to QuickCheck, and focus on the *properties* that the code under test should satisfy. Program development with QuickCheck strongly resembles formal program development, emphasizing formal models, invariants, and so on—but with labour-intensive proofs replaced by instant feedback from testing.

This approach has proved very attractive to the Haskell community, and QuickCheck has become widely used. One of the most impressive applications is in the development of `Data.ByteString`, described elsewhere in this volume. The code contains over 480 QuickCheck properties, all tested every time a new version of the code is checked in. The various `ByteString` types are modelled abstractly by lists of characters—just as we modelled skew heaps by ordered lists above. Many properties test that `ByteString` operations are accurately modelled by their list equivalents, just like our `prop_insert` and `prop_deleteMin`. `Data.ByteString` achieves its high performance in part by programming GHC's optimiser with custom rewrite rules that perform loop fusion and other optimisations. Of course, it's vital that such rewrite rules, which are applied silently to user code by the compiler, preserve the meanings of programs. Around 40 QuickCheck properties are used to test that this is in fact the case.

QuickCheck is also used by Haskell developers in industry. For example, Galois Connections' Cryptol compiler uses 175 QuickCheck properties, tested nightly, to ensure that symbolic functions used by the compiler correspond correctly to their Haskell equivalents.

3 Software Testing

QuickCheck is a novel approach to software testing. But software testing enjoys a somewhat patchy reputation among academics. Dijkstra's influence runs deep: his famous observation that "Program testing can at best show the presence of errors, but never their absence" suggests that mere testing is a waste of time. His comment in the preface to *A Discipline of Programming*, that "None of the programs in this monograph, needless to say, has been tested on a machine", makes us almost ashamed to admit that we do indeed test our own code! We know that even after rigorous testing, countless errors remain in production software—around one every hundred lines on average [13]. Those errors impose a real cost on software users—according to a Congressional report in 2002, \$60 billion annually to the US economy alone. That is a lot of money, even in the US—\$200 a year for every man, woman and child. Isn't it time to give up on such an inadequate technique, and adopt formal program verification instead?

Before drawing that conclusion, let us put those figures in perspective. The US software industry turns over \$200–\$240 billion per year. Thus the additional cost imposed by residual errors is around 25–30%. To be economically viable, even a development method that guarantees to eliminate *all* software errors must cost no more than this—otherwise it is more economical simply to live with the errors. How does formal program verification measure up?

An impressive recent case study is Xavier Leroy's construction of the back end of a certified C compiler using Coq [12]. Leroy wrote around 35,000 lines of Coq, of which the compiler itself made up around 4,500 lines, and concluded that the certification was around eight times larger than the code that it applied to. It is reasonable to infer that certification also increased the cost of the code by a similar factor. While such a cost is acceptable in the aerospace domain that Leroy was addressing, it is clearly not acceptable for software development in general. It is not reasonable to expect formal verification to compete with testing unless the cost can be cut by an order of magnitude[2].

Thus we can expect testing to be the main form of program verification for a long time to come—it is the only practical technique in most cases. This does not mean that practitioners are happy with the current state of the art! But while they are concerned with the problem of residual errors, they are really rather more concerned about the *cost* of testing—around half the cost of each software project. This cost is particularly visible since it is concentrated towards the *end* of each project, when the deadline is approaching, sometimes imposing an uncomfortable choice between skimping on testing and meeting the deadline. Current best practice is to automate tests as far as possible, so they can be run nightly, and to derive additional value from automated test cases by interpreting them as *partial specifications*, as Extreme Programming advocates [3].

Yet automated testing of this sort has its problems. It is a dilemma to decide, for each property that the code should satisfy, whether one should write one test case, or many? Writing a single test case makes for concise test code, with a clear relationship between test cases and properties—but it may fail to test the property thoroughly, and it may be hard to infer what the property is from a single example. Writing many test cases is more thorough, but also more expensive, imposes future costs when the test code must be maintained, and may obscure the "partial specification" by its sheer bulk—anyone reading the testing code may fail to see the wood for the trees. As an example of the code volumes involved, Ericsson's AXD301 ATM-switch is controlled by 1.5 million lines of Erlang code, which is tested by a further 700,000 lines of test cases!

A further problem is that nightly regression testing is really testing for errors that have *already been found*—while it protects against the embarrassment of reintroducing a previously fixed error, it is clear that unless the code under test is changed, no new errors can be found. Indeed, 85% of errors are found the *first* time a test case is run [8], so repeating those tests nightly is only a cheap way to

[2] This is also the motivation for "lightweight" formal methods such as Microsoft's Static Driver Verifier [2] or ESC/Java [9], which use automated proof techniques to reveal bugs at a very *low* cost in programmer time. But these tools offer no guarantees of correctness—a fact brought home by ESC/Java's use of an unsound theorem prover! They can "at best show the presence of errors, but never their absence" just like testing—although potentially with greater accuracy and at lower cost.

find the remaining 15%. In other words, it can only play a relatively small part in the overall testing process.

QuickCheck has the potential to address all of these problems. QuickCheck properties make much better specifications than automated test cases, because they cover the general case rather than one or more examples. For the same reason, there is no need to write more than one QuickCheck property for each logical property to be tested—a wide variety of cases will be generated anyway. Thus QuickCheck code can be concise and maintainable, without compromising the thoroughness of testing. Moreover, each time QuickCheck is run, there is a chance of *new* test cases being generated, so if QuickCheck is run nightly then, as time passes, we can expect more and more errors to be found. We have demonstrated in practice that the *same* QuickCheck property can reveal widely varying errors, depending on the data which is generated. As a bonus, QuickCheck *adds value to formal specifications* by interpreting them as testing code, making it more worthwhile to construct them in the first place.

We conclude that not only is testing here to stay, but that a tool such as QuickCheck has much to offer software developers in industry today.

4 Shrinking

One of the problems with randomly generated test inputs is that they can contain much that is irrelevant—the "signal", that causes a test to fail, can be hidden among a great deal of "noise", that makes it hard to understand the failure. We saw an example of this above, where the counter-example found to `prop_balanced` was the long sequence of operations

```
[DeleteMin,Insert (-9),Insert (-18),Insert (-14),Insert 5,
Insert (-13),Insert (-8),Insert 13,DeleteMin,DeleteMin]
```

Clearly, at the very least the first `DeleteMin` is irrelevant, since it has no effect at all—it is ignored by the `make` function that converts this list to a skew heap!

To address this problem, newer versions of QuickCheck automatically *shrink* failing test cases after they are found, reporting a "minimal" one in some sense. Using one of these new versions instead, testing `prop_balanced` might yield

```
Skew> quickCheck prop_balanced
Falsifiable, after 22 successful tests (shrunk failing case 10 times):
[Insert (-9),Insert 12,Insert 8,Delete]
```

in which the failing case has been reduced to just four operations. Moreover, we know that removing any of these four would make the test succeed: all four operations are essential to the failure. (There is no guarantee, though, that there is *no* shorter sequence that provokes a failure: just that one cannot be obtained by removing an element from this particular test case. We do still sometimes produce longer failing cases for this property.)

Shrinking failing cases dramatically increases QuickCheck's usefulness. In practice, much time is devoted either to simplifying a failing case by hand, or to debugging and tracing a complex case to understand why it fails. Shrinking failing cases automates the first stage of diagnosis, and makes the step from automated testing to locating a fault very short indeed.

