Using eqc_fsm to test a finite state machine: an example

John Hughes, Quviq AB

March 16, 2009

1 Introduction

This is a simple example to demonstrate the use of eqc_fsm, which can be used as a starting point for other fsm specifications. We define a simple "locker", which manages a proplist, with an API which provides read, write, lock, and unlock operations. To keep things simple, this module contains *both* the implementation and the specification of the locker.

```
-module(lock_eqc).

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

2 The Locker Implementation

The locker is implemented using `gen_fsm`, so this section defines the call-backs that `gen_fsm` expects. The locker has two states, `locked` and `unlocked`. In the `unlocked` state, the initial one,

```
init([]) ->
    {ok,unlocked,[]}.
```

we can asynchronously lock the locker,

```
unlocked(lock,S) ->
    {next_state,locked,S}.
```

or read the value associated with a key, returning it as the reply.

```
unlocked({read,Key},Caller,S) ->
    gen_fsm:reply(Caller,proplists:get_value(Key,S)),
    {next_state,unlocked,S}.
```

In the `locked` state, we can unlock the locker, or write a key-value pair, via an asynchronous call,

```
locked({write,Key,Value},S) ->
    {next_state,locked,[{Key,Value}|proplists:delete(Key,S)]};
locked(unlock,S) ->
    {next_state,unlocked,S}.
```

or we can read a value, just as in the `unlocked` state.

```
locked({read,Key},Client,S) ->
    gen_fsm:reply(Client,proplists:get_value(Key,S)),
    {next_state,locked,S}.
```

Clients interact with the locker via the following functions, which just send synchronous or asynchronous events to the `gen_fsm`, whichever is appropriate:

```
lock() ->
    gen_fsm:send_event(locker,lock).

unlock() ->
    gen_fsm:send_event(locker,unlock).

write(Key,Val) ->
    gen_fsm:send_event(locker,{write,Key,Val}).

read(Key) ->
    gen_fsm:sync_send_event(locker,{read,Key}).
```

Finally, we need to be able to start the locker

```
start() ->
    gen_fsm:start({local,locker},?MODULE,[],[]).
```

and to stop it again

```
stop() ->
    gen_fsm:sync_send_all_state_event(locker,stop).
```

This is achieved by sending a `stop` event to the locker, which can be handled in any state.

```
handle_sync_event(stop,_,_,_) ->
    {stop,normal,ok,[]}.
```

No special clean-up is required on termination.

```
terminate(_,_,_) ->
    ok.
```

3 The Locker Specification

Now we move on to the locker specification. The code in this section provides the call-backs that `eqc_fsm` expects, and uses `eqc_fsm` to define a test property. In this simple case, the specification duplicates much of the implementation, but in general this is of course not the case.

3.1 Locker States

The first part of the `eqc_fsm` specification defines the states and transitions of the locker. Once again, there are two states; we just list the transitions from each state, and give generators for the calls that make each transition.

```
unlocked(S) ->
    read_transition(S) ++
    [{locked,{call,?MODULE,lock,[]}}].

locked(S) ->
    read_transition(S) ++
    [{unlocked,{call,?MODULE,unlock,[]}},
     {locked,{call,?MODULE,write,[key(),value()]}}].
```

The parameter `S` is the state data, in this case a proplist that models the one in the implementation. Notice that we *abstract out* the read transition, since it can be made from either state.

```
read_transition(S) ->
    [{history,{call,locker,read,[elements(proplists:get_keys(S))]}}].
```

We specify the target state as `history` here, which means the current state, whatever it is. We also need to specify how to generate keys and values in the test cases:

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

value() ->
    int().
```

Just as for `gen_fsm`, we have to specify which state is the initial one, and the value of the initial state data:

```
initial_state() ->
    unlocked.

initial_state_data() ->
    [].
```

3.2 State Transitions

While the definitions above specify how we moved between the `locked` and `unlocked` states, we still have to specify how each transition affects the *state data*. Since both our specification and our implementation use a proplist to hold the data, then in this case they do precisely the same thing:

```
next_state_data(_,_,S,_,{call,_,write,[Key,Value]}) ->
    [{Key,Value}|proplists:delete(Key,S)];
next_state_data(_,_,S,_,{call,_,_,_}) ->
    S.
```

We can see that only `write` affects the state.

