Software Testing with QuickCheck

John Hughes, Chalmers University of Technology and Quviq AB

May 22, 2009

1 Properties and Generators

1.1 Testing and Test Automation

How do we know that software works? The answer, almost always, is that we test it. For example, consider the `delete` function in the Erlang `lists` module, which removes an element from a list. We might test it in the Erlang shell as follows:

```
4> lists:delete(2,[1,2,3]).
[1,3]
5> lists:delete(4,[1,2,3]).
[1,2,3]
```

See, it obviously works!

The biggest problem with testing is actually *not* that bugs may remain in well-tested code—although of course they may. The *biggest* problem is that reaching acceptable quality by testing is so inordinately expensive! Testing typically accounts for around half the cost of a software project—so finding ways to reduce its cost, without compromising the quality of the result, is very valuable indeed.

Running tests manually is the most expensive way to perform them. In practice, developers automate their tests in order to write them once, then run them many times. For example, the two tests above might be automated via the function definitions

```
delete_present_test() ->
    lists:delete(2,[1,2,3]) == [1,3].

delete_absent_test() ->
    lists:delete(4,[1,2,3]) == [1,2,3].
```

Many *unit testing* tools exist to find and run such tests automatically, of which EUnit is the most popular for Erlang. Such tools make running the test suite very easy, which encourages developers to do so often. Running tests often, as software evolves, helps developers to catch mistakes that break something in

the code as soon as the mistake is made. This practice is the foundation of *test-driven development*, which encourages developers to write their tests even before the code itself. Under test-driven development, the tests are used as a criterion for when the job is done—coding is considered complete when all the tests pass. This approach to development is claimed to improve quality and productivity (although the results of actual studies are somewhat mixed).

Yet even automated testing is expensive. Thorough testing demands an enormous volume of test code: at Ericsson, 35% of all code written is test code. Even so, it can never be economic to write a test case for every possible scenario, with the result that there is always a risk that bugs remain undetected by the test suite. In practice, many test cases are added to the test suite to provoke an error already discovered by other means—such test cases do not help to *find* the error in the first place, but they do help developers to avoid making the same mistake twice.

1.2 Property-based Testing

How can we reduce the cost of automated testing? Since the costly part is writing the test code in the first place, the key to doing so must be to *generalise the test functions*, so that each function covers not one test case, but many. In the case of the `lists:delete` function, let us generalise the element and the list, and write *one* test function that can test `delete` with *any* element and list:

```
...lists:delete(I,L)...
```

We run into a problem immediately: we can no longer predict the "expected" result, as we did in the two tests above, unless of course we reimplement the `delete` function in another way—which rather defeats the purpose. Instead, we must find some other way to determine whether the result of `delete` is correct. We can do so by identifying a *general property* of the result that should hold in all cases—such as, that `I` does not occur in the list after the deletion:

```
prop_delete(I,L) ->
  not lists:member(I, lists:delete(I,L)).
```

This is not a complete test by any means—`lists:delete` might always return the empty list, for example, and still pass this test—but on the other hand, it is applicable in *any* test case. By identifying sufficiently many such properties, we can give a complete specification of the `delete` function, which can be used to test it with any inputs whatsoever.

Once we've identified a general property such as this, we would like to test it automatically in many different cases, *without* the effort of specifying each case manually. But since the property should now hold, whatever the inputs, then we can safely *generate* them instead of choosing them by hand. This is what QuickCheck does—we rewrite the definition above as a *QuickCheck property*,

```
prop_delete() ->
  ?FORALL({I,L},{int(),list(int())},
          not lists:member(I, lists:delete(I,L))).
```

and then test it using QuickCheck:

```
21> eqc:quickcheck(examples:prop_delete()).
.....................................................................
.................................
OK, passed 100 tests
```

We can read the additional line of code *logically* as

$$\forall \{\texttt{I,L}\} \in \texttt{int()} \times \texttt{list(int())}\ldots.$$

making a universal statement of the truth of the property. QuickCheck instead interprets `{int(),list(int())}` as a *test data generator*, which produces pairs of an integer and a list of integers, and uses it to generate test cases which are bound to the pattern `{I,L}` in the rest of the test. Note that the first argument of `?FORALL` is the *binding* occurrence of the variables `I` and `L`. Running `quickcheck` then performs (by default) 100 random tests of the property, and in this case, they all passed—each dot represents a successful test.

Of course, such a property is a partial *formal specification* of the `delete` function; using QuickCheck, we test code against a formal specification, rather than a set of test cases. We focus our effort on *formulating properties*, rather than on coming up with corner cases. We find this shift in perspective not only helps us test more effectively, but also improves our understanding of the code under test, and of the problem that it solves.

1.3 Failure Diagnosis and Shrinking

Having defined a QuickCheck property, we can test it in very many test cases—for example, as follows:

```
29> eqc:quickcheck(eqc:numtests(1000,examples:prop_delete())).
.....................................................................
.....................................................................
.....................................................................
.....................................................................
.....................................................................
.....................Failed! After 346 tests.
{2,[-7,-13,-15,2,2]}
Shrinking.(1 times)
{2,[2,2]}
```

The `numtests` function sets the number of tests that should be performed to 1,000, and QuickCheck then begins testing—but after almost 350 tests, one of them fails! The next line of output

```
{2,[-7,-13,-15,2,2]}
```

is the failed test case itself; it is the data generated in the `?FORALL`, and so is the value bound to the pattern `{I,L}`. So when `I` is `2`, and `L` is `[-7,-13,-15,2,2]`, then the property fails.

In this example, as in many cases, the randomly generated test case contains junk values that have nothing to do with the test failure itself. The next step is therefore to search for a *simpler*, but similar test case that also fails. We call this process "shrinking", and its result in this case is the last line of output:

```
{2,[2,2]}
```

That is, the property fails when `I` is `2` and `L` is `[2,2]`. This is a minimal example, derived (in this case) by discarding irrelevant elements from the first failing case that was generated. We can think of shrinking as filtering away the "noise" from the test case, that random generation inevitably creates, leaving just the "signal" that is actually responsible for the test failure. Shrinking is tremendously important in making QuickCheck useful, since it automates the first stage of fault diagnosis—finding a minimal example that provokes failure. Without it, randomly generated failing cases would often be so large as to be almost useless.

When we see the result of shrinking, then the reason the property fails is almost obvious. In this case,

```
16> lists:delete(2,[2,2]).
[2]
```

we only delete the *first* occurrence of `2` from the list,

```
17> lists:member(2,[2]).
true
```

with the result that `2` is still a member of the list after deletion,

```
18> not true.
false
```

and the test fails.

1.4 Conditional Properties

Now in fact, although the failed test does reveal a problem, it is not in the implementation of `delete`. Rather, the problem is sloppiness or a misconception on our part—the `delete` function does not delete *all* occurrences of the element from a list, but only *one* occurrence. From the documentation:

> "Returns a copy of List1 where the first element matching Elem is deleted, if there is such an element."

So we must correct the *property*, not the definition of `delete`.

One way to correct the property is to note that it does indeed hold, *provided the list contains no duplicates*. So if we instead formulate a *conditional property*, then our tests should pass. We can define such a property as follows:

```
prop_delete() ->
  ?FORALL({I,L},{int(),list(int())},
    ?IMPLIES(no_duplicates(L),
             not lists:member(I,lists:delete(I,L)))).

no_duplicates(L) -> lists:usort(L) == lists:sort(L).
```

where `no_duplicates(L)` returns `true` if `L` contains no duplicate elements, using the library function `lists:usort` which sorts a list and removes any duplicates in one go. The most interesting line here is

```
?IMPLIES(no_duplicates(L),...)
```

which we can read logically as `no_duplicates(L)` $\Longrightarrow \ldots$, and which restricts test cases to those where the precondition holds.

With this new definition, testing the property succeeds:

```
39> eqc:quickcheck(examples:prop_delete()).
.....................x........x..................x.x......xx....
.......x....x....xx....x..x..........x.x.........x..
OK, passed 100 tests
```

The crosses ('`x`') in the output represent test cases which were generated, but were not run because the precondition was not satisfied; they are not counted among the 100 successful tests.

1.5 Custom Generators

Conditional properties can be very useful, but at the same time they incur a cost during testing—some test cases are generated, then discarded. If too many test cases are discarded, then testing will become very slow. Wouldn't it be better just to *generate* lists without duplicates in the first place?