5 Quviq QuickCheck

Although QuickCheck proved popular among Haskell users, the industrial Haskell community is still rather small. However, Erlang supports functional programming, and enjoys a *mainly* industrial community of users. Moreover, that community is growing fast: downloads of the Erlang system were running at 50,000 a month in June 2006, and have been growing quite consistently at 80% a year for the past six years. I therefore decided to develop a version of QuickCheck for Erlang, now called Quviq QuickCheck.

At first sight, adapting QuickCheck for Erlang appears to be rather difficult: Erlang lacks lazy evaluation, and many of the functions in QuickCheck's interface *must* be non-strict; Erlang lacks a static type-checker, and Haskell QuickCheck chooses generators based on the type of argument a property expects; QuickCheck's generator type is a monad, and we make extensive use of Haskell's **do**-notation to define generators. In fact, none of these difficulties proved to be especially problematic.

– QuickCheck functions which must be lazy only use their lazy arguments once, so instead of *call-by-need* it is sufficient to use *call-by-name*—and this is easily simulated by passing 0-ary functions as parameters instead (fortunately, Erlang supports first-class functions). We spare the user the need to pass such functions explicitly by using Erlang *macros* (distinguished by names beginning with a '?') to generate them. Thus Quviq QuickCheck simply provides an interface made up to a large extent of macros which expand to function calls with functions as parameters.
– While Haskell QuickCheck does choose generators for property *arguments* based on their type, it has always provided a way to supply a generator explicitly as well. In Erlang, we must simply always do this. This is a smaller cost than it seems, because in more complex situations, the type of an expected argument is rarely sufficient to determine how it should be generated.
– We can use a monad in Erlang too, in the same way as in Haskell. While we lack Haskell's **do**-notation, we can give a convenient syntax to monadic sequencing even so, via a macro.

The example in the introduction can be rewritten in Erlang like this:

```
prop_revApp() ->
  ?FORALL(Xs,list(int()),
    ?FORALL(Ys,list(int()),
      lists:reverse(Xs++Ys)
      ==
      lists:reverse(Xs)++lists:reverse(Ys))).
```

There are trivial differences: Erlang function definitions use an arrow (`->`), variables begin with a capital letter (`Xs`), external function calls name the module as well as the function to be called (`lists:reverse`). The main difference, though, is the use of the `?FORALL` macro, whose arguments are a bound variable, a generator, and the scope of the ∀—the expansion of `FORALL(X,Gen,Prop)` is just `eqc:forall(Gen,fun(X)->Prop end)`. By using generators which look like types (`list(int())`), and macro parameters which bind variables, we provide a very natural-looking notation to the user.

Testing this property yields

```
13> eqc:quickcheck(example:prop_revApp()).
..........Failed! After 11 tests.
[1]
[-3,1]
Shrinking.....(5 times)
[0]
[1]
```

in which the counterexample found is displayed both before and after shrinking. In this case, we can see that QuickCheck not only discarded an unnecessary element from one of the lists, but shrank the numbers in them towards zero. The fact that the minimal counterexample consists of `[0]` and `[1]` tells us not only that both lists must be non-empty, but gives us the additional information that if the `1` were shrunk further to `0`, then this would no longer be a counterexample.

Quviq QuickCheck thus offers a very similar "look and feel" to the original.

6 State Machine Specifications

In early 2006 we began to apply QuickCheck to a product then under development at Ericsson's site in Älvsjö (Stockholm). But real Erlang systems use side-effects extensively, in addition to pure functions. Testing functions with side-effects using "vanilla QuickCheck" is not easy—any more than specifying such functions using nothing but predicate calculus is easy—and we found we needed to develop another library on top of QuickCheck specifically for this kind of testing. That library has gone through four quite different designs: in this section we shall explain our latest design, and how we arrived at it.

As a simple example, we shall show how to use the new library to test the Erlang *process registry*. This is a kind of local name server, which can register Erlang process identifiers under atomic names, so that other processes can find them. The three operations we shall test are

– `register(Name,Pid)` to register `Pid` under the name `Name`,
– `unregister(Name)` to delete the process registered as `Name` from the registry, and
– `whereis(Name)` which returns the `Pid` registered with that `Name`, or the atom `undefined` if there is no such `Pid`.

Although `register` is supposed to return a boolean, it would clearly be meaningless to test properties such as

```
prop_silly() ->
  ?FORALL(Name,name(),
    ?FORALL(Pid,pid(),
      register(Name,Pid) == true)).
```

The result of `register` depends on what state it is called in—and so we need to ensure that each operation is called in a wide variety of states. We can construct a random state by running a random sequence of operations—so this is what our test cases will consist of. We also need to ensure that each test case leaves the process registry in a "clean" state, so that the side-effects of one test do not affect the outcome of the next. This is a familiar problem to testers.

We made an early decision to represent test cases *symbolically*, by an Erlang term, rather than by, for example, a function which performs the test when called. Thus if a test case should call `unregister(a)`, then this is represented by the Erlang term `{call,erlang,unregister,[a]}`—a 4-tuple containing the atom `call`, the module name and function to call[3], and a list of arguments. The reason we chose a symbolic representation is that this makes it easy to print out test cases, store them in files for later use, analyze them to collect statistics or test properties, or—and this is important—write functions to shrink them.

We can thus think of test cases as small programs, represented as abstract syntax. A natural question is then: how powerful should the *language* of test cases be? Should we allow test cases to contain branching, and multiple execution paths? Should we allow test cases to do pattern matching? For a researcher in programming languages, it is tempting to get carried away at this point, and indeed early versions of our library did all of the above. We found, though, that it was simply not worth the extra complexity, and have now settled for a simple list of commands. We do not regard this is a significant loss of power—after all, when a test fails, we are only interested in *the path to the failure*, not other paths that might conceivably have been taken in other circumstances.

We did find it essential to allow later commands access to the results of earlier commands in the same test case, which presents a slight problem. Remember that *test generation*, when the symbolic test case is created, entirely precedes *test execution*, when it is interpreted. During test generation, the *values* returned by commands are unknown, so they cannot be used directly in further commands—yet we do need to generate commands that refer to them. The solution, of course, is to let symbolic test cases bind and reuse *variables*. We represent variables by Erlang terms of the form `{var,N}`, and bindings by terms of the form `{set,{var,N},{call,Mod,Fun,Args}}`. The test cases we generate are actually lists of such bindings—for example,

```
[{set,{var,1},{call,erlang,whereis,[a]}},
 {set,{var,2},{call,erlang,register,[b,{var,1}]}}]
```

[3] `unregister` is a standard function, exported by the module `erlang`.

which represents the Erlang code fragment

```
Var1 = erlang:whereis(a),
Var2 = erlang:register(b,Var1)
```

We refer to Erlang terms of the form `{var,...}` and `{call,...}` as *symbolic values*. They represent values that will be known during test execution, but must be treated as "black boxes" during test generation—while it is permissible to embed a symbolic value in a generated command, the actual *value* it represents cannot be used until test execution. Of course, this is an application of *staged programming*, which we know and love.

