Quviq QuickCheck

Getting Started

Thank you for your interest in Quviq QuickCheck! QuickCheck is a powerful automated testing tool, which combines random test case generation with automated diagnosis of faults once they are found. This document aims to get you running quickly. It is not a reference manual, but it will show you how to test several different kinds of software using QuickCheck. Try out the examples herein, and play a little, and you should be well on the way to using QuickCheck for real. The sections that follow will show you how to

- Add QuickCheck to your Erlang installation
- Define test data generators, and use them to test pure functions
- Test systems with an internal state
- Test code in other programming languages

How to Install QuickCheck

First of all, find your Erlang installation. Under Windows, for example, this might be in C:\Program Files\erl5.5.5. Once you have found your installation, you will see that it contains a sub-directory called lib. This sub-directory contains a collection of packages with names such as crypto-1.5.1.1, and it is here that QuickCheck will be installed.

In this download, you will see a folder called eqc-1.14. Copy this folder into the lib directory.

Now start an Erlang shell, and type eqc:start(). If you see a message such as

```
Starting eqc trial version 1.147 (compiled at {1194,208520,890000})
```

then QuickCheck is correctly installed.

The eqc-1.14 directory contains a subdirectory called doc, containing a file index.html. This is the QuickCheck reference manual, in edoc form. Don't try to read it yet, but why not bookmark it in your browser, so that you can refer to it easily as you play with QuickCheck?

Introducing QuickCheck Properties

When you test using QuickCheck, you don't write test *cases*, you write *general properties* that your code should always satisfy. QuickCheck generates test cases from your properties, runs tests, and reports counterexamples when they are found. You can think of writing a QuickCheck property as writing *many test cases at once*—or you can think of a property as a (partial) formal specification. Either way, you can run a lot of tests for little effort.

Let's get started by testing a simple library function—lists:reverse. We will test the well-known property that

```
lists:reverse(lists:reverse(Xs)) == Xs
```

This is far from a complete test of the reverse function, but it is certainly likely to reveal many possible faults.

Defining a Property and Running Tests

QuickCheck properties are just Erlang definitions using the QuickCheck API, so we'll need to create a module to contain one. Let's call it reverse_eqc. We'll need to include QuickCheck's header file to make the API available.

```
-module(reverse_eqc).
-compile(export_all).
-include_lib("eqc/include/eqc.hrl").
```

Now to define the property itself. We have to choose the kind of test data we want—let's use lists of integers. The property can then be written as

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

Here `list(int())` is a *test data generator* which specifies how test cases should be created. The `?FORALL(Xs,…)` just binds `Xs` to the generated data, and the third argument of `?FORALL` checks the property we want to test. Copy this definition into your file, and compile it.

Now you can test the property by calling QuickCheck in an Erlang shell:

```
4> eqc:quickcheck(reverse_eqc:prop_reverse()).
.....................................................................
.............................
OK, passed 100 tests
true
```

Each dot represents a successful test—so indeed, the property seems to be true.

Collecting Statistics on Test Cases

One of the unfamiliar things about using QuickCheck is that we don't actually see the test data that is used—and indeed, we don't want to, because there is so much of it. But we can *collect statistics* about the test data, by instrumenting the property as follows:

```
prop_reverse() ->
    ?FORALL(Xs,list(int()),
            collect(length(Xs),
                    lists:reverse(lists:reverse(Xs)) == Xs)).
```

Try testing this—you'll see the distribution of list lengths after testing is complete. It's a good idea to keep an eye on the distribution of test data, so that you don't unwittingly run a large number of very trivial tests.

Note that the call of collect *encloses* the rest of the property—the boolean expression we're testing is the second argument of collect. This is a pattern we'll see repeatedly with QuickCheck.

Failing Tests and Shrinking

Successful tests are fun, but it's more interesting to see what happens when a test fails. So let's sabotage the property now by deleting one call of reverse:

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

Of course, testing will now fail. Try it!

```
6> eqc:quickcheck(reverse_eqc:prop_reverse()).
..........Failed! After 11 tests.
[1,3]
Shrinking...(3 times)
[0,1]
False
```

What has happened here is that QuickCheck finds a failing test case, and prints it—the list [1,3], which is not its own reversal. But then QuickCheck does something rather interesting—it starts to simplify the failing case as far as possible, in this case reducing the elements to 0 and 1. Every dot printed after "Shrinking" represents a successful simplification step. Interestingly, no elements are discarded from the list—which tells us that *both elements are necessary* for the test case to fail.