QuickCheck provides a powerful API for defining custom generators, and when test data satisfying complex invariants is needed, then writing a custom generator is the only reasonable approach. In this case, a good way to generate a random list without duplicates is first to generate a random list, and then remove duplicates from it. We can define a custom list generator which does so as follows:

```
ulist(Elem) ->
  ?LET(L,list(Elem),
       lists:usort(L)).
```

Here `Elem` is a *generator* for list elements, so `list(Elem)` is a generator for arbitrary lists of these elements. `?LET` *sequences* two generators: it binds `L` to the list generated by `list(Elem)`, then uses it in a second generator—in this case `lists:usort(L)`, which just returns `L` with the duplicates removed. Note that it would be wrong to write

```
ulist(Elem) ->
  L = list(Elem),
  lists:usort(L).
```

because `list(Elem)` returns a generator, not a list, and `lists:usort` would complain of an argument of the wrong type. Generators must be treated as an *abstract data type*, and sequenced using `?LET` when multi-stage generation is needed[1].

Once a custom generator is defined, then we can use it like any other:

```
prop_delete() ->
    ?FORALL({I,L},{int(),ulist(int())},
      not lists:member(I,lists:delete(I,L))).
```

This property will pass any number of tests *without* discarding any test cases.

1.6 Distribution of Test Cases

Returning to our original—buggy—property, note that the problem was quite hard to find: we had to run over 300 tests to do so. In retrospect, it's clear why: `I` is chosen from `int()`, while `L` is chosen from `list(int())`, and for the property to fail then `I` must occur in `L` not just once, but *twice*. The probability of finding a random integer twice in a randomly generated list is not very high! This is why many tests were needed to provoke the error.

However, it is dangerous to guess the probabilities of different kinds of test data; much better is to *measure* them. We can do so by instrumenting a QuickCheck property to collect statistics during testing. For example, we might instrument `prop_delete` as follows, to measure how often `I` appears at all in `L`:

```
prop_delete() ->
  ?FORALL({I,L},{int(),list(int())},
    collect(lists:member(I,L),
            not lists:member(I,lists:delete(I,L)))).
```

The effect of the line

```
collect(lists:member(I,L),...)
```

is to collect the value of `lists:member(I,L)` in each test, and after testing is complete, to display the distribution of the values collected. In this case, testing the instrumented property yields

```
34> eqc:quickcheck(examples:prop_delete()).
.....................................................
.....................................
OK, passed 100 tests
88% false
12% true
```

[1] Generators are a monad, and `?LET` is its bind—for those familiar with those concepts.

We can see from this that, most of the time, `I` doesn't even occur *once* in `L`, let alone twice!

Thus, almost ninety percent of the time, we are testing `lists:delete` in the case of an element that does not appear at all in the list. It is clear that this is not an efficient use of testing time. We can improve test efficiency by generating test cases more carefully, to increase the probability—or even to guarantee—that the element *will* occur in the list, at least once. One way to do so is to generate the list *first*, and then simply choose one of its elements to delete. We can express this by nesting `?FORALL`,

```
prop_delete_2() ->
  ?FORALL(L,list(int()),
    ?FORALL(I,elements(L),
      not lists:member(I,lists:delete(I,L))).
```

where `elements(L)` generates a random element of the list `L`. In fact, this property has to be refined slightly further, since `elements([])` fails with an exception—what could it generate, after all? We add a precondition to avoid this case:

```
prop_delete_2() ->
  ?FORALL(L,list(int()),
    ?IMPLIES(L /= [],
      ?FORALL(I,elements(L),
        not lists:member(I,lists:delete(I,L))))).
```

and now testing this revised property finds a counterexample quickly.

```
45> eqc:quickcheck(examples:prop_delete_2()).
.xx.x.x.xx...x.x...x....x.......xx.....Failed! After 28 tests.
[-8,0,7,0]
0
Shrinking...(3 times)
[0,0]
0
```

In this small example, the poor distribution of tests data wasn't really important. We could find the fault quickly anyway, just by running a few hundred tests. In more complex situations, measuring the distribution of test data, and ensuring it is appropriate, is *essential* to find errors in a reasonable time.

1.7 Properties that Fail

When a property fails, it is tempting just to correct it or delete it—after all, it was proven to be incorrect. Yet in some cases, it's worth retaining such properties, and recording the fact that they fail. We can do so by adding `fails` to the property definition:

```
prop_delete_misconception() ->
  fails(
    ?FORALL(L,list(int()),
      ?IMPLIES(L /= [],
        ?FORALL(I,elements(L),
          not lists:member(I,lists:delete(I,L))))))).
```

When such a property is tested, it is *expected* to fail:

```
49> eqc:quickcheck(examples:prop_delete_misconception()).
x...x.x....x..........OK, failed as expected. After 19 tests.
```

An error will be reported only if the property unexpectedly *passes*.

Failing properties serve two useful purposes.

- Firstly, they *document a misconception* that one might well harbour about the code. "You might think that XYZ holds, but oh no! Here is a counterexample." By documenting the misconception, we help to ensure that it will not be repeated.
- Secondly, they *test our test case generation*. If the distribution of test data that we generate is not good enough to falsify the property within the specified number of tests, then an error is reported.

1.8 Points to Remember

Using QuickCheck, we test code against a *formal specification*. Often, as in the example we presented, the errors that emerge are in the specification, not the code. Do not hold formal specifications in awe just because they are *called* specifications—they are as likely to be wrong as programs. Errors are revealed by *inconsistencies* between the properties and the code, which describe the same behaviour in two different ways. When an inconsistency is found, it is the developer's responsibility to decide whether the code or the specification is wrong, and make an appropriate correction. This results not only in high quality code, but in a *specification which has been tested against the code*, and is therefore much more likely to be correct than a formal specification which has simply been formulated.

The biggest danger in using QuickCheck is that we no longer *see* the test data—nor would we want to, there is far too much of it. As a result, we may be lulled into a false sense of security by a large number of passing tests, but fail to notice that the distribution is badly skewed. For example, if the `list(int())` generator were always to generate the empty list, then all the tests in this section would pass—but that would mean very little. QuickCheck users cannot abdicate responsibility for the test data, just because they do not see it—but they exercise that responsibility at a higher level. Rather than focussing on individual test cases, we *collect statistics* on the test data, and satisfy ourselves that it is relevant and thorough. Traditional tools such as code coverage measures can be used to help evaluate the quality of our test data.

1.9 Exercises

1. **Getting Started**.
   QuickCheck should already be installed on your account. To start it, start an Erlang shell, and type

   ```
   eqc:start().
   ```

   QuickCheck will download an activation from the licence server, and begin a session—you should see something like

   ```
   Starting eqc version 1.162 (compiled at {{2009,4,22},{16,10,31}})
   Licence reserved until {{2009,4,22},{19,1,51}}
   ```

   If QuickCheck asks you for a registration identifier when you try to start it, then it is not correctly installed. In this case, ask for help.

   When you finish using QuickCheck, you should use `eqc:stop()` to release the licence again (although this will happen anyway if you terminate the Erlang shell).

   You can test whether QuickCheck is working by running

   ```
   eqc:quickcheck(true).
   ```

   which tests the property `true`—which should pass 100 tests, of course.

   QuickCheck is supplied with HTML documentation (generated by `edoc`); you should find a bookmark to it in your browser. The two modules we will use today are `eqc`, which defines *properties* and the shell API, and `eqc_gen` which defines *generators*. Don't take more than a quick look at the documentation right now: it is more important to get on with some practical exercises.

   You have been provided with a file `lecture1.erl`, containing the example properties presented above. Open the file in an editor. If you are using Emacs, then you should find that a "QuickCheck" menu appears whenever you edit an Erlang file. If you do not see this menu, then try running

   ```
   eqc_emacs_mode:install().
   ```

   in an Erlang shell, then restarting Emacs. Clicking on the menu, you should see sub-menus "Properties" and "Generators" that contain all the forms of property and generator that QuickCheck provides: if you're looking for something, this is a good way to find it. Choosing one of the functions from the menu prompts you for the parameters. The "full module header" entry under "Properties" can be used for quickly creating a new QuickCheck specification module. If you aren't using Emacs, then no such support is provided. You will need to manually insert

```
-include_lib("eqc/include/eqc.hrl").
```

at the head of each module that uses QuickCheck, to make the QuickCheck API available.

Compile the `lecture1` file, and use QuickCheck to test the two properties it contains.

```
c(lecture1).
eqc:quickcheck(lecture1:prop_delete()).
eqc:quickcheck(lecture1:prop_delete_misconception()).
```

2. **Simple numeric properties**. Add properties to the lecture1 file stating that the square of a real number is always positive, and that the square root of a real squares to the original real. You will need the generator for real numbers, which is called `real()`, and the square root function, which is called `math:sqrt`.

   You may find this informal partial grammar of properties helpful:

```
<property> ::= ?FORALL(<pattern>,<generator>,<property>)
             |  ?IMPLIES(<boolean expression>,<property>),
             |  collect(<expression>,<property>)
             |  begin <erlang expressions>, <property> end
```

   The last case was not presented in the lecture, and is just there to remind you that a `begin`...`end` block whose value is a property can also be used as a property. The Erlang expressions can also bind variables—for example, they might be

```
Y = math:sqrt(X)
```

3. **Properties of deletion**. The properties developed in the lecture are far from a complete characterization of deletion. To strengthen the specification, implement, test, and debug if necessary, these properties too:

   (a) Deleting an element that does *not* occur in a list leaves the list unchanged.