Now, in order to generate sensible test cases, we need to know what state the system under test is in. Thus we base our test generation on a *state machine*, modelling enough about the actual state to determine which calls make sense, and express the desired properties of their outputs. In this case, we need to know which pids are currently registered. We also need to know which pids are *available* to register: to guarantee that the pids we use don't refer, for example, to crashed processes, we will generate new process identifiers in each test case—and these need to be held in the test case state. Thus we can represent our state using a record with two components:

```
-record(state,{pids,    % list(symbolic(pid()))
               regs}).  % list({name(),symbolic(pid())})

initial_state() -> #state{pids=[], regs=[]}.
```

We have indicated the expected type of each field in a comment: `pids` should be a list of (symbolic) process identifiers, spawned during test generation, while `regs` should be a list of pairs of names and (symbolic) pids.

To define such a state machine, the QuickCheck user writes a module exporting a number of callbacks, such as `initial_state()` above, which tell QuickCheck how the state machine is supposed to behave. This idea is quite familiar to Erlang users, because it is heavily used in the OTP (Open Telecoms Platform) library.

We define how commands are generated in each state via a callback function `command(State)`:

```
command(S) ->
  frequency([{1,stop},
             {10,oneof(
                   [{call,?MODULE,spawn,[]}]++
                   [{call,erlang,register,
                     [name(),elements(S#state.pids)]}
                   || S#state.pids/=[]]++
                   [{call,erlang,unregister,[name()]},
                    {call,erlang,whereis,[name()]}
            ])}]).
```

Test cases are generated by starting from the initial state, and generating a sequence of commands using this generator, until it generates the atom `stop`. Thus, on average, the generator above will result in test cases which are 11 commands long. We choose (with equal probability) between generating a call to `spawn` (a function defined in the current module `?MODULE` to spawn a dummy process), `register`, `unregister`, and `whereis`. Generating a call to `register` chooses one of the `elements` of the `pids` field of the state—to guarantee that such a choice is possible, we include this possibility only if this field is non-empty. (`[X || Y]` is a degenerate list comprehension with no generator, which returns either the empty list if `Y` is `false`, or `[X]` if `Y` is `true`).

We also separately define a *precondition* for each command, which returns `true` if the command is appropriate in the current state. It may seem unnecessary to define *both* a command generator, which is supposed to generate an appropriate command for the current state, and a precondition, which determines whether or not it is. There are two reasons to define preconditions separately:

– We may wish to generate a wider class of commands, then exclude some of them via a more restrictive precondition—for example, after testing reveals that a tighter precondition is needed than we first supposed!
– Shrinking deletes commands from a test case, which means that the following commands in a shrunk test case may appear in a *different* state from the one they were generated in. We need to be able to determine whether they are still appropriate in the *new* state.

In this example, though, we need state no non-trivial preconditions:

```
precondition(S,{call,_,_,_}) -> true.
```

Of course, we also have to define how each command changes the state. This is done by the `next_state(S,V,{call,Mod,Fun,Args})` callback, which returns the state after `Mod:Fun(Args)` is called in state `S`, with the result `V`. In this example, `spawn` adds its result to the list of available pids,

```
next_state(S,V,{call,?MODULE,spawn,_}) ->
  S#state{pids=[V | S#state.pids]};
```

(where `[X|Y]` means "`X` cons `Y`", and `S#state{pids=...}` is a *record update* that returns a record equal to `S` except for its `pids` field). The `register` operation records its arguments in the `regs` component of the state,

```
next_state(S,V,{call,erlang,register,[Name,Pid]}) ->
  S#state{regs=[{Name,Pid} | S#state.regs]};
```

`unregister` removes its argument from that component,

```
next_state(S,V,{call,erlang,unregister,[Name]}) ->
  S#state{regs=[{N,P} || {N,P} <- S#state.regs, N/=Name]};
```

while `whereis` leaves the state unchanged:

```
next_state(S,V,{call,erlang,whereis,[Name]}) -> S.
```

These clauses make up a simple specification of the intended behaviour of the operations under test. The only tricky point to note is that the result parameter, `V`, is symbolic during test generation—its value will be `{var,1}`, `{var,2}` etc. Thus the states that we build are also partly symbolic—for example, spawning a new process and registering it under the name `a` results in the state `{state,[{var,1}],[{a,{var,1}}]}`. We also use the `next_state` callback during test execution, when it is applied to real values rather than symbolic ones—during execution the state after the same two operations will be something like `{state,[<0.51.0>],[{a,<0.51.0>}]}`.

Finally, we define a postcondition for each command—if any postcondition fails, then the test case fails. To begin with, let us define a trivial postcondition, so that tests fail only if an exception is raised.

```
postcondition(S,{call,_,_,_},R) -> true.
```

Now, using the state machine library, we define a property to test:

```
prop_registration() ->
  ?FORALL(Cmds,commands(?MODULE),
  begin {H,S,Res} = run_commands(?MODULE,Cmds),
        [catch unregister(N) || {N,_} <- S#state.regs],
        [exit(P,kill) || P <- S#state.pids],
        ?WHENFAIL(io:format("~p\n~p\n",[H,Res]),
                  Res==ok)
  end).
```

Here `commands(?MODULE)` generates test cases using the callbacks in the current module, and `run_commands(?MODULE,Cmds)` runs those test cases, returning a history (list of states and results), final state, and "result", which is `ok` if the test case succeeded. The next two lines clean up after the test case, by unregistering any processes that were left registered, and killing the processes that were spawned. For convenience, we use the `?WHENFAIL` macro to add an action that is performed only in failing cases—we print out the history and result.

Testing this property immediately reveals a problem:

```
15> eqc:quickcheck(registration_eqc:prop_registration()).
.Failed! After 2 tests.
[{set,{var,1},{call,registration_eqc,spawn,[]}},
 ...
 {set,{var,41},{call,erlang,register,[a,{var,26}]}}]
...
Shrinking.....(5 times)
[{set,{var,4},{call,erlang,unregister,[a]}}]
[]
{exception,
  {'EXIT',{badarg,[{erlang,unregister,[a]},
                   {eqc_statem,run_commands,5},...
```

We can see immediately how effective shrinking is: a test case of 41 commands was shrunk to just one! This single call to `unregister(a)` failed with a `badarg` exception, and the rest of the output is an uninteresting stack backtrace.

The problem in this case is that `unregister` raises an exception if there is no registered process with the given name—so if a test case begins with `unregister`, then it is bound to fail. Our specification does not take this into account. There are two ways to do so:

– *Positive testing*—restrict test cases to avoid the exception, by adding a suitable precondition to `unregister` (and optionally modifying the command generator to avoid generating such commands in the first place), or
– *Negative testing*—*catch* the exception in a local version of `unregister` which we use in test cases instead, and define a postcondition to check that the exception is raised in the correct cases.

