QuickCheck Exercises

1. List deletion.

The function lists:delete(X,Xs) removes an element from a list. Formalize a property prop_delete() expressing the fact that a deleted element is no longer present in the list. Test it with QuickCheck,--we expect it to fail.

Useful functions

- lists:member(X,Xs)—tests whether X occurs in the list Xs
- **?FORALL**(X,Gen,Body)—a property that is true if the Body holds for every X generated by Gen
- int()—a generator for (small) integers
- list(Gen)—a generator for lists of elements generated by Gen
- {Gen1,Gen2}—a generator for pairs {X1,X2}, with Xi generated by Geni
- eqc:quickcheck(Property)—tests a property
- eqc:numtests(N,Property)—sets the number of tests that eqc:quickcheck will run

Correct the property so that it passes a large number of tests.

Useful functions

- **?IMPLIES**(Condition,Body)—a property which is true if Body holds whenever Condition is satisfied

How often is the element you delete actually a member of the list you are deleting it from? Find out by instrumenting your property to gather statistics.

Useful functions

- collect(Value,Body)—a property equivalent to Body, which displays statistics over the values of Value

Useful functions

- elements(List)—a generator that chooses one of the elements of List

Modify your property to *guarantee* that the element deleted is actually present in the list.

2. The process registry

In this exercise you will develop a specification of one of the components of the Erlang system—the process registry. Erlang processes are created by spawn(Fun), which returns a dynamically allocated *process identifier*. It is impossible to communicate with a process without its process identifier, so to

Useful functions

- **?WHENFAIL**(Action,Property) is logically equivalent to Property, but performs Action while reporting a failed test
- eqc:check(Property)—tests a property in the last test case to fail, rather than generating a new one

enable processes to find each other Erlang provdes the registry—a kind of local name server. The three operations we will test are

- register(Name,Pid)—register the process Pid under the name Name, which must be an atom
- unregister(Name)—remove any registration for the given name
- whereis(Name)—return the Pid registered with Name

Study the file **`reg_eqc.erl`**, which contains an incomplete state-machine specification of these operations. Your task will be to complete the specification—thus using QuickCheck as a *program understanding tool*. The end result will be a precise—and testable—formal specification of the registry's behavior. Of course, we do not expect to find bugs in such a central component of the Erlang system—but we will make some surprising discoveries. When applying QuickCheck to less well tested systems, getting the specification right in this way is the prelude to finding real bugs.

The state of the registry is modeled by an Erlang record, containing a list of process identifiers spawned during the test case, and a list of name-pid pairs which should currently be held in the registry. The initial_state() function is a *call-back* used by QuickCheck to initialize the model state at the start of each test. The command(S) call-back specifies how operations to include in the test should be generated from the model state. These operations are represented as *symbolic function calls*, of the form {call,ModuleName,FunctionName,Arguments}. The model state transitions are defined by the call-back next_state(State,Result,Call), which updates the State to reflect the execution of Call, with the given Result. The main property, prop_registration(), states that all command sequences generated using call-backs in this module can be executed with a successful result (Res==ok). It includes clean-up code to ensure each test leaves the registry in a known state, and code to report additional information when displaying a failed test.

The QuickCheck Emacs mode

- There is a short-cut for copying a function clause, often useful when new clauses are to be added to functions like precondition. Place the cursor in an existing clause, and type CTRL-Q CTRL-F c. The clause will be copied, and you can then edit the copy to create the new clause you want.

Useful functions

- lists:keymember(Key,Index,List) returns true if the Key is found at position Index in one of the tuples in the List

Positive testing

Use QuickCheck to test prop_registration(). It should fail, with a message resembling

```
[{set,{var,1},{call,erlang,unregister,[a]}}]
Reason: {exception,{'EXIT',{badarg,[{erlang,unregister,[a]},
                                    {eqc_statem,f515_0,5},
                                    {eqc_statem,f507_0,5},
                                    {eqc_statem,run_commands,2},
                                    {reg_eqc,'-prop_registration/0-
fun-2-',1},
                                    {eqc,'-f778_0/2-fun-4-',3},
                                    {eqc_gen,'-f321_0/2-fun-0-',5},
                                    {eqc_gen,f186_0,2}]}}}

unregister(a) -> no_return
```

The problem is, of course, that no process is registered under the name **a**, and so unregister(a) raises an exception. Amend the specification by adding a *precondition* specifying that unregister may only be called with a name that is present in the registry.