   (b) Deleting `X` from `L1++[X]++L2` returns `L1++L2`.

   Together, these properties completely characterize deletion.

4. **Choice of test data** So far, we have generated list elements using the primitive generator `int()`, which generates values between ± 30, roughly. We could have specified the upper and lower bounds explicitly, using the generator `choose(-30,30)` instead. Use this generator to experiment with larger and smaller ranges of test data, and investigate the effect this variation has on the number of tests needed to falsify the property

```
prop_delete() ->
  ?FORALL({I,L},{elem(),list(elem())},
          not lists:member(I, lists:delete(I,L)))
```

(where `elem()` is the element generator that you define). What general lesson can you draw about the choice of test data generator?

5. Generating lists without duplicates. We saw a simple way to generate lists without duplicate elements above: just generate a random list, then sort it with `usort`. One disadvantage of this method is that it only generates *sorted* lists, and so the properties we tested using it might only hold for sorted lists. Define your own generator for lists without duplicates that does not suffer from this limitation—rather, it should potentially be able to generate *any* list of elements without duplicates. You may find the function `eqc_gen:sample(G)` useful, which displays a selection of values generated by `G`; alternatively you can collect statistics about your generated values using a property of the form

   ```
   prop_collect() ->
     ?FORALL(X,<my generator>,
             collect(<some property of X>,
                     true)).
   ```

   Hint: one solution to this exercise uses the list difference operator `Xs--Ys`.

1.10 lecture1.erl

```
-module(lecture1).
-include_lib("eqc/include/eqc.hrl").

-compile(export_all).

prop_delete() ->
    ?FORALL({I,L},
    {int(),list(int())},
    ?IMPLIES(no_duplicates(L),
     not lists:member(I,lists:delete(I,L)))).

no_duplicates(L) ->
    lists:usort(L) == lists:sort(L).

ulist(Elem) ->
    ?LET(L,list(Elem),
 lists:usort(L)).

prop_delete_misconception() ->
  fails(
    ?FORALL(L,list(int()),
      ?IMPLIES(L /= [],
        ?FORALL(I,elements(L),
          not lists:member(I,lists:delete(I,L)))))).
```

2 Symbolic Test Cases

2.1 An Abstract Data Type of Dictionaries

In this section we shall introduce a number of methods useful for testing data-structure libraries. As an example, we shall test the `dict` module from the Erlang standard libraries. This module implements a key-value store as a purely functional abstract data type, with a rich API containing such functions as

- `new()`—which returns a new, empty dictionary,
- `store(Key,Val,Dict)`—which returns a new dictionary extending `Dict` with the pair `{Key,Val}`,
- `fetch(Key,Dict)`—which returns the value associated with `Key` in `Dict`.

The representation of dictionaries is complex, and our goal is not to understand it: we aim to test `dict` *without* needing to understand the internal representations. Thus we are engaged in **black box** testing, where we test the module's API, but need know nothing about the module internals. This is an appropriate kind of testing for the *user* of the `dict` module to apply.

We note in passing that the *developer* of the `dict` module would no doubt be interested in other properties, such as that invariants of the dictionary representation are preserved. This would be an example of "white box" testing, in which the module internals are also tested. Both kinds of testing are appropriate in some situations, and of course, QuickCheck can be used for either.

2.2 Generating Dictionaries

We begin by testing a very simple property indeed: the `dict` library provides a way to extract the list of keys from a dictionary; we shall check that each key is unique.

```
prop_unique_keys() ->
  ?FORALL(D,dict(),
    no_duplicates(dict:fetch_keys(D))).
```

Now this property might or might not be true, depending on how the `dict` library is designed, but it is reasonable to test it in any case—by doing so, we will improve our understanding of the `dict` library. Thus at this point we are using QuickCheck as a program understanding tool, rather than a testing tool.

However, we cannot begin to test this property *until we can generate dictionaries*. Since we do not understand the structure of dictionaries, the only way we can do so is using the API that the `dict` library provides. To begin with, we shall use only the `dict:new` and `dict:store` operations to generate test data.

A first stab at a dictionary generator might be as follows:

```
dict() ->
  oneof([dict:new(),
```

```
        ?LET({K,V,D},{key(),value(),dict()},
             dict:store(K,V,D))]).
```

The `oneof` function is provided by QuickCheck: it combines a list of generators into a generator that chooses one of the generators randomly, and then uses it to generate its own result. In this case we either generate a new dictionary, or generate a dictionary in two stages (`?LET`), first by choosing a key, value, and (recursively) a dictionary, then by storing the key value pair in that dictionary `D`. Note that we can freely use values as generators (`dict:new()`), and use tuples (or lists for that matter) containing generators as generators themselves.

We can use `oneof` to generate more interesting data as keys and values also—for example,

```
key() -> oneof([int(),real(),atom()]).
value() -> key().
```

Here the `atom()` generator just chooses from a representative sample of atoms:

```
atom() -> elements([a,b,c,undefined]).
```

(where `elements` is another QuickCheck function that just generates one of the elements of a list).

Now although the `dict()` generator above looks appealing, it does not actually work—and the reason is that Erlang is a strict programming language. The second choice in the list passed to `oneof` contains a recursive call of `dict()`, and this is an infinite recursion. Even though `oneof` only *uses* one element of its argument, the entire list, with all of its elements, must be evaluated since Erlang is strict. Thus, even when `oneof` chooses an empty dictionary, we still recursively construct another dictionary to store into—and the infinite loop is a fact.

We would often like to use *lazy evaluation* when building generators, so that only the part of the generator that is actually used is ever constructed. To make this possible, QuickCheck provides a generator construction `?LAZY(Gen)`, which is entirely equivalent to `Gen`, except that *constructing* the generator is always constant time. The price of constructing the argument is paid only if the generator is actually used. Thus we can avoid the infinite loop in the `dict()` generator as follows:

```
dict() ->
  ?LAZY(
    oneof([dict:new(),
           ?LET({K,V,D},{key(),value(),dict()},
                dict:store(K,V,D))])
       ).
```

Only one `?LAZY` is needed to guarantee that recursive calls to `dict()` terminate at once.

Now we can test our simple property, and... surprise, surprise... it fails!

```
9> eqc:quickcheck(eqc:numtests(10000,examples2:prop_unique_keys())).
.....................................................................
.....................................................................
.....................................................................
.....................................................................
.....................................................................
.....................................................................
.................................Failed! After 451 tests.
{dict,2,16,16,8,80,48,
      {[],[],[],[],[],[],[],[],[],[],[],[],[],[],[],[]},
      {{[[0.0|0.0],[0|0.0]],[],[],[],[],[],[],[],[],[],[],[],[],[],[],[]}}}
false
```

Thus this is an example of a dictionary which contains duplicate keys—but the failure report reveals a problem with our approach. Since we are engaged in *black box* testing, we do not understand the *representation* of the problematic dictionary that QuickCheck has reported. We do not even know where the keys are in the structure that we see! Clearly, displaying the representation of failing test cases is not appropriate for black box testing—and in the next subsection, we will see how to avoid it.

2.3 Symbolic Test Cases

As we just saw, the *representation* of test data is often not useful for understanding a test failure. Instead, we would like to know *how the test data was constructed* using the API under test. We can indeed display this information instead—if we construct a *symbolic representation* of the test case, instead of its actual value.

QuickCheck supports a simple symbolic representation of function calls as Erlang terms: the term `{call,`M`,`F`,`*Args*`}` is used to represent the function call M`:`F`(`...*Args*...`)`. Symbolic representations can be interpreted using the function `eval(X)`, which takes any term `X` *containing* symbolic function calls, and replaces each call by its value. Thus we can convert our dictionary generator to generate symbolic dictionaries instead, and add a call of `eval` to the property like this:

```
prop_unique_keys() ->
  ?FORALL(D,dict(),
          no_duplicates(dict:fetch_keys(eval(D)))).
```

The symbolic dictionary generator can be defined as follows:

```
dict() ->
  ?LAZY(oneof([{call,dict,new,[]},
               {call,dict,store,[key(),value(),dict()]}])).
```

Note that we no longer need to use `?LET` to generate the key, value, and recursive dictionary *before* constructing the call to `dict:store`...we can simply insert

their generators directly into the symbolic call. These generators will still be converted to proper values *before* the call to `dict:store` is actually made, since this now happens later on in `eval(D)`, rather than during the generation of the test data.