Whichever approach we choose, QuickCheck quickly reveals another problem:

```
60> eqc:quickcheck(registration_eqc:prop_registration()).
Failed! After 1 tests.
...
Shrinking.........(9 times)
[{set,{var,5},{call,registration_eqc,spawn,[]}},
 {set,{var,6},{call,erlang,register,[a,{var,5}]}},
 {set,{var,16},{call,erlang,register,[a,{var,5}]}}]
[{{state,[],[]},<0.869.0>},{{state,[<0.869.0>],[]},true}]
{exception,
  {'EXIT',{badarg,[{erlang,register,[a,<0.869.0>]},
                   ...
```

Of course! We tried to register process `a` twice! If we try to register a process with the same name as an *already registered* process, we would expect registration to fail! Indeed, the Erlang documentation confirms that `register` should raise an exception if either the name, or the process, is already registered. We define a function to test for this case

```
bad_register(S,Name,Pid) ->
  lists:keymember(Name,1,S#state.regs) orelse
  lists:keymember(Pid,2,S#state.regs)
```

(`lists:keymember(Key,I,L)` tests whether a `Key` occurs as the `I`th component of any tuple in the list `L`). We define a local version of `register` which catches the exception, and add a postcondition to check that the exception is raised exactly when `bad_register` returns `true`.

Testing quickly revealed another error, in the case:

```
[{set,{var,4},{call,...,spawn,[]}},
 {set,{var,5},{call,...,register,[c,{var,4}]}},
 {set,{var,12},{call,...,spawn,[]}},
 {set,{var,13},{call,...,register,[c,{var,12}]}},
 {set,{var,21},{call,...,register,[a,{var,12}]}}]
```

The problem here was *not* that the second call to `register` raised an exception—that was expected. The test case failed because the postcondition of the *third* call to `register` was not satisfied—the call *succeeded*, but was specified to fail. The reason was an error in our specification—the definition of `next_state` above takes no account of whether or not `register` raises an exception. As a result, after the first two calls to `register` then our state contained *both* processes `{var,4}` and `{var,12}`, registered with the same name `c`! Then the third call was expected to raise an exception, because the process being registered was already registered as `c`. Correcting the specification, so that `next_state` returns an unchanged state if `bad_register` is true, fixed the problem. In fairness, the Erlang documentation does not *say* explicitly that the process is not registered if `register` raises an exception, even if that is a fairly obvious interpretation!

A subtlety: note that when we use `bad_register` in `next_state`, then it is applied to a partially symbolic state. So when `bad_register` tests whether the pid is already registered, it compares a *symbolic* pid with those in the state. Fortunately this works: symbolic pids are always variables bound to the result of a spawn, and different calls to spawn return different pids—so two symbolic pids are equal *iff* the pids they are bound to are equal. Care is required here!

We have now seen all of our state machine testing library: to summarize, the user defines callback functions

- `command` and `precondition`, which are used during test generation to generate and shrink test cases that "make sense",
- `postcondition`, which is used during test execution to check that the result of each command satisfies the properties that it should,
- `initial_state` and `next_state`, which are used during both test generation and test execution to keep track of the state of the test case.

Given these callbacks, the user can generate test cases using `commands(Mod)`, and run them using `run_commands(Mod,Cmds)`.

As we saw in the example, the definitions of these callbacks make up a simple and natural specification of the code under test. We quickly found misconceptions in our specification, and enhanced our understanding of the process registry. While most of the information in our specification is also present in the Erlang documentation, we did discover and resolve at least a slight ambiguity—that a process is not actually registered when `register` raises an exception.

As an interesting extension of this example, we decided to test the process registry in the presence of crashing processes. We could easily model process crashes at known points by inserting operations to stop processes explicitly into our test cases[4]. The Erlang documentation says nothing about a relationship between process termination and the registry, but we discovered, by refining our QuickCheck specification, that such a relationship does indeed exist. In brief, dead processes are removed automatically from the registry; attempts to register a dead process apparently succeed (return `true`), but do not change the registry state. This means that sequences such as

[4] This doesn't test a process crashing *during* a call to a registry operation.

```
register(a,Pid),
register(a,Pid)
```

can indeed succeed—*if* `Pid` refers to a dead process. We discovered that stopping a process causes it to be removed from the registry—but *only* after other processes have had a chance to run! To obtain predictable behaviour, we stopped processes using

```
stop(Pid) -> exit(Pid,kill), erlang:yield().
```

where the call to `yield()` gives up control to the scheduler, allowing time for deregistration. Without such a `yield()`, a sequence such as

```
register(a,Pid),
stop(Pid),
unregister(a)
```

may or may not succeed, depending on whether or not the scheduler preempts execution after the `stop`! We were quickly able to develop a formal specification covering this aspect too, despite the absence of documentation—and in the process discovered details that are unknown even to many Erlang experts.

7 Ericsson's Media Proxy

We developed our state machine library in parallel with a project to test Ericsson's Media Proxy, then approaching release. The Media Proxy is one half of a media firewall for multimedia IP-telephony—it opens and closes "media pinholes" to allow media streams corresponding to calls in progress to pass through the firewall, thus preventing other IP packets from travelling through the owner's network for free, and defending equipment behind the firewall from denial of service attacks. The Media Proxy opens and closes media pinholes in response to commands from a Media Gateway Controller, a physically separate device which monitors signalling traffic to detect calls being set up and taken down.

This architecture, of a Media Gateway controlled by a Media Gateway Controller, is standardised by the International Telecommunication Union. The ITU specifies the protocol to be used for communication between the two—the H.248, or "Megaco" protocol [16]. This specification is quite complex: the current version is 212 pages long. The Media Proxy only uses a subset of the full protocol though, which is defined in an internal Ericsson document, the Interwork Description—a further 183 pages. The Media Proxy is controlled by about 150,000 lines of Erlang code, of which perhaps 20,000 lines are concerned with the Megaco protocol.

When we began our project, the Media Proxy had already completed Function Test, and was undergoing System Test in preparation for release. This process takes 3-4 months, during with the development team focus all their efforts on finding and fixing errors. The team follow a disciplined approach to testing, with a high degree of test automation, and have a strong track record for quality and reliability [18]. We worked with Ulf Wiger and Joakim Johansson at Ericsson to test the Megaco interface of the Media Proxy in parallel, by using QuickCheck to generate sequences of Megaco messages to send to the Proxy, and check that its replies were valid.

The biggest part of the work lay in writing generators for Megaco messages. These messages can carry a great deal of information, and the message datatype is correspondingly complex. It is specified in the ITU standard via an ASN.1 grammar, which specifies both the logical structure of messages, and their binary representation on a communications channel, both at the same time. This grammar can be compiled by the Erlang ASN.1 compiler into a collection of Erlang record types, together with encoding and decoding functions for the binary representation. We could thus generate messages as Erlang data structures, and easily encode them and send them to the Proxy—and this test infrastructure was already in place when we began our project.

We did try generating purely random messages conforming to the ASN.1 grammar, and sending them to the Proxy. This was not a successful approach: the messages were all semantic nonsense, and so were simply rejected by the Proxy. This could be an effective form of negative testing, but in this project we were more concerned to test the *positive* behaviour of the Proxy—that it responds correctly to *meaningful* messages.

Thus we had to write QuickCheck generators for complex structures, respecting all the constraints stated in the standard and the Interwork Description. To give a flavour of this, here is a fragment of the ASN.1 grammar in the standard, specifying the structure of a media descriptor:

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

A media descriptor is a record (sequence), with fields `termStateDescr`, `streams`, etc. Some of the fields can be optional, as in this case, and each field name is followed by its type. In this case the `streams` field is of a union type—it can either be tagged `oneStream` and contain the parameters of a single media stream, or it can be tagged `multiStream` and contain a list (sequence) of stream descriptors. Clearly the protocol designers expect a single media stream to be a common case, and so have included an optimised representation for just this case.