Generating Different Kinds of Test Data

QuickCheck makes it very easy to generate different kinds of test data. Try replacing `list(int())` in the property above by `list({int(),int()})`, `list(list(int()))`, and `list(char())`. You might even want to see what happens if you replace it by `int()`—the wrong kind of test data for reverse.

If you want to bind several variables, just generate a tuple. For example, try the following property:

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

We just generate a pair of lists, and match them against the pattern {Xs,Ys} to bind two variables. Notice how easy it is to generate pairs of lists—we just embed two list generators in a pair. This works in general: QuickCheck will find generators embedded in any data-structure, and use them to generate data-structures of the same shape with random components. By the way, the property above is wrong… but you should find it easy to test and fix it.

Take a look at the QuickCheck documentation now. Module eqc_gen provides a rich API for defining generators (with fine control over the distribution). Module eqc provides ?FORALL, collect, and a number of other useful functions on properties. (We'll cover module eqc_statem in the next section). Remember, *properties* and *generators* are the two abstract types that QuickCheck is built upon.

Summary

Using the functions in eqc_gen, you can generate any kind of test data you like, and use it to test virtually any code, provided you can think of suitable general properties to test. This is easiest for pure functions without side-effects… for example, encoding and decoding functions. The simple property

```
prop_decode_encode() ->
    ?FORALL(Msg,message(),
            decode(encode(Msg)) == Msg).
```

can be used to find surprisingly many bugs! Why not try this idea out on an encoder and decoder of your own? You can probably think of many other examples you would like to test in this way.

When testing an API with an internal state, though, more complex properties are needed. Read the next section to try out an example of this sort!

Testing Systems with Internal State

When we test an API with an internal state, we need to run more complex tests than we've seen so far. Typically, we want to run sequences of operations of various kinds, that take the system through a succession of state transitions. QuickCheck provides a *state machine* library, eqc_statem, which can be used to generate random sequences of calls. It also simplifies failing cases to a shortest sequence of calls that provokes the failure—a very powerful feature that usually makes bugs easy to find.

A Simple Example

As an example, we'll test the Erlang local process registry. To keep things simple, we'll just test the operations register, unregister, and whereis, and we'll restrict registered names to a small set of atoms. We'll need to include one more operation in our tests, to spawn a dummy process, so that we have some clean pids available to register.

In the QuickCheck Tutorial folder, you'll find a file called reg_eqc.erl. This is the beginnings of a state machine specification of the registry—you can look at the code while you read on in this document.

Defining a State Machine

The first thing to note is that we include another header file, providing the state machine testing operations. Every state machine specification also exports a collection of call-backs, which QuickCheck uses to exercise the state machine.

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

-export([command/1, initial_state/0, next_state/3,
         precondition/2, postcondition/3]).
```

Then we define a record to track the state of a test case. This needn't model the *entire* state of the system under test—it just needs to contain enough information to tell us which calls are valid at each point in a test, and whether or not the results returned are correct. In this case, we'll need to keep track of which processes have been spawned in the test case, and also what names and pids have been registered.

```
-record(state,{pids,          % list of spawned pids
               regs}).        % list of registered names and pids
```

At the same time, we can define the initial state we expect each test to start in.

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

Specifying State Transitions

The *effect* of each call is specified by the next_state function, which maps the current state, the result of a call, and a symbolic representation of the call (module—function name—arguments) into the expected state after the call.

```
next_state(S,V,{call,?MODULE,spawn,[]}) ->
    S#state{pids=[V | S#state.pids]};
next_state(S,V,{call,erlang,register,[Name,Pid]}) ->
    S#state{regs=[{Name,Pid} | S#state.regs]};
next_state(S,V,{call,erlang,unregister,[Name]}) ->
    S#state{regs = lists:keydelete(Name,1,S#state.regs)};
next_state(S,V,{call,erlang,whereis,[Name]}) ->
    S.
```

Note that the case for spawn adds the newly spawned pid to the list of pids in the state; the other cases ignore the actual result of the call, and just update the list of registered names and pids as we would expect. This is a very simple and abstract specification of how we expect the registry to behave.

Generating Calls to the Registry

We also need to specify how calls should be generated, which is done by the function command.

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

This function is given the current state as a parameter, and should generate a command that is possible in that state. We've used the QuickCheck function oneof to choose between a list of alternatives—each alternative being a symbolic call to a different function in the API under test. Here name() is a generator we've written which chooses a random atom from a small set:

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

The function elements used here just chooses an element from a list. We use a small set of names so that there is a high probability of choosing the *same* name in several different calls in a test case. Otherwise, the majority of our tests would be rather boring!

Conditional Alternatives

Notice that when we generate a call to register, we not only choose a random name, we also choose a random pid from the list kept in the state. But we can only do this if there *is* at least one pid in the state. In other words, if S#state.pids is the empty list, then we cannot generate a call to register, and this alternative should not be included in the list we pass to oneof. That's what the strange notation