Now when we test the property, then the failing case is reported in an understandable form:

```
13> eqc:quickcheck(eqc:numtests(10000,examples2:prop_unique_keys())).
...Failed! After 4 tests.
{call,dict,store,
    [1,0.0,
     {call,dict,store,
         [0.0,undefined,
          {call,dict,store,
              [-1.0,c,
               {call,dict,store,
                   [0.0,1,
                    {call,dict,store,
                        [0,b,
                         {call,dict,store,
                             [1.0,1,
                              {call,dict,store,
                                  [a,-1.0,{call,dict,new,[]}]}]}]}]}]}]}]}]}
```

We can see exactly how the dictionary with non-unique keys was constructed using the `dict` API: it is the value of

```
dict:store(1,0.0,
  dict:store(0,0,undefined,
    dict:store(-1.0,c,
      dict:store(0.0,1,
        dict:store(0,b,
          dict:store(1.0,1,
            dict:store(a,-1.0,
              dict:new()))))))))
```

It must be admitted that this information is less useful than we might have hoped. It is hard to believe that such a complex construction is *necessary* to provoke the problem we have discovered. But wait... this is just the test case that QuickCheck first generated, which as we already learned contains a lot of random noise. Shouldn't it shrink to a simpler case? Indeed, it does—here is the result of shrinking in this case:

```
Shrinking.........(9 times)
{call,dict,store,
    [0,0.0,
     {call,dict,store,
         [0.0,a,
```

```
          {call,dict,store,
              [0.0,a,
               {call,dict,store,
                   [0.0,0,
                    {call,dict,store,
                        [0,a,
                         {call,dict,store,
                             [0.0,0,
                              {call,dict,store,
                                  [a,0.0,{call,dict,new,[]}]}]}]}]}]}]}]}]}
false
```

Unfortunately, this term is the same size as, and has the same structure as, the original. The only simplification is that some keys and values have been replaced by others—all the numbers are now zero, for example. Sadly, this is only of limited help in understanding the cause of the failure.

What we would like, of course, is to see QuickCheck simplify the test case be eliding unnecessary symbolic calls. Why doesn't this happen? The answer is that QuickCheck cannot know the *semantics* of the test data, and so cannot know which simplifications make sense. For example, *we* know that it makes sense to simplify `{call,dict,store,[K,V,D]}` to just `D`, because `D` is a simpler argument of the same type. But QuickCheck cannot know this—unless we tell it! In the next section we will see how to tailor QuickCheck's shrinking strategy so that it shrinks these examples well.

2.4 Shrinking Symbolic Tests

Shrinking is one of the most useful features of QuickCheck, and so in addition to built-in strategies, QuickCheck provides many ways to tailor the shrinking for particular kinds of test data. Shrinking is thus very much under the users' control, and can be used to replace test data by any data that the user considers "simpler"—even terms that are actually larger, if that is what the user wants. Shrinking is specified as a part of generators: a QuickCheck generator specifies a set of possible test data, a probability distribution over that set, and a shrinking strategy.

In particular, QuickCheck provides a construction

```
?LETSHRINK([X1,X2,...],
           [G1,G2,...],
           Result(X1,X2,...))
```

which is intended for use in recursive generators, such as the `dict()` generator above. `?LETSHRINK` generates a list of values using `[G1,G2,...]` (which are intended to be the recursive components of the result), binds them to the variables `X1`, `X2`,..., then returns a value generated by `Result`. Should the resulting test fail, then QuickCheck will try to simplify this result to each of the

values X1, X2,... in turn—in addition to the other shrinking steps inherent in G1,G2,\dots and \verbResult!.

For example, to allow symbolic calls of dict:store to be elided, we could use ?LETSHRINK as follows:

```
dict() ->
  ?LAZY(oneof([{call,dict,new,[]},
               ?LETSHRINK([D],[dict()],
                          {call,dict,store,[key(),value(),D]})])).
```

Now symbolic calls {call,dict,store,[K,V,D]} can shrink to D.

With this new generator, failing test cases shrink much more effectively. For example,

```
20> eqc:quickcheck(eqc:numtests(10000,examples2:prop_unique_keys())).
...Failed! After 4 tests.
{call,dict,store,
      [-1.0,b,
       {call,dict,store,
             [-1,-1.0,
              {call,dict,store,
                    [c,c,
                     {call,dict,store,
                           [0,-1,
                            {call,dict,store,
                                  [1.0,b,{call,dict,new,[]}]}]}]}]}]}]}
Shrinking.....(5 times)
{call,dict,store,[-1.0,a,{call,dict,store,[-1,0.0,{call,dict,new,[]}]}]}
false
```

All but two calls of store shrink away, and we are left with a dictionary that just contains two keys— -1.0 and -1. Hmmm. Does this dictionary contain duplicate keys? That may depend very critically on the exact definition of "duplicate"...!

We will not diagnose this problem further, but just close by remarking that, just as in the last section, effective shrinking makes the *cause* of a test failure very easy to spot.

2.5 Hoare Testing of Abstract Data Types

We have now developed a good *generator* for dictionaries, but the property we are testing is still very weak—many incorrect implementations would satisfy this property. In this section, we shall see how to define stronger properties that characterize correct behaviour of the library. These properties are based on Hoare's 1972 article on proving the correctness of abstract data type implementations, and so we refer to this kind of testing as "Hoare testing".

The key idea is to relate the implementation to an *abstract model* of the data type, which is to taken to define correct behaviour. Hoare used set-theoretic

models, but we need executable Erlang ones, and so we shall model dictionaries by lists of key-value pairs, or "property lists". With this simple representation, it is easy to see how each operation should behave; moreover we can use another Erlang library, the `proplists` library, to define the operations in the model.

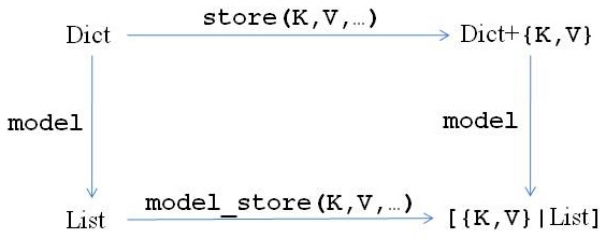
Having chosen a model, the first step is to *relate the implementation to the model*. This is done by defining an *abstraction function* that uses the `dict` API to convert a dictionary representation into the corresponding abstract value. In this case, this is very easy, since the `dict` API already contains an operation to do precisely that:

```
model(Dict) -> dict:to_list(Dict).
```

Now, to test each operation, we need to define a corresponding operation in the model. For example, adding a key-value pair to a property list can be achieved by

```
model_store(K,V,L) -> [{K,V} | L].
```

Correctness of each operation can then be formulated as a commuting diagram:



In other words, for any dictionary `Dict`, storing a key-value pair, and converting the result to its model, should give the same result as converting the original dictionary to its model, and adding the key-value pair using the model operation. If *every* operation in the API satisfies such a commuting diagram, then *any* program using the `dict` operations will behave in the same way as an abstract program using model operations instead. That is, if these properties hold, then `dict` is a correct implementation of the model.

Commuting diagrams such as these can easily be represented as QuickCheck properties. For example, the diagram above can be represented as:

```
prop_store() ->
  ?FORALL({K,V,D},
          {key(),value(),dict()},
    begin
      Dict = eval(D),
      model(dict:store(K,V,Dict)) ==
        model_store(K,V,model(Dict))
    end).
```

A similar property can be written for each of the API operations; if all these properties pass an arbitrary number of tests, then we know that the implementation of dictionaries is correct. Thus, a complete test suite for the `dict` module should contain one such property for each API operation.

Yet this alone is not enough to test the `dict` module thoroughly. It is also important that the `dict()` generator used in these properties can generate *any possible dictionary*—otherwise we are testing the correctness properties in only a subset of the relevant cases. The generator presented above may not do so, because it only constructs dictionaries using `dict:new` and `dict:store`. The real `dict` API provides a variety of other ways to construct dictionaries, such as `append`, `update`, `filter` and `merge`—and any one of these operations might break a dictionary invariant and construct a dictionary representation on which other operations fail. We therefore need to include *every* API operation that returns a dictionary in the `dict()` generator.

When we do so, then note that the property above does not only test the `store` operation, it actually tests `store` plus all of the operations used to construct the test data `Dict`—a much more thorough test.

We leave completion of the test suite as an exercise for the reader. For now, let us just test the property above—which reveals that it is not true!

```
29> eqc:quickcheck(eqc:numtests(10000,examples2:prop_store())).
..Failed! After 3 tests.
{0,0.0,{call,dict,store,[0,0.0,{call,dict,new,[]}]}}
false
```

That is, when we insert they key-value pair `{0,0.0}` into a dictionary that already contains it, then the implementation and the model disagree. Assuming that a well-tested library module is probably correct, then there is a deficiency in our model—but it is far from clear what that might be. In the next section we will see how to debug failing properties of this sort.

2.6 Debugging Failing Properties

When a property fails, we often need more information than just the inputs to the test if we are to diagnose the failure easily. An easy way to generate this information is just to print additional values using `io:format` from within the property. Yet if we simply add such calls to our properties, then output is generated during *every* test QuickCheck runs—including the tests that pass, and the tests that QuickCheck runs while searching for a minimal failing example. The result is usually an enormous volume of information, almost all of which is irrelevant.