The Interwork Description restricts media descriptors a little, as follows:

```
MediaDescriptor ::= SEQUENCE
{ streams CHOICE
  { oneStream   StreamParms,
    multiStream SEQUENCE OF StreamDescriptor
  }
}
```

When generating media descriptors, we must thus choose between the `oneStream` form and the `multiStream` form, depending on how many streams are to be included. The QuickCheck generator is as follows:

```
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}}.
```

Analysing this code, we can distinguish three distinct parts.

– Datastructure construction—the `'MediaDescriptor'` record paired with a `mediadescriptor` tag, containing a `streams` field that is either a `oneStream` or a `multiStream`. Very similar code appears in conventional test cases.
– We analyse the streams to be included, distinguishing the cases of one stream and many streams. Here we express part of the logic of the specification.
– At *one* point, we embed a QuickCheck function—`oneof`—to express a choice between alternatives.

Thus the code looks mostly familiar to Ericsson developers—the overhead of turning it in to a QuickCheck generator is very light.

Another example: the standard specifies stream parameters as follows,

```
StreamParms ::= SEQUENCE
{ localControlDescriptor LocalControlDescriptor OPTIONAL,
  localDescriptor        LocalRemoteDescriptor  OPTIONAL,
  remoteDescriptor       LocalRemoteDescriptor  OPTIONAL,
  ...,
  statisticsDescriptor   StatisticsDescriptor   OPTIONAL
}
```

but the Interwork Description says also that “LocalControl will be included in all cases except when no media (m-line) is defined in the remote SDP”, the remote SDP being a part of the remote descriptor appearing among the stream parameters above. Thus we need to know whether or not a remote media will be defined, at the time we decide whether or not to include a local control descriptor. There are quite simply two cases for stream parameters: with, and without, a defined remote media. This is simple enough to express in a QuickCheck generator—we simply decide which case we are in *first*:

```
streamParms(Mode) ->
 ?LET(RemoteMediaDefined, bool(),
case RemoteMediaDefined of
  true ->
    #StreamParms{ localControlDescriptor =
                    localControl(Mode),
                  localDescriptor =
                    localDescriptor(RemoteMediaDefined),
                  remoteDescriptor =
                    remoteDescriptor(RemoteMediaDefined)};
  false -> ...
end).
```

We choose a random boolean, `RemoteMediaDefined`, and if it is `true`, we both include a local control descriptor, and pass the boolean inward to `remoteDescriptor`, which then ensures that an m-line is indeed generated. `?LET(X,G1,G2)` binds the variable `X` to the value generated by `G1` in the generator `G2`—it is syntactic sugar for the 'bind' operator of the generator monad, and corresponds to Haskell's **do**-notation. Of course, this code itself is quite trivial—the interesting thing is that we can only write it thanks to the monadic interface that generators provide.

As soon as our message generators were complete, we began to experience crashes in the Media Proxy. They turned out to be related to the `StreamParms` above. The ASN.1 specification says that all the fields of a `StreamParms` record are optional—which means that it is valid to omit them all, which QuickCheck quickly did. Yet the ITU standard also defines an alternative concrete syntax for messages, as readable ASCII—and we were actually using the ASCII form of messages, to ease debugging. The ASCII form of messages is generated and parsed by a *hand-written* encoder and decoder—obviously, these cannot be generated from the ASN.1 grammar, because they use another syntax. That syntax in turn is defined in the ITU standard by an ABNF grammar... and *this* grammar requires a `StreamParms` record to contain *at least one field*! It doesn't matter which field it is, but at least one must be there. This story illustrates the dangers of giving two formal descriptions of the same thing, with no way to enforce consistency! Now, one would expect the ASCII encoder to reject the messages we generated with empty `StreamParms`, but it turned out that Ericsson's *encoder* followed the ASN.1 specification and permitted an empty record, while the *decoder* followed the ABNF and required at least one field. Thus we could generate and encode a message, that when sent to the Media Proxy, caused its decoder to crash. Clearly, the underlying fault here is in the standard, but Ericsson's encode and decoder should at least be consistent.

Our next step was to generate valid command *sequences*. The Megaco standard defines twelve different commands that the controller can send to the gateway, but we focussed on the three most important, which manipulate the state of a call, or *context* as they are known in Megaco-speak.

– The *Add* command adds a caller (or *termination*) to a context, creating the context if it does not already exist. Terminations are added to a context one-by-one—the Megaco standard permits arbitrarily many callers in a call, while the Media Proxy is designed to handle a maximum of two.

- The *Modify* command modifies the state of a termination, typically activating media streams once both terminations have been added to a context.
- The *Subtract* command is used to remove a termination from a context—when a call is over, both terminations need to be subtracted. When the last termination is subtracted from a context, the context is automatically deleted from the Proxy.

The normal case is that two terminations are added to a context, they are both modified to activate their streams, and then they are both subtracted again.

Contexts and terminations are assigned identifiers when they are first added to the Proxy, which are returned to the controller in the Proxy's reply to the Add message. These identifiers are then used in subsequent messages to refer to already created contexts and terminations. So it was vital that the test cases we generated could use the replies to previous messages, to construct later ones.

We used a predecessor of our state machine testing library to generate and run sequences of Megaco commands. We used a state which just tracked the identifier and state of each termination created by the test case:

```
-record(state,
  termination=[]    % list({symbolic(termid()),termstate()})
).
```

(The empty list is a default field value). For each termination, we kept track of which context it belonged to, and the streams that it contained:

```
-record(termstate,
  context,      % symbolic(contextid())
  streams=[]    % list({streamid(),streammode()})
).
```

Note that since both termination identifiers and context identifiers are allocated by the Proxy, then they are unknown during test generation, and are represented by symbolic components of the state. For example, the identifier of the first termination added might be represented by

```
{call,?MODULE,get_amms_reply_termid,[{var,1}]}
```

where `get_amms_reply_termid` extracts the identifier of a new termination from the reply to an *Add* message. As before, since we know where each termination and context identifier is created, we can refer to them symbolically by *unique* expressions, and compare identifiers by comparing their symbolic form.

We generated *Add*, *Modify*, and *Subtract* commands, being careful to modify and subtract only existing terminations, and to add no more than two terminations at a time to any context. To achieve the latter, we defined functions on the state to extract a list of *singleton contexts* (those with only a single termination), and *pair contexts* (those with two terminations). We could use these functions during test generation, thanks to our unique symbolic representation for context identifiers—we could tell, just from the symbolic state, whether or not two terminations belonged to the same context. Using these functions, we could define, for example, a precondition for *Add*, which ensures that we never try to add a third termination to any context:

```
precondition(S,{call,_,send_add,[Cxt,Streams,Req]}) ->
  lists:member(Cxt,
    [?megaco_choose_context_id
    | singletoncontexts(S)]);
```

(Here `?megaco_choose_context_id` is a "wild card" context identifier, which intructs the Proxy to allocate a new context—so this precondition allows *Add*s which both create new contexts and add a termination to an existing one.)

All of the sequences we generated were valid according to the Interwork Description, and so should have been executed successfully by the Proxy. But they were not—we found a total of four errors by this means. In each case, shrinking produced a minimal command sequence that provoked the error.