```
[{call,erlang,register,[name(),elements(S#state.pids)]}
 || S#state.pids/=[]] ++
```

achieves—this is actually a *list comprehension without a generator*, which evaluates to the empty list if S#state.pids is empty, and a singleton list containing a call to register otherwise. The compiler issues a warning for list comprehensions of this sort, but just ignore it: this is perfectly valid code, and is by far the most concise and readable way to achieve the effect we want.

Defining the Top-Level State Machine Property

The other call-backs can be stubbed for now: all we really need is a property to test. That's found towards the bottom of the file:

```
prop_registration() ->
    ?FORALL(Cmds,commands(?MODULE),
            begin
              {H,S,Res} = run_commands(?MODULE,Cmds),
              [catch unregister(N) || N<-?names],
              Res==ok
            end).
```

Here Cmds is a list of commands, generated by the QuickCheck function commands using the call-backs in ?MODULE. The actual test is a little more complex than we've seen earlier: we use QuickCheck's run_commands function to actually run the commands, and check that the Res part of its result is ok. (The other two components H and S contain more information about the history of the test, useful to see when a test fails, but not of importance just now). Finally, since running a test may leave the registry in an unknown state, we include the line

```
[catch unregister(N) || N<-?names],
```

to clean up any processes we left registered, so that the next test will also start in the right initial state.

The Results of Testing

That's all we need to start testing—although this specification is woefully incomplete. Try testing prop_registration() now. You'll see something like this:

```
6> eqc:quickcheck(reg_eqc:prop_registration()).
Failed! After 1 tests.
…
Shrinking...(3 times)
[{set,{var,3},{call,erlang,unregister,[a]}}]
false
```

where the … is an *unsimplified* failing list of commands—usually long and uninteresting. Ignore it.

The final shrunk failing case just consists of a single symbolic command, which sets variable 3 to the result of calling unregister. (As you see, the results of calls are always bound to numbered variables, which can be reused in the arguments of later calls). In this case, it's obvious what the problem is: we

called *unregister* without first calling *register* on the name a, so of course, unregister raises an exception---and the test fails.

Specifying Preconditions

The problem here is not with the code under test, but with our specification, which says that *any* call of unregister is allowable. How can we correct it? One way is to specify that unregister has a *precondition*—that it should not be called unless the name has previously been registered. We can do so by adding a clause to the precondition callback:

```
precondition(S,{call,erlang,unregister,[Name]}) ->
    lists:keymember(Name,1,S#state.regs);
```

Luckily, our state S contains enough information to determine whether or not a call of unregister should succeed. Add this clause to the specification, and test it again. QuickCheck will now restrict the test cases generated so that unregister's precondition is always satisfied, and as a result, the error we just saw should disappear. However, you'll immediately find another problem. Try to fix the new problem in the same way, and continue adding preconditions until testing succeeds.

Specifying Postconditions

What we've done here is *positive testing*—we have restricted test cases to calls that should be valid. But what if we want to test negative behaviour too—that unregister raises an exception precisely when it ought to? We can do that too, by removing the precondition and writing a *postcondition* instead. The postcondition will just check that the result is an exception if the name passed to unregister is *not* already registered in the state:

```
postcondition(S,{call,_,unregister,[Name]},R) ->
    case lists:keymember(Name,1,S#state.regs) of
          true -> R==true;
          false -> {'EXIT',_}=R, true
    end;
```

Here the third parameter R is the actual result that unregister returned: we just check that it's true if the Name was previously registered, and an exit value otherwise.

There's just one catch: if unregister raises an exception, then the test will be considered to have failed anyway, no matter that the postcondition says. To avoid this, we need to define a local version of unregister that catches the exception:

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

We also have to change the command generator and the next_state function to refer to this local version instead. Once we've done so, we can test the property again—and once again, move on to the next error. Try to complete the specification again using only *postconditions*, so that no further errors occur. This is a little trickier than for preconditions!

Reflections and Extensions

Of course, we haven't found any actual *bugs* in the registry… but then again, it would be surprising if we had! What we have seen, is that *inconsistencies* between the specification and the code are rapidly identified and diagnosed using QuickCheck. An inconsistency may be due to a bug in the code, or an erroneous specification—a misunderstanding, in other words. In either case, it's valuable to find and fix it.

For more fun, add a command to these test cases to *kill* processes at random points in a test case. Do so by generating calls to the following function:

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

You will need to extend the call-backs to handle this new kind of command. When you extend the next_state function, don't remove killed pids from the list of available pids in the state. That would prevent QuickCheck trying to register any dead processes, which is precisely what we want to test! If you find you need to know whether processes are alive are dead, then you should add a list of killed pids to the state to keep track of it.

You may be wondering: why the call to yield in the definition above? Because it turns out that killing a process triggers actions that take a little time to complete, and calling yield allows them to do so, which keeps things simple. Once you have a specification that QuickCheck finds no more errors in, try removing the call to yield. The effect is staggering!

Testing Foreign Language Code

Nothing restricts QuickCheck to testing code written in Erlang. Provided we have a way to *invoke* foreign language code, then we can use Erlang to express a functional model and QuickCheck to generate and simplify test cases, just as we do when testing Erlang code. Of course, there are many ways to invoke foreign code from Erlang, but in this case we can get by with the very simplest: we can generate *source code* in the foreign language, which we compile and run in each test. This can be a little slow, since we must compile and link in each test, but it has the benefit of great simplicity, and applicability to any programming language. In this section, we'll see how to apply this idea to testing C code.

A Simple Example

Once again, we will take library functions as an example API to test, because they are available no matter what C compiler you are using. We'll test part of the file I/O API: the functions fread, fwrite, feof (which tests for end of file), and fseek (which sets a position in a file at which the next read or write will occur). For simplicity, we'll test only byte input/output—even so, unless you are an expert C programmer, you're in for a few surprises!

In the Tutorial folder, you will find another folder called CFileIO: take a look at the file cfileio_eqc.erl, which contains an (inaccurate) specification of these functions.

Modelling the Contents of a File

The operations we are testing read and write sequences of bytes at arbitrary positions in a file, we start off by modelling these operations on lists. Reading is modelled by the function extract:

```
extract(L,Pos,Len) ->
    {_,X} = split(Pos,L),
    {Res,_} = split(Len,X),
    Res.
```

which returns a list of Len bytes from the list L, starting at position Pos—truncated to a shorter length if L is shorter than Pos+Len bytes. Writing is modelled by the function insert, which replaces the sequence of bytes at position Pos in list L, extending L with null bytes if we try to insert at a position beyond its end.

```
insert(L,Pos,Data) ->
    {Pre,X} = split(Pos,L),
    {_,Suf} = split(length(Data),X),
    extend(Pre,Pos,0)++Data++Suf.

extend(L,N,X) ->
    case length(L) < N of
          true -> L ++ [X || _ <- lists:seq(1,N-length(L))];
          false -> L
    end.
```

(We define our own version of the lists:split function, because the function in the lists library raises an exception if we try to split at position zero).

Modelling a C File Stream

These functions comprise a very simple model of the effect of reading and writing bytes to a file. But a C file stream contains more information than just a sequence of bytes: it also records *where* in the file the next read or write should take place. So we model the state of a C file stream by a record, with both a contents and a current position.

```
-record(c_file,{contents=[],position=0}).
```

We'll generate test cases which just read and write a single file; this means we need to track the state of each test case, but a c_file record contains enough information to do so. So we can use QuickCheck's state machine module again, using a c_file as the state.

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

Generating Calls to the C Functions

We write a command generator just like the one for the process registry, but now the commands we generate will be calls of the C functions under test:

```
command(S) ->
    oneof([{call,c,fread,[size()]},
           {call,c,fwrite,[list(noshrink(choose(0,255)))]},
           {call,c,fseek,[pos()]},
           {call,c,feof,[]}])}]).
```

There are several points to note here:

- QuickCheck's symbolic commands are designed to represent calls of Erlang functions, so each call contains both a module name and a function name. Since we're now going to interpret these as C calls, the module name is irrelevant—we just use the name c as a place-holder.
- Each C function has several additional parameters not shown here, which will be filled in when we translate these calls to C source code. We only need to generate the parameters that *vary* from call to call. These are:
    - *fread*: the number of bytes to read.
    - *fwrite*: the list of bytes to write.
    - *fseek*: the position to seek to.
    - *feof*: none.
- When generating calls to fwrite, we applied `noshrink` to the generator for the bytes to be written. The effect is to disable shrinking of these values. This isn't necessary, but it is helpful. We do so partly because replacing one byte value by another is hardly a big simplification, but mainly because we expect rather many bytes to be written in each test. Minimizing all of these values could force shrinking to run a very large number of tests. By introducing noshrink, we trade off faster shrinking against the degree of minimization.

Specifying the State Transitions

Now we can specify the *effect* of each command via the next_state function, as before:

```
next_state(S,_,{call,c,fread,[Size]}) ->
    #c_file{contents=L,position=Pos} = S,
    S#c_file{position = Pos + length(extract(L,Pos,Size))};
next_state(S,_,{call,c,fwrite,[Data]}) ->
    #c_file{contents=L,position=Pos} = S,
    S#c_file{contents=insert(L,Pos,Data),
             position=Pos+length(Data)};
next_state(S,_,{call,c,fseek,[Pos]}) ->
    S#c_file{position=Pos};
next_state(S,_,_) ->
    S.
```

There are no surprises here, really, except perhaps to note that for fread, we are careful only to advance the file position by the number of bytes actually read. This is important when we reach the end of the file. Once again, what we have here is a simple abstract specification of how the operations we are testing are supposed to behave.

The Skeleton C Program

But how do we actually *run* these commands? They are not Erlang functions, so we cannot use run_commands: instead, we have to compile them to C source code. Look in the CFileIO directory again: you'll see a file main.c, which is actually the program compiled and run in each test. It contains the following code:

```
#include <stdio.h>
#include "macros.h"

main()
{ char buffer[1000];
  FILE* stream = fopen("data.txt","w+b");
  int i, n;
  open_eqc();
  #include "generated.c"
  close_eqc();
  fclose(stream);
}
```

As you can see, this is a program skeleton into which calls generated by QuickCheck can be inserted by writing them to the file generated.c. Otherwise, this code declares a buffer and some integer variables that will be needed by the generated calls, and opens a file stream that will be used for the test. The calls `open_eqc()` and `close_eqc()` open and close a file `to_eqc.txt`, which is used to return results to QuickCheck. They are defined in the header file `macros.h`, along with some useful macros.

Compiling Test Cases to C

Returning to the QuickCheck specification, let's look at the code that writes generated.c. It's found in the compile function, which writes the corresponding C code for each command that QuickCheck generates. For example, calls of fseek are compiled as

```
compile(C,{call,c,fseek,[Pos]}) ->
    io:format(C,"INT(fseek(stream,~w,SEEK_SET))",[Pos]);
```

The `INT(…)` surrounding the call is a macro that writes the result as an integer to `to_eqc.txt`, so that QuickCheck can retrieve it. As a more complex example, calls of fread are compiled as

```
compile(C,{call,c,fread,[Size]}) ->
    io:format(C,"TUPLE(INT(n=fread(buffer,1,~w,stream));"++
                    "LIST(for(i=0;i<n;i++) INT(buffer[i])));",
              [Size]);
```

The call of fread reads up to Size bytes into `buffer`, returning the number of bytes actually read. The result of fread is sent back to QuickCheck as an integer, as are `n` elements of the buffer. The buffer elements are enclosed in a `LIST(…)` macro, which makes QuickCheck see them as a list of elements. The result of fread, and this list, are themselves enclosed in a `TUPLE(…)` macro, which makes QuickCheck see them as a pair. Thus the result QuickCheck sees from a call of fread might be something like {3,[1,2,3]}. Using these macros, it is easy to compile C code that transmits a rich variety of Erlang terms back to QuickCheck.

Specifying Postconditions

Now we know how the return values appear to QuickCheck, we can write postconditions that check that the C functions are behaving as we expect. Here are the postconditions you will find on the CD: a first stab at a specification that you will need to refine to make testing succeed.