Ideally, we would like to add print-outs that are performed *only* in test cases that QuickCheck is reporting, namely the first failed test, and the final result of shrinking. QuickCheck provides another form of property to achieve this:

```
?WHENFAIL(Action,Property)
```

is equivalent to `Property`, but also performs `Action` when a failed test is reported to the user. Typically `Action` is a call to `io:format` that displays useful additional information.

For example, in the properties used in Hoare testing, then the property holds if two terms in the model are equal. We can replace the equality test by a call of the following function

```
equal(X,Y) ->
  ?WHENFAIL(io:format("~p /= ~p~n",[X,Y]),
            X==Y).
```

which is logically equivalent, but reports the two values that differ when a test fails. With this modification, retesting `prop_store` generates the following output:

```
31> eqc:quickcheck(eqc:numtests(10000,examples2:prop_store())).
Failed! After 1 tests.
{a,0,
   {call,dict,store,
         [a,undefined,
          {call,dict,store,
                [0,b,{call,dict,store,[undefined,0,{call,dict,new,[]}]}]}]}]}}
[{0,b},{a,0},{undefined,0}] /= [{a,0},{0,b},{a,undefined},{undefined,0}]
Shrinking...(3 times)
{a,0,{call,dict,store,[a,a,{call,dict,new,[]}]}}
[{a,0}] /= [{a,0},{a,a}]
false
```

The penultimate line displays the two abstract values that differ; we see immediately that the one on the right (produced by our `model_store` function) contains a duplicate key, which the real implementation avoided. Referring back to the definition of `model_store`,

```
model_store(K,V,L) -> [{K,V} | L].
```

we see at once that it does *not* model a dictionary which maintains only a *single* value per key—which is what `dict` is intended to do. So as we suspected, our model needs to be adjusted to match the real intent of the library.

In the exercises that follow, we will improve this specification of the `dict` library.

2.7 Exercises

You are provided with a file `lecture2.erl`, which contains some of the definitions from this lecture—namely, a partial specification of the `dict` library. Compile the file, and run

```
eqc:module(lecture2)
```

which tests all the properties exported from the module. You will see that both properties fail.

1. **Property debugging**. The property `prop_dict()` expresses a correspondance between dictionaries constructed from a property list using the `from_list` function, and the original property list—but it fails. Generate several counter-examples using QuickCheck, examine them for common features, and form a hypothesis about why the property fails. Make a simple correction to the property so that it passes instead.

2. **Correcting the model**. The property `prop_store()` performs Hoare testing of the `store` operation, but it fails because the model is incorrect. Use QuickCheck to derive counterexamples to the property, then adjust the model (*i.e.* the functions `model` and `model_store`) until the property passes. Do *not* modify the property itself—there is no need to.

3. **Adding an operation**. So far, the only operations tested are `new` and `store`. Read the documentation of `dict:erase(Key,Dict)`, and define a model version `model_erase` and Hoare correctness property `prop_erase`. Use QuickCheck to check that your model is correct.

4. **Extending the generator**. To test `erase` properly, it should also be added to the `dict()` generator, since it provides yet another way to build dictionaries. Do so, and think specifically about how to shrink symbolic dictionaries once `erase` is introduced. In general, calling `erase` may make a dictionary smaller—does this mean that shrinking ought to *insert* calls of `erase`, rather than remove them?

5. **Controlling size in generators**. The purpose of this exercise is to add the `merge` function to the tested API—read the documentation of this function now. Testing `merge` demands that we construct a model and a Hoare property, but also that we add it to the `dict()` generator. This step is the trickiest, so it is the one we will focus on.

   To begin with, add a generator for calls of `merge` to the `dict()` generator, using the approach illustrated in the lecture. You will need to generate random 3-argument functions that return values, which you can do using the generator `function3(value())`. Compile your specification and test `prop_store()` again. You are likely to find that you run out of memory.

   The problem is that a symbolic call of `merge` contains *two* recursive dictionaries, with the result that generated terms can grow exponentially. In this situation, it is essential to limit the *size* of terms that `dict()` can generate. Add a natural number parameter `Size` to the `dict()` generator, and reduce it in the recursive calls, to ensure that generated terms cannot contain more than `Size+1` symbolic calls. (The smallest possible dictionary, `{call,dict,new,[]}`, contains one call, and must be generated when `Size` is zero). You can redefine an unparameterised dictionary generator via

```
dict() -> ?SIZED(Size,dict(Size)).
```

which gives control of the `Size` parameter to QuickCheck. Now check that your properties still pass using

```
eqc:module(lecture2).
```

6. **Adding real numbers**. The `key()` generator provided in `lecture2.erl` generates only integers and atoms—real numbers, which caused problems in the lecture, have been omitted for simplicity. Add `real()` as a possible way to generate keys, and retest all the properties in your module using

   ```
   eqc:module({numtests,1000},lecture2).
   ```

   which returns a list of the properties that fail. You can inspect the counterexamples found using `eqc:counterexamples()`. Diagnose the failures, and correct your model so that the properties pass once again. *Hint:* read section 6.11 of the Erlang reference manual, on "Term Comparisons".

7. **Specifying merge**. Complete your specification of `merge`, by adding a `model_merge` to the model, and a Hoare property `prop_merge()`. (This will be easiest to do if you do *not* include real numbers in the test data).

This exercise can of course be continued much further. There are many more operations in the `dict` module, waiting to be specified. There are other similar modules—such as `orddict` and `gb_trees`—which provide related functionality in a different way. These modules could be tested against the same (or a very similar) model. Why not choose an abstract datatype implementation from the Erlang libraries, and build a complete QuickCheck specification of your own?

2.8 lecture2.erl

```
%%% File    : lecture2.erl
%%% Author  :  <John Hughes@HALL>
%%% Description :
%%% Created : 21 May 2009 by  <John Hughes@HALL>

-module(lecture2).
-include_lib("eqc/include/eqc.hrl").

-compile(export_all).

%% A generator for symbolic dictionaries

dict() ->
    ?LAZY(oneof([{call,dict,new,[]},
 ?LETSHRINK([D],[dict()],
    {call,dict,store,[key(),value(),D]})])).

%% Auxiliary generators

key() ->
    oneof([int(),atom()]).

value() ->
    key().

atom() ->
    elements([a,b,c,undefined]).

%% Dictionaries constructed using from_list behave like property lists

prop_dict() ->
  ?FORALL(L,list({key(),value()}),
    ?IMPLIES(L/=[],
      ?FORALL(K,elements(proplists:get_keys(L)),
      equal(proplists:get_value(K,L) ,
    dict:fetch(K,dict:from_list(L))))))).

%% Hoare testing: the model

model(Dict) ->
    dict:to_list(Dict).

%% Hoare testing of store
```

```
model_store(K,V,L) ->
    [{K,V}|L].

prop_store() ->
    ?FORALL({K,V,D},
    {key(),value(),dict()},
    begin
Dict = eval(D),
equal(model(dict:store(K,V,Dict)),
      model_store(K,V,model(Dict)))
    end).

%% Property auxiliary

equal(X,Y) ->
    ?WHENFAIL(io:format("~p /= ~p~n",[X,Y]),
      X==Y).
```

3 Testing Stateful Systems

3.1 The Process Registry

In many cases, real software manipulates internal state and offers a stateful API. A useful testing tool must be able to test stateful systems also, not just purely functional libraries. In this section we introduce QuickCheck's support for testing stateful software using abstract state machines.

The example we use for illustration is the Erlang *process registry*. This is a local name server that enables Erlang processes to find services running on the same node. Processes are named by atoms, and the central part of the API is the three functions

- `register(Name,Pid)`—which creates an association between an atom `Name` and a process identifier `Pid`,
- `unregister(Name)`—which removes any association for the given `Name`, and
- `whereis(Name)`—which returns the pid associated with `Name`, if any.

These functions are all in the `erlang` module containing built-in functions; detailed documentation can be found in the documentation of that module.

Interestingly, the process registry is really just another example of a key-value store—and we shall test it against a model in a similar way.

3.2 Testing Stateful Interfaces

Stateful APIs offer new challenges for testing (one good reason to prefer purely function APIs whenever possible!). The state is an implicit argument to and result from every API call, yet it is not directly accessible to the test code. We would like to model the state abstractly, just as we did in the previous section, and test in a similar way—but because the state is not directly observable, then we *cannot* define an abstraction function `model` which recovers the abstract state before and after a test. Although such a function should exist in principle, and the same diagram ought to commute as in the previous section, we cannot *implement* our tests in this manner. We must find an alternative.