– Firstly, adding *one* termination to a context, and then modifying it immediately, led to a crash. This turned out to be because the code for *Modify* assumed that each media stream would have two "ends"—when only one termination was present, this was not the case.
– Secondly, adding a termination to a new context, and then subtracting it immediately, also led to a crash. Interestingly, we found this bug one day, but could not reproduce it on the next. This was because the main development team had also found the bug, and issued a patch in the meantime!
– Thirdly, adding two terminations to a context, and then modifying one of them, led to a crash *if the two terminations had differing numbers of streams.* For example, an attempt to connect a caller with audio and video to a caller with only audio might lead to this failure. The underlying reason was the same as in the first case: *Modify* assumed that every stream has two ends.
– Lastly, adding two terminations to a context, removing the second, adding a third and removing it again, and adding a fourth and removing it again, provoked a crash when the fourth termination was removed! We found this case by shrinking a sequence of over 160 commands, which demonstrates the power of shrinking quite convincingly! It is a test case that a human tester would be very unlikely to try. Of course, it is also unlikely to occur in practice—but the particular test case is just a symptom, not a cause. The underlying cause turned out to be that data-structures were corrupted the *first* time a termination was removed. Even if the corruption was survivable in the normal case, it is obviously undersirable for a system to corrupt its data. If nothing else, this is a trap lying in wait for any future developer modifying the code. It is interesting that QuickCheck could reveal this fault, despite knowing nothing at all about the Proxy's internal data.

One observation we made was that after each bug was found, virtually every run of QuickCheck found the same problem! There seems always to be a "most likely bug", which is more likely to be reported than any other. This is partly because of shrinking: a longer sequence provoking a more subtle bug, such as the fourth one above, is likely also to provoke the most likely one—at least, once some commands have been deleted. So shrinking tends to transform any failing case into one for the most likely bug. We found that, to make progress, we had to add *bug preconditions* to our specification to guarantee that the known bugs would not be provoked. For example, we changed the precondition for *Modify* to

```
precondition(S, {call,_,send_modify,[Cxt,...]}) ->
  lists:member(Cxt, paircontexts(S));
```

to avoid the first bug above. Formulating these bug preconditions is useful in itself: it makes us *formulate a hypothesis* about when the bug appears, *test the hypothesis* by verifying that the precondition does indeed avoid the bug, and *document the bug* in the form of this extra precondition.

This entire study took only around 6 days of work (spread over 3 months), during which we wrote about 500 lines of QuickCheck code (since reduced to 300 by using our latest state machine library). Bearing in mind that the Proxy was already well tested when we started, finding five errors is a very good result.

In a way, it is rather surprising that such simple sequences as the first three cases above were not tested earlier! We believe this is because, while it is quite easy to adapt existing test cases by varying parameters in the messages they contain, it is much harder to construct a sensible sequence of messages from scratch. Indeed, a number of "normal case" sequences are contained in the Interwork Description, and it is likely that these formed a basis for early testing at least. By generating any valid message sequence, we could explore a much wider variety of sequences than could reasonably be tested by manually constructed cases—and so the bugs were there to be found.

We were curious to know how valuable QuickCheck would have been if it had been available earlier in the development process. To find out, we recovered an older version of the Proxy software from Ericsson's source code repository, and tested it using the *same* QuickCheck specification. We found nine errors in six hours, most of the time being spent on formulating appropriate bug preconditions, so that the next bug could be discovered. Ericsson's fault reporting database contained just two reported faults for that version of the software, one of which was among the nine that QuickCheck found, and the other of which was in a lower level part of the software not tested by our specification. This suggests QuickCheck could have helped to find many bugs much earlier. It also demonstrates that the same properties can be used to find many different errors.

It is true that the bugs we found (with the exception of the *Add/Subtract* problem) would not have affected Ericsson's customers—because the Media Proxy is initially sold *only* as part of a larger system, which also contains an *Ericsson* media gateway controller. Ericsson's controller does not send message sequences of the kind that we discovered provoke bugs. On the other hand, we may wonder how the Proxy developers *know* that? After all, the interface between the two is specified by the Interwork Description, which makes no such restrictions. It turns out that the documentation does not tell the whole truth—the teams developing the two products also communicate informally, and indeed, the products have been tested together. So if Ericsson's controller *did* send sequences of this sort, then the bugs would probably have been found sooner. Part of the benefit of QuickCheck testing may thus be to clarify the specification—by making our "bug preconditions" part of the Interwork Description instead. Clarifying the specification is important, not least because the Media Proxy will eventually be used together with controllers from other manufacturers, and at that point it will be important to specify *precisely* what the Proxy supports.

This project was both instructive and sufficiently successful to persuade Ericsson to invest in a larger trial of QuickCheck. We are now in the process of training more users, and helping to introduce QuickCheck testing into several other projects at varying stages of development. We look forward to exciting developments as a result!

8 Concurrency

Concurrent programs are more difficult to test with QuickCheck, because they may exhibit non-deterministic behaviour. Finding a test case which *sometimes* fails is not nearly as useful as finding a test case which always fails. In particular, shrinking is difficult to apply when testing is non-deterministic, because the smaller tests performed while we search for a simplest failing case may succeed or fail by chance, leading to very unpredictable results. Nevertheless, we have had some success in applying QuickCheck to concurrent programs.

In one experiment, we tested a distributed version of the process registry, written by Ulf Wiger to provide a global name server. We constructed an abstract model of the registry, much like that in section 6, and used it to test that sequences of `register`, `whereis` and `unregister` calls returned the expected results. Then we wrote a property stating that for all *pairs* of command sequences, executed in separate processes, each call gave the expected result.

Unfortunately, the "expected result" depends on how the calls in the two processes are interleaved. Observing the actual interleaving is difficult, especially since the registry need not service the calls in the order in which they are made! Indeed, all we can really require is that the results returned by the registry calls in each process correspond to *some* interleaving of the two command sequences—any interleaving will do. We therefore formalised precisely this property in QuickCheck. Potentially we might need to explore *all possible* interleavings of the two sequences, and compare their results to the abstract model, which would be prohibitively expensive. However, we discovered that a simple depth-first search, cut off as soon as the interleaving prefix was inconsistent with the actual results, gave a fast testable property.

Initially, testing succeeded—because the Erlang scheduler allocates quite long time slices, and so although we spawned two parallel processes, each one ran to completion within its first time-slice. But then we instrumented the implementation of the registry with calls to `yield()` between atomic operations, thus ensuring that execution of our two processes would indeed be interleaved. As soon as we did so, we began to find errors. Moreover, they were repeatable, because by calling `yield()` so often, we were effectively using cooperative multi-tasking instead of the pre-emptive variant, and since the Erlang scheduler is actually a deterministic algorithm, it schedules cooperatively multi-tasking programs in a deterministic way. This form of testing proved to be very effective, and ultimately forced a complete redesign of the distributed process registry.

In another experiment, Hans Svensson applied QuickCheck to fault-tolerant distributed leader election algorithms [1]. In such algorithms, a group of nodes elect one to be the "leader", for example to maintain a global state. If the current leader crashes, a new one must be elected, and something sensible must also happen if a crashed leader recovers. Correctness properties include that a leader is eventually elected, and all nodes informed of its identity, and that there are never *two* leaders at the same time.