Precondition call-back

- precondition(State,Call) is used to define a precondition for the symbolic Call in a given model State. Symbolic calls generated by command(S) are not used in tests unless their precondition is true.

Retest your previous counterexample and ensure that your precondition now excludes this case.

Test the property again, and continue adding preconditions until your tests pass.

Negative testing

Interpreting the result of run_commands

- run_commands(Cmds) returns a complete history of the execution of the command sequence, including all the model states. The result is {History,FinalState,Res}, where
  - History is a list of {ModelState,Result} pairs, giving the ModelState before each call in the Cmds, and the actual Result returned by the command
  - FinalState is the model state after the last command
  - Res is ok if there were no exceptions and all postconditions were true, otherwise a value indicating why the test failed
- By printing the History in a **?WHENFAIL**, one can observe and debug problems in the model's state

This is an example of *positive testing*, in which we deliberately restrict ourselves to using the API under test correctly. Now we shall proceed with *negative testing*, in which we also test calls that raise exceptions—and we will check that those exceptions are raised correctly.

Comment out the preconditions you added to the specification, and modify the command generator to generate calls to ?MODULE:register(Name,Pid) and ?MODULE:unregister(Name), rather than erlang:register(Name,Pid) and erlang:unregister(Name). These locally defined versions just catch the exception. Your tests should now pass, but only because the exceptions are caught and ignored.

When an Erlang exception is caught, the catch expression returns a tuple of the form {'EXIT',Reason} (where 'EXIT' is enclosed in quotes because it is an atom that begins with a capital letter). Add a *postcondition* to the specification to check that unregister(Name) raises an exception if Name is not present in the registry.

Add a postcondition for register to check that it also raises an exception exactly when it should. You will encounter a more subtle failure, which requires a little more information to debug. Negative testing exposes the system under test to a wider range of stimuli, and so can provoke unexpected behaviours.

Postcondition call-back

- postcondition(State,Call,Result) is used to determine whether a Result returned by Call is consistent with the model State. When run_commands(Cmds) returns a Res of ok in prop_registration(), this means that all postconditions were true when Cmds were executed.

Correct your specification to reflect the actual behavior of the registry.

Processes which crash

Sometimes Erlang processes crash, including registered processes. To simulate process crashes, we can include calls of stop(Pid) in our tests. Amend the command generator to include calls of ?MODULE:stop(Pid), for pids in the model state. Like register(Name,Pid), stop(Pid) can only be generated if there *are* pids in the model state, so you will need to include the stop alternative conditionally—see the generation of register for how to do this.

Useful functions

- **?SOMETIMES**(N,Property) retries Property up to N times; if any test passes, then the **?SOMETIMES** passes

Once stop is included, your tests will fail. Extend the model state to track dead processes, and adjust your specification so that the tests pass again.

Note: one way to do so is to add preconditions excluding dead processes as arguments. Of course, you should not do this! The whole point is to *allow* dead processes as arguments, and figure out how the specification of the registry changes when this is done.

Non-determinism

The stop function provided works in two stages: it first kills the target process, then waits one millisecond (timer:sleep(1)) to allow the registry to react. This avoids any race conditions between the execution of the registry and the remainder of the test. Remove the call to timer:sleep and observe the effect.

Amend your specification to correspond to the new behavior of the registry. Because there is now a race condition in your tests, it's likely that tests will occasionally fail anyway through sheer bad luck, making it difficult to verify your model. In this case, you can *retry* each test a few times to ignore such occasional failures. You will also need to restrict the length of your test cases, because a sufficiently long test will always lose the race with the registry.

Useful functions

- lists:keydelete(Key,Index,List) removes the first tuple in List in which the Key is found at position Index, or returns List unchanged if there is no such tuple

Once your tests are passing reliably, add calls of timer:sleep(1) to the generated test cases. This will allow you to specify what the registry does in the background, because it will "catch up" during the call of sleep.

In conclusion

We hope you have enjoyed this foray into property-based testing, property-driven development, and specification development. We hope you have learned that a specification is not correct just because it is *called* a specification—it's just program code, and as likely to be wrong as any other program code. But QuickCheck does not just test a program against a specification—it also tests the specification against the program! A failed test represents an *inconsistency* between the two, but the fault may be in either one. When using QuickCheck to test existing products, the specification is the *new* code, and thus usually the buggiest. But once the specification bugs are ironed out, then quirks in the system under test start to appear, and then the effort spent on the specification is often rewarded by the discovery of extremely subtle bugs in the implementation.

We hope you have been convinced that property-based testing is both fun and highly productive. Enjoy!