However, although we cannot *observe* the abstract state directly, we can *compute* it using state transition functions that model the operations in the API under test. These functions correspond to `model_store`, `model_erase` and so on defined above, and are used in the same way to predict the model state after each operation—but whereas in the last section we could directly compare the predicted model state to the actual observed one, in this section we will only be able to use the *results* returned by the API operations. Our approach will thus be to test whether the real observed results of each API call are consistent with the *predicted* model state. If the predicted state and the actual state ever become inconsistent, then this inconsistency will eventually be observed via the

result of some sequence of API calls—unless, of course, *no* sequence of calls can distinguish the two, in which case the difference is arguably of no importance.

The test cases we generate for stateful systems are *sequences* of API calls. We can gain a general idea of how such test cases are generated and used from the property used to test the registry:

```
prop_registry() ->
  ?FORALL(Cmds,commands(?MODULE),
    begin
      {H,S,Res} = run_commands(?MODULE,Cmds),
      cleanup(),
      ?WHENFAIL(
        io:format("History: ~p\nState: ~p\nRes: ~p\n", [H,S,Res]),
        Res == ok)
    end).
```

Referring to this definition, we see that:

- We *quantify* over a list of symbolic commands, `Cmds`, generated by the function `commands`, which is a part of QuickCheck. These correspond to the symbolic dictionaries of the previous section.
- We *execute* the list of commands using `run_commands`, another function provided by QuickCheck, which corresponds to `eval` in the previous section.
- The result of `run_commands` is a triple whose third component, `Res`, summarizes the outcome of the test. If this is the atom `ok`, then the test passed.
- The other components of the result contain debugging information; it is useful to display this when a test fails using `?WHENFAIL`.
- After each test, the registry must be restored to a known state in preparation for the next test; properties for testing stateful systems almost always need to contain clean-up code to do this.
- Both `commands` and `run_commands` take a *module name* as a parameter. This module provides callbacks that define the behaviour of the state machine model, and it is these callbacks that are the heart of the QuickCheck specification.

In the next sections we will explain the callbacks that make up a QuickCheck state machine specification. Definitions of these callbacks must be placed in, and exported from, the module whose name appears in the property above.

3.3 Generating Commands

The first callback we shall consider, called `command`, defines how the symbolic API calls that make up a test case should be generated. Since we intend to test `register`, `unregister` and `whereis`, then we might expect to define

```
command() ->
  oneof([{call,erlang,register,[name(),pid()]},
         {call,erlang,unregister,[name()]},
         {call,erlang,whereis,[name()]}
        ]).
```

where `name()` generates a random atom, and `pid()` generates a random process identifier. We generate names via

```
name() -> elements([a,b,c,d]).
```

choosing atoms from a small set so that a test case is likely to refer to the same name several times. But what about `pid()`? This should choose from a small pool of process identifiers—but process identifiers must be created dynamically. It is important to create *new* process identifiers each time a test case is run, since otherwise earlier tests may corrupt the test data (process identifiers) used in later ones, leading to test failures that are virtually impossible to diagnose.

It follows that process creation must be a part of the generated test cases. This is easy to achieve by defining a local function `spawn()` that just starts a process that does nothing, and including calls in test cases:

```
command() ->
  oneof([{call,erlang,register,[name(),pid()]},
         {call,erlang,unregister,[name()]},
         {call,erlang,whereis,[name()]},
         {call,?MODULE,spawn,[]}
        ]).
```

However, we still have to arrange for `pid()` to choose one the processes previously created by a `spawn()` in the same test case.

In order to do so, we maintain a *test case state* which collects all the process identifiers generated by `spawn()`, and we parameterise `pid()` (and `command()`) on this state so that previously spawned pids can be chosen:

```
command(S) ->
  oneof([{call,erlang,register,[name(),pid(S)]},
         {call,erlang,unregister,[name()]},
         {call,erlang,whereis,[name()]},
         {call,?MODULE,spawn,[]}
        ]).
```

The `command` callback is thus a one-argument function, not a zero-argument one as it appeared to be earlier. In fact, this kind of situation arises frequently: it

is often the case that a resource allocated by one API call needs to be returned as a parameter to a later one, and so a mechanism for supporting this is an important part of a state machine testing framework.

We now need to maintain a test case state for two different reasons: firstly, as an abstract model of the state of the system under test, and secondly, so that we can generate call sequences that reuse the results of earlier calls later in the sequence. In principle we might track two separate states, but we have chosen instead to merge them into one. We are thus now in a position to choose a model state for testing the process registry: it is an Erlang record defined as follows:

```
-record(state,{pids=[],
               regs=[]
              }).
```

The first component is a list of the pids spawned in the current test, while the second is a property list modelling the state of the registry. With this definition, we can finally define

```
pid(S) -> elements(S#state.pids).
```

3.4 Modelling State Transitions

We have now defined how to generate the next command from a test case state, but we have yet to define how each command *changes* the state. We do this via a `next_state` callback, which maps the state before a call, and the symbolic call, into the resulting state afterwards. Since this state may depend on the actual result of the call, then this is also supplied as a parameter. For example, the clause for `spawn` adds the result of the call (the parameter `V`, a pid) to the list of pids in the test case state:

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

The `next_state` function is used to simulate the test case state as a test case is generated, enabling the `command` generator in the previous section to refer to previously spawned pids.

Similarly, the state transitions for `register` and `unregister` make the expected changes to the modelled registry contents:

```
next_state(S,_V,{call,_,register,[Name,Pid]}) ->
  S#state{regs=[{Name,Pid}|S#state.regs]};
next_state(S,_V,{call,_,unregister,[Name]}) ->
  S#state{regs=proplists:delete(Name,S#state.regs)};
```

Finally, we include a catch-all clause specifying that other calls do not change the model state.

```
next_state(S,_V,{call,_,_,_}) ->
  S.
```

In this case the only other call is `whereis`, and sure enough, we do not expect it to change the registry state.

We also need to define the initial test case state, which is done by a third callback that can simply construct a `state` record with default field values:

```
initial_state() -> #state{}.
```

3.5 Conditional Generation

Glossing over a few details, we are now ready to run our first tests. When we do so, they fail immediately:

```
40> eqc:quickcheck(reg_eqc:prop_registry()).
Failed! Reason:
{'EXIT',{eqc,elements,[[]]}}
After 1 tests.
false
```

The error message indicates that the test failed because of an exception raised when `elements([])` was called—to generate an element of the empty list! Of course, this is impossible, so `elements` should not be called in this way—we have a bug in our command generator. The problem is that in the initial test case state, `S#state.pids` is empty—and so if we try to generate a call of `register`, then we must choose a pid from the empty list, which raises the exception. The solution is to include `register` as an alternative in the list of possible commands *only* if `S#state.pids` is non-empty. We could do so using a `case`-expression, but we prefer a little trick which offers a very concise notation:

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

The first operand of `++` above is a *list comprehension with no generators*, a form one does not see very often, but which is just what we need here. Such a comprehension is evaluated as follows: `[X||true]` evaluates to `[X]`, while `[X||false]` evaluates to `[]`. Thus the effect is to include the call to `register` in the list from which `oneof` chooses precisely when the list of pids in the state is non-empty. In fact, the generator for `register` is not even evaluated if the condition is false—so such a list comprehension can be used as a kind of "**if**...**then**..." expression in Erlang, something many programmers more used to imperative languages miss sorely.

With this change, our command generator works, and we can start running tests for real.

3.6 Specifying Preconditions

Once again, when we try to test our property, then it fails immediately:

```
42> eqc:quickcheck(reg_eqc:prop_registry()).
Failed! After 1 tests.
[{set,{var,1},{call,reg_eqc,spawn,[]}},
 {set,{var,2},{call,erlang,whereis,[b]}},
 {set,{var,3},{call,erlang,unregister,[d]}},
 {set,{var,4},{call,reg_eqc,spawn,[]}},
 {set,{var,5},{call,reg_eqc,spawn,[]}},
 {set,{var,6},{call,erlang,register,[b,{var,5}]}},
 {set,{var,7},{call,erlang,whereis,[c]}},
 {set,{var,8},{call,erlang,whereis,[a]}},
 {set,{var,9},{call,erlang,whereis,[a]}},
 {set,{var,10},{call,erlang,register,[d,{var,1}]}},
 {set,{var,11},{call,erlang,unregister,[d]}},
 {set,{var,12},{call,erlang,register,[c,{var,1}]}}]
History: [{{state,[],[]},<0.2066.0>},{{state,[<0.2066.0>],[]},undefined}]
State: {state,[<0.2066.0>],[]}
Res: {exception,{'EXIT',{badarg,[{erlang,unregister,[d]},
                                 <...7 more lines...>]}}}
```

We can see from the output that test cases are more than just a list of symbolic commands: they are a list of *symbolic variable bindings*. Just as `{call,M,F,Args}` represents a function call symbolically, so `{var,N}` represents a variable... think of it as v_N. The test case above binds `{var,1}` to the result of `spawn`, `{var,2}` to the result of `whereis`, and so on. Variables can be reused in the arguments of later symbolic calls—for example, `{var,1}` is reused as an argument in the line binding `{var,10}`. One of the main differences between `eval` and `run_commands` is just that the latter manages these symbolic variable definitions, as well as performing symbolic calls.