Svensson used an extension of QuickCheck which records a trace of events, and—by acknowledging events at random—controls the scheduling in ths system under test. The recorded traces then revealed whether or not testing succeeded.

Svensson began by testing an open source implementation by Thomas Arts and Ulf Wiger, already in use in the Erlang community. QuickCheck (and another random testing tool) revealed problems so severe that the code had to be abandoned. Svensson implemented a different algorithm due to Stoller [17], whose proof of correctness supplied many lemmata that could be tested by QuickCheck. Interestingly, QuickCheck revealed an error here too, connected with the way that node crashes are detected in Erlang, but it was easily fixed.

Both algorithms were proven correct in the literature, but their implementations did not work. The underlying reason is interesting: theoretical papers quite rightly make simplifying assumptions about the environment the algorithm will be used in, but real systems do not fulfill them precisely. Practitioners need to adapt the algorithms to the real situation, but then the correctness proofs no longer really apply. In fact, the assumptions are rarely even stated formally, with the result that we cannot really say whether the bug in the second implementation is also present in Stoller's paper—it depends on an aspect of the environment where Stoller's assumptions are not 100% precise.

Thus another way to use QuickCheck is to gain confidence that a formally verified algorithm has been correctly transferred to a real situation!

9 Testing Imperative Code

Can QuickCheck testing be applied to code written in imperative languages? Certainly it can! In fact, we tested the Media Proxy by sending it Megaco commands over TCP/IP—the fact that the Proxy software itself was also written in Erlang was quite irrelevant. In one of our follow-up projects, the system under test is actually written in C++, but this requires no changes at all in the approach. Provided we can conveniently invoke the system under test from Erlang or Haskell, then we can test it using QuickCheck.

But what if we just want to test a C or C++ API, for example, rather than a system that obeys commands sent over a network? Koen Claessen has worked extensively on this. One quite successful approach is just to generate random

C programs that exercise the API, and compile and run them in each test! C compilers are fast enough to make this practical. Another method is to generate an interpreter for API calls from an API specification, link that interpreter with the code under test, and run it in a separate process. QuickCheck can then be used to generate sequences of calls which are sent to the interpreter for execution, and to check the results which are sent back. By using this approach, Claessen has found (and simplified) many bugs in C++ applications.

Would it make more sense to make a *native* version of QuickCheck for C or C++? In fact, Claessen has done this too. The result was certainly fast, but ultimately, not as satisfactory. Remember that QuickCheck code consists not only of random generators, but usually also of a formal *model* of the system under test. QuickCheck is most effective if these models can be built simply and easily, and here, declarative programming languages are playing to their strengths. In comparison, a native imperative version is clumsy to use.

In fact, I believe that testing code is a very promising application area for declarative languages. It is not performance-critical, and since it does not form a part of the final system, the constraints on choice of programming language are much looser than usual. Indeed, it is already quite common to use a separate test scripting language, different from the implementation language of the code under test—so why shouldn't that language be declarative? I believe that the barriers to adopting declarative languages are much lower in this area than for programming in general—particularly if that makes a tool such as QuickCheck more convenient to use. Time will tell if I am correct!

10 Erlang vs. Haskell

It is interesting to compare Erlang and Haskell as host languages for QuickCheck. We initially expected an Erlang version to be a little clumsier to use than the Haskell original, because of the lack of lazy evaluation, Haskell's type system, and monadic syntax (see section 5). Yet the difficulties these caused turned out to be minor. On the other hand, Erlang's *lack* of a type system turned out to bring unexpected benefits. For example, the Haskell QuickCheck generator for times of day, represented as pairs of hours and minutes, is

```
liftM2 (,) (choose 0 23) (choose 0 59)
```

(where `(,)` is the pairing operator, and `liftM2` lifts it to operate on monadic values). The Erlang QuickCheck generator is

```
{choose(0,23), choose(0,59)}
```

(where `{X,Y}` is Erlang's notation for pairs). The Erlang notation is more concise and intuitive, and definitely easier to sell to customers! In general, Quviq QuickCheck permits any data-structure containing embedded generators to be used as a generator for data-structures of that shape—something which is very convenient for users, but quite impossible in Haskell, where embedding a generator in a data-structure would normally result in a type error. This technique is used throughout the generators written at Ericsson.

Moreover, our approach to state machine testing involved symbolic representations of programs. In Haskell, we would need to define a datatype to represent function calls, with one constructor per function under test, and write an interpreter for those calls—just as we did in section 2 for `insert` and `deleteMin`. In Erlang, we could represent a call just by two atoms and a (heterogenous) argument list, and provide a single generic interpreter `run_commands`, thanks to Erlang's ability to call a function given only its name and arguments. This reduces the programming effort for the library user quite significantly.

Of course, the penalty for using Erlang is that type errors are not found by a type checker! Instead they must be found by testing... but this is easier than usual thanks to QuickCheck. We made many type errors when constructing the complex datatype of messages intended for the Media Proxy—but we found them immediately by testing that all messages we generated could be encoded to ASCII, and decoded again. Far from being second best, we conclude that Erlang is actually a very suitable host language for QuickCheck!

11 Discussion

Random testing is an old technique [10], which is attracting renewed interest—as shown, for example, by the new *International Workshop on Random Testing*, first held this year. It has been very successful for so-called *fuzz testing*, where nonsense inputs are supplied to try to provoke software to crash [14]—"monkey testing" of GUIs is an example of this. Random testing is more difficult to apply for *positive* testing, where meaningful inputs are supplied to the software under test, and its correct behaviour is tested. QuickCheck's flexible control of random generation makes it particularly suitable for this task.

Shrinking failing test cases is a powerful diagnostic technique, due to Hildebrandt and Zeller [11], who used it, for example, to shrink a test case that crashed Mozilla from 95 user actions on a web page consisting of almost 900 lines of HTML, to three user actions on one line of HTML ! Their *delta debugging* method starts from *two* tests, a successful one and a failing one, and uses a generic algorithm to search the space between them for two most-similar tests, one successful, and one failing. QuickCheck searches only from a failed test, towards smaller test cases, but using arbitrary user-defined shrinking methods.

Even though shrinking is powerful, we find it works best when the *original* failing test is not too large. New QuickCheck users are often tempted to generate *large* test cases, probably because doing so by hand is labour intensive, while using QuickCheck it is easy. Yet large test cases run slowly—so fewer tests can be run in a reasonable time—and when they fail, the reason is hard to understand. Shrinking them is at best time consuming (because many tests must be run), and at worst, may not result in as small a failing test as possible. In our experience, it is better to run many, many small tests, rather than a smaller number of large ones. Most errors can be provoked by a small test case, *once the error is understood*—and it is these small test cases which we want QuickCheck to find.

There are, of course, errors that no small test can find, such as errors that occur when a large table overflows. We encountered such an error when testing an implementation of purely function arrays, represented as binary trees with lists of up to ten elements in the leaves. An optimised array construction function failed when constructing a tree more than two levels deep... that is, with more than 40 elements. QuickCheck rarely provoked this case, until we explicitly increased the test size. Yet, in such cases, the software *should* still work if the table were smaller, or if the lists in the leaves were up to three elements, rather than ten—so why not reduce these constants for testing? Doing so makes the boundary cases much more likely to be exercised, and so increases the probability of revealing errors. When looking for a needle in a haystack, nothing helps so much as making the haystack smaller!