```
postcondition(S,{call,c,fread,[Size]},V) ->
    {N,Data} = V,
    #c_file{contents=L,position=Pos} = S,
    N == length(Data) andalso Data == extract(L,Pos,Size);
postcondition(S,{call,c,feof,[]},V) ->
    #c_file{contents=L,position=Pos} = S,
    (Pos > length(L)) == (V /= 0);
postcondition(_,{call,_,_,_},_) ->
    true.
```

We just check that the data returned by fread is the data we expected to find in the file, and that feof returns true exactly when the file position lies beyond the file contents.

The Top-Level Property

Before we can start testing, we need a property to test. Here it is:

```
prop_cfileio() ->
    ?FORALL(Cmds,commands(?MODULE),
            begin
              Vals = run(Cmds),
              ?WHENFAIL(io:format("~p~n",[Vals]),
                        postconditions(?MODULE,Cmds,Vals))
            end).
```

This is very like the property we used to test the process registry, the main difference being that we cannot use run_commands to run the list of commands that is generated. Instead we use a function of our own, run, that translates the commands to C source code, compiles and runs them, and reads the list of return values back into Erlang. QuickCheck's `postconditions` function is then used to check that all the post-conditions hold. The `?WHENFAIL` macro is optional: we are just using it to print out the list of return values together with a failing test case, which makes diagnosing test failures easier.

Compiling and Running the Generated C Code

Finally, let's look at the run function itself:

```
run(Cmds) ->
    {ok,C} = file:open("generated.c",[write]),
    [compile(C,Call) || {set,_,Call} <- Cmds],
    io:format(C,"~n",[]),
    ok = file:close(C),
    os:cmd("c:\\cygwin\\bin\\tcsh < runtest.csh"),
    file:delete("data.txt"),
    {ok,Vals} = file:consult("to_eqc.txt"),
    Vals.
```

The first four lines just write the test case to generated.c, the next two lines run and clean up after the test, and the last two read the list of return values back into Erlang. Simplicity itself. You will need to adapt the fifth line to reflect the way you invoke your C compiler; in our installation (Cygwin under Vista), the file runtest.csh contains the two commands

```
gcc main.c
./a.exe
```

Refining the Specification

Once you have adapted the code to fit your setting, compile the specification and start running tests! Of course, you will find that testing fails. The first error found on our system is usually this one:

```
17> eqc:quickcheck(cfileio_eqc:prop_cfileio()).
.....Failed! Reason:
{'EXIT',postcondition}
After 6 tests.
…
Shrinking..(2 times)
Reason:
{'EXIT',postcondition}
[{set,{var,1},{call,c,fread,[1]}},{set,{var,2},{call,c,feof,[]}}]
[{0,[]},1]
false
```

As you can see, feof returned true (1), but QuickCheck expected it to return false. Funnily enough, the test does *not* fail if the call to fread is removed… even though this call reads zero bytes! See if you can refine the QuickCheck specification so that all tests pass. Along the way, you'll learn more than you imagined there is to know about C file I/O!

Going Further

We hope this tutorial has whetted your appetite for QuickCheck, and shown you some of the things that it can do. To take it further, and start using QuickCheck for real, you can obtain full licences from Quviq AB, together with training courses and expert services to help you formulate QuickCheck specifications of your real systems. Contact sales@quviq.com for more information.

What Customers Say

”When developing my SIP stack, I used QuickCheck in parallel with programming the encoder/decoder (3,300 lines of code), and as a result no bugs at all have been reported in that part of the code! In contrast, bugs have been reported in another part of the code, 600 lines developed without using QuickCheck. When I finally tested this part with QuickCheck too, I found a couple of new bugs that traditional testing had missed! The comparison is interesting, because all the code was written by the same programmer (me!), and tested by the same testers.”   *Hans Nilsson, IMS Gateways, Ericsson, Stockholm*
"Using QuickCheck, we have doubled the number of bugs found in the early stages of testing and more than halved the number of faults that slip through to the customer acceptance testing. This results in less faults reported back to us, and a more confident customer."   *Francesco Cesarini, CTO Erlang Training and Consulting, London*
It’s just amazing how QuickCheck is changing my mindset on Quality Assurance. Code quality has always been important to me, and I've automated my testing to the best extent possible. However, exploratory testing after unit and integration tests is always a must –you always find some more bugs that way. QuickCheck introduces an interesting dynamic by virtue of its ability to *do* exploratory testing. It's like having a Quality Assurance department in my back pocket!   I've really, really enjoyed working with QC. I just can't say enough good things about it.   *A Happy Customer, Boston*