Looking at the diagnostic output from the `?WHENFAIL`, we see that the reason for the test failure was a `badarg` exception raised by `unregister`. However, the test case is too complex for us to understand the problem immediately. Luckily, it shrinks to a much simpler one—continuing the QuickCheck output, we see

```
Shrinking....(4 times)
[{set,{var,3},{call,erlang,unregister,[a]}}]
History: []
State: {state,[],[]}
Res: {exception,{'EXIT',{badarg,[{erlang,unregister,[a]},
                                 <...7 more lines...>]}}}
false
```

This test does nothing other than call `unregister(a)`. Now it is pretty clear that the exception is raised because we are unregistering a name which has not previously been registered—and indeed, the documentation confirms that

`unregister` is supposed to raise an exception in this case. We must revise our model to reflect this.

One way to do so is to specify a *precondition* for `unregister`, stating that it may only be called when the name to be unregistered is actually in the registry. QuickCheck uses a fourth callback, `precondition`, to determine when API calls may be made. We can define `unregister`'s precondition as follows:

```
precondition(S,{call,_,unregister,[Name]}) ->
  unregister_ok(S,Name);
precondition(_S,{call,_,_,_}) ->
  true.

unregister_ok(S,Name) ->
  proplists:is_defined(Name,S#state.regs).
```

(where the second clause of the `precondition` callback was already present—otherwise the tests would have failed with an undefined function). All preconditions are satisfied in all tests that QuickCheck generates, so with this addition then the problem we encountered can no longer occur.

3.7 Specifying Postconditions

The change we just made to the specification is an example of "positive testing"—we restrict test cases to those in which we expect `unregister` to work. Yet it is in a sense unsatisfactory. We know that `unregister` is *intended* to raise an exception when called with a name that is not in the registry, but positive tests will not test this. We might therefore prefer *not* to specify a precondition for `unregister`, but instead check that an exception is indeed raised whenever it ought to be. This kind of testing is referred to as "negative testing", because we also test the (specified) error behaviour of the system under test.

We can perform this kind of testing using QuickCheck by removing the unnecessary precondition, and adding a *postcondition* for `unregister` instead. Postconditions are defined via the fifth and final callback that makes up a QuickCheck state machine model, which is passed the state before a call, the symbolic call, and the actual result of the call, and is expected to return `true` or `false`. We can add a postcondition for `unregister` as follows:

```
postcondition(S,{call,_,unregister,[Name]},Res) ->
  case Res of
    {'EXIT',_} -> not unregister_ok(S,Name);
    true ->       unregister_ok(S,Name)
  end;
postcondition(_S,{call,_,_,_},_Res) ->
  true.
```

where once again, the last clause was already present. Examining the code, we see that if the actual result `Res` is an exit value, then the postcondition holds

only if the call was expected to fail, while if the actual result is `true`, then the postcondition holds only if the call was expected to succeed.

Note that we have made heavy use of `unregister_ok`, which defines when a call of `unregister` is expected to succeed. It is because such conditions are often moved between pre- and post-conditions that we defined a separate function to test it in the first place.

Our job is not quite complete. We have added a postcondition to *test* for an exception, but this will have no effect unless we arrange to *catch* the exception first—QuickCheck always considers an uncaught exception to be a test failure. We can catch the exception by defining a local version of `unregister` that does so,

```
unregister(Name) -> catch erlang:unregister(Name).
```

and replacing calls of `erlang:unregister` in test cases by calls of the local version:

```
command(S) ->
  oneof([...{call,?MODULE,unregister,[name()]},...]).
```

With these changes, we can now rerun our tests—and they will fail for a *different* reason!

```
44> eqc:quickcheck(reg_eqc:prop_registry()).
.Failed! After 2 tests.
<... 35 lines omitted ...>
Shrinking......(6 times)
[{set,{var,1},{call,reg_eqc,spawn,[]}},
 {set,{var,3},{call,erlang,register,[a,{var,1}]}},
 {set,{var,4},{call,erlang,register,[a,{var,1}]}}]
History: [{{state,[],[]},<0.2101.0>},{{state,[<0.2101.0>],[]},true}]
State: {state,[<0.2101.0>],[{a,<0.2101.0>}]}
Res: {exception,{'EXIT',{badarg,[{erlang,register,[a,<0.2101.0>]},
                                 <... 7 lines omitted ...>]}}}
false
```

This time the failed test spawns a process, then tries to register it twice. We can see from the failure reason that one of the calls to `register` raised a `badarg` exception. The diagnostic information contains a history, in the form of the state before and result of each operation performed, and the final state after the last call—we can see that `spawn` returned the pid `<0.2101.0>`, the first call of `register` returned `true`, and that `{a,<0.2101.0>}` was in the model state after that call. It is not too big a stretch to guess that `register` is supposed to raise an exception when the name to be registered *is* already present in the registry, and so we now have enough information to refine our model further.

We will leave further model refinement to the exercises. For now, notice that the *same* property has revealed two *different* bugs—or rather, inconsistencies between the model and the real code. The ability to find different bugs using

the same property is a major advantage of property-based testing over a fixed set of test cases, in which one test case can hardly ever reveal more than one bug.

3.8 QuickCheck in Industry

In these notes, we show how to use QuickCheck to test parts of the Erlang standard libraries—relatively simple (and well-tested) code, in comparison to real industrial products. Yet the principles we have presented are equally applicable to testing real software.

In a typical industrial scenario, QuickCheck might be used to test a system that communicates with the outside world using one of the standardized telecom or internet protocols. In this case, a QuickCheck state machine is used to generate test cases containing commands that send a protocol message to the system under test, then wait for and check the response. Typically the protocol messages are large and complex, and a significant amount of work goes into contructing suitable generators. This work can be partially automated—tools exist to convert both ABNF and ASN.1 grammars into QuickCheck generators—but manual intervention is needed to generate protocol messages that "make sense", as opposed to random rubbish. The state machine then models just enough of the internal state of the system to generate sensible message sequences, and validate the responses. When a test fails, then QuickCheck finds a minimal failing sequence as usual. We shall illustrate this with a couple of examples of real bugs found using QuickCheck.

For example, in one project, QuickCheck was used to test a 3G radio base station for mobile telephony (RBS), by sending control messages using the NBAP protocol and observing the responses. An RBS maintains a number of radio channels which are used to exchange information with the mobile phones in each cell. During a call, there is a dedicated radio channel between the handset and the base station, but even handsets which are not currently making a call need to communicate with the base station occasionally. A single signalling channel is used for this purpose, shared between all the handsets in a cell. Now, every channel needs to be set up, and parameters such as the power level assigned—this is done when another product, a Radio Network Controller, sends NBAP commands to the base station telling it how to set up each channel. Note that *even though there is only one signalling channel, it still needs to be set up and configured.* Thus there is a command in the NBAP protocol for this purpose. Although this command should only be sent once, since there can only be one such channel, it is of course possible to send it to the base station *more than once*—and it is easy to imagine situations in which this might really happen, for example when a Radio Network Controller has crashed and is coming back online. If the signalling channel has already been set up, than an RBS is supposed to *reject* subsequent attempts to set it up again. We found this to be true for the software we were testing—almost always. But QuickCheck found a combination of parameters in the first and second set-up messages which caused the RBS to *accept* the second set-up command—and to create *two* signalling channels at the

same time. This violated a fundamental invariant of the base station software, leading within a few seconds to a hard failure, with the base station becoming unresponsive to all further commands. Interestingly, the first sign of trouble turned out to be Java exceptions in the base station log... which surprised us since we knew that the base station was implemented using C++! It transpired that Java had been used to build an operator GUI, which visualized the state of the base station, and this visualization was the first part of the software to crash when the signalling channel invariant was violated.

In another project, QuickCheck was used to test an Ericsson Media Proxy. This is a kind of firewall for multimedia IP-telephony (that is, carried using the internet protocol); it is one component of a Session Border Gateway, that isolates an operator's network from the internet at large. The Media Proxy opens and closes "media pinholes" to allow the media streams making up a call to pass the firewall. It is controlled by the H.248 or Megaco protocol, which defines commands to add and remove callers from a call (or "terminations" from a "context", in Megaco-speak). A call is created when the first caller is added, and deleted when the last caller is removed—and just as in the process registry tests, a call-ID is created by the first addition, and must be embedded in later messages to the Proxy concerning the same call.

The Megaco protocol supports any number of callers in a call, but the Media Proxy is restricted to just two. Nevertheless, QuickCheck found the following command sequence that provoked a crash:

- First, two callers are added to a call (the normal situation).
- Now the call is "full", and no further caller can be added—but one caller can be *removed*.
- Now the call is not full: a third caller can be added, and then removed again.
- The call is still not full; a fourth caller can be added and removed. On this final removal, something inside the Proxy crashes!