We have found that a complete formal specification of the code under test is often unnecessary. Simple properties are often enough to reveal even subtle errors, which is good news for testers. A nice example is our discovery of data-structure corruption in the Media Proxy, using properties which only interact with it via protocol commands. However, more precise specifications may find errors with fewer tests, and find smaller failing cases, because the error is revealed faster. In our example, a single *Add* and *Subtract* would have been sufficient to reveal the corruption, instead of the seven-command sequence we found.

One subtle change that QuickCheck brings about is a change in the *economic value* of failing test cases. Developers tend to pounce on the first failing case, re-run it, turn on debugging and tracing, and generally invest a lot of effort in understanding *that particular case*. When test cases are constructed painfully by hand, or even reported in the field, then this makes sense—test cases are valuable, compared to the developer's time. When a new failing case can be generated in seconds, then this no longer makes sense. Perhaps the next run of QuickCheck will find a smaller case, and save much diagnostic effort! It makes sense to generate several failing cases, and choose the simplest to work with, rather than rush into debugging as soon as the first failure is found. Or, if the cases found are overcomplex, it may be worthwhile to improve the shrinking strategy, and see whether that leads to a simpler case to debug. Improved shrinking may bring benefits in many future tests as well, so the effort is well invested.

We have found that testing with QuickCheck is perceived as quite difficult by developers. It is initially hard to see what to test, and the temptation is to make minor random variation of parameter values, rather than formulate more general properties. Using QuickCheck successfully is close in spirit to finding a good way to formalise a problem—which has occupied plenty of researchers over the years! It is therefore important to develop good "model specifications" that developers can follow, and to *simplify, simplify, simplify* the use of QuickCheck as much as possible. A good example of this is our state machine testing library, which is built entirely on top of the QuickCheck core. In principle, this could have been written by any user—but in practice, if it took me four iterations to get the design right, after seven years experience of QuickCheck, then it is unreasonable to expect new users to develop such toolkits for themselves.

Thomas Arts and I have founded a start-up, Quviq AB, to develop and market Quviq QuickCheck. Interestingly, this is the *second* implementation of QuickCheck for Erlang. The first was presented at the Erlang User Conference in 2003, and made available on the web. Despite enthusiasm at the conference, it was never adopted in industry. *We tried to give away the technology, and it didn't work!* So now we are selling it, with considerably more success. Of course, Quviq QuickCheck is no longer the same product that was offered in 2003—it has been improved in many ways, adapted in the light of customers' experience, extended to be simpler to apply to customers' problems, and is available together with training courses and consultancy. That is, we are putting a great deal of work into helping customers adopt the technology. It was naive to expect that simply putting source code on the web would suffice to make that happen, and it would also be unreasonable to expect funding agencies to pay for all the work involved. In that light, starting a company is a natural way for a researcher to make an impact on industrial practice—and so far, at least, it seems to be succeeding.

Finally, recall that Koen Claessen and I originally developed QuickCheck for fun. Perhaps for that very reason, using QuickCheck *is* fun! We see developers on our courses filled with enthusiasm, raring to test their code. Testing is not always seen to be so alluring—indeed, it is often regarded as something of a chore. QuickCheck really *makes testing fun*—and that, in itself, is a worthwhile achievement.

Acknowledgements

Most of the material in this paper is based on joint work, with my colleages Koen Claessen and Hans Svensson at Chalmers, Thomas Arts at the IT University in Gothenburg, and Ulf Wiger and Joakim Johansson at Ericsson. Time constraints made it impossible to write this paper together, but their contributions are here nonetheless. Any mistakes are, of course, my own!

References

1. Thomas Arts, Koen Claessen, John Hughes, and Hans Svensson. Testing implementations of formally verified algorithms. In *Proceedings of the 5th Conference on Software Engineering Research and Practice in Sweden*, 2005.
2. T. Ball, E. Bounimova, B. Cook, V. Levin, J. Lichtenberg, C. McGarvey, B. Ondrusek, S. Rajamani, and A. Ustuner. Thorough static analysis of device drivers. In *EuroSys 2006*, 2006.
3. Kent Beck. *Extreme Programming Explained: Embrace Change, Second Edition.* Addison Wesley Professional, second edition, November 2004.
4. Koen Claessen and John Hughes. Quickcheck: a lightweight tool for random testing of haskell programs. In *ICFP '00: Proceedings of the fifth ACM SIGPLAN international conference on Functional programming*, pages 268–279, New York, NY, USA, 2000. ACM Press.
5. Koen Claessen and John Hughes. Testing monadic code with quickcheck. In *Haskell '02: Proceedings of the 2002 ACM SIGPLAN workshop on Haskell*, pages 65–77, New York, NY, USA, 2002. ACM Press.

6. Koen Claessen and John Hughes. Specification-based testing with QuickCheck. In Jeremy Gibbons and Oege de Moor, editors, *Fun of Programming*, Cornerstones of Computing. Palgrave, March 2003.
7. Koen Claessen, Colin Runciman, Olaf Chitil, John Hughes, and Malcolm Wallace. Testing and tracing lazy functional programs using quickcheck and hat. In Johan Jeuring and Simon Peyton Jones, editors, *4th Summer School in Advanced Functional Programming*, volume 2638 of *LNCS*. Springer, 2003.
8. Mark Fewster and Dorothy Graham. *Software test automation: effective use of test execution tools.* ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, 1999.
9. Cormac Flanagan, K. Rustan M. Leino, Mark Lillibridge, Greg Nelson, James B. Saxe, and Raymie Stata. Extended static checking for Java. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation (PLDI'2002)*, volume 37, pages 234–245, June 2002.
10. Dick Hamlet. Random testing. In J. Marciniak, editor, *Encyclopedia of Software Engineering*, pages 970–978. Wiley, 1994.
11. Ralf Hildebrandt and Andreas Zeller. Simplifying failure-inducing input. In *ISSTA '00: Proceedings of the 2000 ACM SIGSOFT international symposium on Software testing and analysis*, pages 135–145, New York, NY, USA, 2000. ACM Press.
12. Xavier Leroy. Formal certification of a compiler back-end or: programming a compiler with a proof assistant. In *POPL*, pages 42–54, 2006.
13. John Marciniak and Robert Vienneau. Software engineering baselines. Technical report, Data and Analysis Center for Software, 1996. http://www.dacs.dtic.mil/techs/baselines/.
14. Barton P. Miller, Louis Fredriksen, and Bryan So. An empirical study of the reliability of unix utilities. *Commun. ACM*, 33(12):32–44, 1990.
15. Chris Okasaki. Fun with binary heap trees. In Jeremy Gibbons and Oege de Moor, editors, *Fun of Programming*, Cornerstones of Computing, pages 1–16. Palgrave, March 2003.
16. Telecommunication Standardization sector of ITU. ITU-T Rec. H248.1, gateway control protocol. Technical report, International Telecommunication Union, September 2005.
17. S. Stoller. Leader election in distributed systems with crash failures. Technical Report 169, Indiana University, 1997.
18. Ulf Wiger, G&#246;sta Ask, and Kent Boortz. World-class product certification using erlang. *SIGPLAN Not.*, 37(12):25–34, 2002.