This sequence of seven commands was minimal for provoking this crash.

This case is interesting for several reasons. Firstly, because it cannot reasonably be found using manually constructed test cases—no tester in their right minds would test this case! After all, if adding and removing a caller works once, and works twice, then "by induction" it *must* work three times—right? Yet it does not. More seriously, because it is expensive to construct test cases manually, then it is impractical to test *all* sequences of seven commands, and there is no *a priori* reason to suspect that this one is worth testing. Coverage measures would not help here—it is likely that all the code, all the paths, and all the state transitions were already covered by the first five commands, so coverage measures would not indicate that we should go on to test the last two also.

Secondly, this example illustrates the tremendous power of shrinking—this minimal test failure was extracted from a random failing sequence of over 160

commands. The problem was tricky enough to diagnose from this minimal example; from a sequence of 160 commands, diagnosis would have been impossible.

Thirdly, this case will probably never, ever arise in practice—but the failed case is just the *symptom* of a fault, not the cause. The cause turned out to be corruption of internal data-structures on removing a caller from a call—and this corruption occurred *every time* a caller was removed. It just so happened that, in the normal case, when the only thing following the removal of the first caller was the removal of the second, then the data corruption passed unobserved. The fourth to the seventh commands in the failing test are needed just in order to convert the corrupted data into a crash. Even if it was not causing a problem at the time, it was well worth while discovering and correcting this corruption. After all, in a year's time another developer might well modify the code, and would naturally assume that the data structures inside such a tried-and-tested product were correct—but they were not. The corrupt data would have been a trap lying in wait for future developers, if QuickCheck had not revealed it and enabled the problem to be fixed.

Interestingly, QuickCheck was able to reveal these corrupt data structures *even though we did not know of their existence!* Had we known about them, and added code to the specification to check their invariants after each call, then the fault would have been found more quickly, and with a shorter failing test—only the first three commands would have been needed to provoke a failure. But this more detailed specification was not *necessary* to reveal the fault. This is a general observation—we find that surprisingly simple properties often suffice to reveal deeply-hidden bugs. Once software *begins* to go wrong, then there is often some continuation which makes it go very wrong indeed—and so testing basic properties can often reveal subtle bugs, if we only run enough tests. For this reason it is often not worthwhile to formulate a *complete* formal specification of the system under test, which can be quite expensive—we can trade off specification effort against testing time instead.

An interesting study of two larger projects at Erlang Training and Consulting, one developed with and one without using QuickCheck, appeared at the ACM Erlang Workshop last year.

3.9 Exercises

You have been provided with an incomplete specification of the process registry, in the file `reg_eqc.erl`. Inspect this file now. Note the line

```
-include_lib("eqc/include/eqc_statem.hrl").
```

at the top of the file, which is needed to make the QuickCheck state machine features available. Should you need to develop such a specification yourself, then a skeleton specification can be created by selecting "Complete eqc_statem spec" from the "State machine specs" menu in the QuickCheck Emacs mode.

1. **Adding a precondition**. Compile `reg_eqc.erl`, and use QuickCheck to test `prop_registry()`. You will find that the property fails, because

`register` raises an exception. Define a function `register_ok(S,Name,Pid)` which returns `true` of `register(Name,Pid)` ought to succeed in state `S`, and use it to specify a precondition for `register` that enables this property to pass.

Hint: the function `lists:keymember` may be useful.

2. **Adding a postcondition**. The previous exercise implements positive testing of `register`. Now we shall implement negative testing instead. First remove the precondition just added, and instead add a *postcondition* for `register` saying that an exception is raised precisely when `register_ok` is `false`. Revise your specification until `prop_registry()` passes.

3. **Testing whereis**. Add another postcondition to your specification, testing that `whereis` returns the correct result.

4. **Killing processes**. In this exercise, we shall investigate how the registry behaves when processes crash. Add the following definition to `reg_eqc.erl`:

```
stop(Pid) ->
  exit(Pid,kill),
  timer:sleep(1).
```

   This function kills the process referred to by `Pid` (and then waits a millisecond for it to die completely). Add calls of `stop` to your generated test cases, and retest `prop_registry()`. Since the registry documentation does not mention dead processes, then we would expect `stop` to have no effect on test case behaviour. Since the default clauses in `next_state`, `precondition` and `postcondition` say that `stop` may be called at any time, and has no effect, then we would expect all tests to pass. If this is the case, then you are finished with this exercise.

   If *not*, then your task is to revise the model so that the property passes again. You should do this *without* imposing restrictive preconditions on the operations, because the whole point of this exercise is to test what happens to the registry when relevant processes really do die.

   Hint: the function `lists:keydelete` may be useful.

5. **Dealing with non-determinism** This is an ambitious exercise, which you are likely to require help with—don't worry if you don't have time for it. Also, if you encounter behaviour you do not understand, ask for help.

   Taking your passing specification from the previous exercise, remove `timer:sleep(1)` from the definition of `stop`, and observe the effect on tests. Try to adjust your model so that tests pass again. You are likely to encounter evidence of non-deterministic behaviour, caused when the Erlang scheduler preempts your running test at an unpredictable point.

   To make any further progress, you need to recover deterministic testing. This can be achieved by a combination of techniques:

- Restrict your test cases to a maximum of 30 commands. This keeps them short enough to run to completion in one time slice.
- Add `?SOMETIMES` to your property, to permit occasional failures (read the documentation, to be found in module `eqc`).

Once your tests run deterministically, then failed tests should shrink to a test case that makes diagnosis possible. Finish adjusting your model.

Finally, extend your command generator to add calls of `timer:sleep(1)` to your test cases. Once again, you will need to adjust your model to make tests pass: having done so, you will have a good understanding of both the immediate, and the delayed effects of a process crash.

Having finished the exercises, why not select one of the Erlang library modules with a stateful interface, and construct a QuickCheck state machine specification of your own?

3.10 rec_eqc.erl

```
%%% File    : reg_eqc.erl
%%% Author  :  <John Hughes@HALL>
%%% Description : Process registry specification for CEFP
%%% Created : 21 May 2009 by  <John Hughes@HALL>

-module(reg_eqc).

-include_lib("eqc/include/eqc.hrl").
-include_lib("eqc/include/eqc_statem.hrl").

-compile(export_all).

-record(state,{pids=[],    % pids spawned in this test
       regs=[]     % list of {Name,Pid} in the registry
      }).

%% Initialize the state
initial_state() ->
    #state{}.

%% Command generator, S is the state
command(S) ->
    oneof([{call,erlang,register,[name(),pid(S)]} || S#state.pids/=[]] ++
  [{call,?MODULE,unregister,[name()]},
   {call,erlang,whereis,[name()]},
   {call,?MODULE,spawn,[]}
   ]).

-define(names,[a,b,c,d]).

name() ->
    elements(?names).

pid(S) ->
    elements(S#state.pids).

%% Next state transformation, S is the current state
next_state(S,V,{call,_,spawn,_}) ->
    S#state{pids=[V|S#state.pids]};
next_state(S,_V,{call,_,register,[Name,Pid]}) ->
    S#state{regs=[{Name,Pid}|S#state.regs]};
next_state(S,_V,{call,_,unregister,[Name]}) ->
    S#state{regs=proplists:delete(Name,S#state.regs)};
next_state(S,_V,{call,_,_,_}) ->
```

```
    S.

%% Precondition, checked before command is added to the command sequence
precondition(_S,{call,_,_,_}) ->
    true.

%% Postcondition, checked after command has been evaluated
%% OBS: S is the state before next_state(S,_,<command>)
postcondition(S,{call,_,unregister,[Name]},Res) ->
    case Res of
{'EXIT',_} ->
    not unregister_ok(S,Name);
true ->
    unregister_ok(S,Name)
    end;
postcondition(_S,{call,_,_,_},_Res) ->
    true.

%% The conditions under which operations ought to succeed.

unregister_ok(S,Name) ->
    proplists:is_defined(Name,S#state.regs).

%% The main property.

prop_registry() ->
    ?FORALL(Cmds,commands(?MODULE),
    begin
{H,S,Res} = run_commands(?MODULE,Cmds),
cleanup(),
?WHENFAIL(
   io:format("History: ~p\nState: ~p\nRes: ~p\n",[H,S,Res]),
   Res == ok)
    end).

cleanup() ->
    [catch erlang:unregister(Name) || Name <- ?names].

%% Exception-catching versions of the API under test

unregister(Name) ->
    catch erlang:unregister(Name).

%% Spawn a dummy process to use as test data. Processes die after 5
%% seconds, long after the test is over, to avoid filling the Erlang
%% heap with dummy processes.
```

```
spawn() ->
    spawn(timer,sleep,[5000]).
```