3.3 Transition Postconditions

The only API call to return an interesting result is `read`, so we simply check that the value it returns is correct.

```
postcondition(_,_,S,{call,_,read,[Key]},R) ->
    R == proplists:get_value(Key,S);
postcondition(_,_,_,_,R) ->
    R == ok.
```

3.4 Transition Preconditions

Although there there's no real need, we'll restrict `read` operations to keys which have previously been written—just to illustrate what preconditions look like.

```
precondition(_,_,S,{call,_,read,[Key]}) ->
    proplists:is_defined(Key,S);
precondition(_,_,_,_) ->
    true.
```

3.5 The Property to Test

It just remains to define the property to be tested, which starts the locker, runs a random sequence of commands, stops it, and then checks that the test was ok—that is, no exceptions were raised, and all the postconditions were true.

```
prop_locker() ->
    ?FORALL(Cmds,commands(?MODULE),
            begin
                start(),
                {H,_S,Res} = run_commands(?MODULE,Cmds),
                stop(),
```

```
            aggregate(zip(state_names(H),command_names(Cmds)),
                      Res == ok)
    end).
```

Note that the property looks just like an `eqc_statem` property—but `commands` and `run\_commands` are imported from `eqc_fsm`, not `eqc_statem`.

We've instrumented the property to collect statistics on the states and transitions that were actually tested. Running the tests, which can be done by

```
eqc:quickcheck(lock\_eqc:prop\_locker())
```

reveals that `read` is tested less often than the other API operations—try it now. Of course, this is because no `read` can be performed until at least one key has been written. In the next section, we'll see how to fix this.

4 Adjusting Test Case Distribution

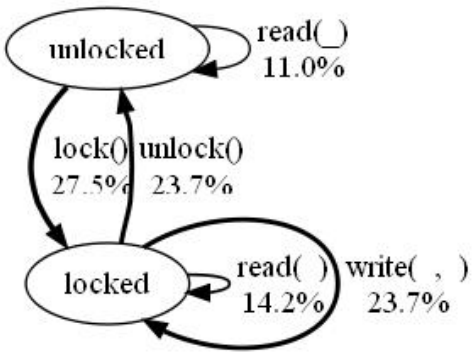
We just saw how to *measure* the actual distribution of transitions followed, but it can also be useful to *analyse* it. Calling

```
eqc_fsm:visualize(lock\_eqc).
```

analyses the state space of the `lock\_eqc` specification, and generates a diagram in which transitions are annotated with their execution frequency, as a percentage of the total number of transitions executed in tests. However, since the analyser doesn't actually generate test cases and test preconditions, it can't take into account that the precondition of `read` is often false—unless we tell it. We do so by providing an *estimate* of the probability that a precondition will succeed, via the following (optional) call-back:

```
precondition_probability(locked,_,{call,_,read,_}) ->
    0.6;
precondition_probability(unlocked,_,{call,_,read,_}) ->
    0.4;
precondition_probability(_,_,_) ->
    1.
```

That is, we estimate that a `read` in a `locked` state will be possible 60% of the time, and so on. Now when we run the visualizer, then these probabilities are taken into account, and we see the following diagram:



The diagram illustrates that read transitions are tested rather rarely.

The way to address this is by *weighting* read transitions more heavily. This can be done either by defining a `weight` callback manually, or by using QuickCheck's automated weight assignment. To do the latter, call

```
eqc_fsm:automate_weights(lock\_eqc).
```

It will output a suitable definition of the `weight` callback, which you can paste into this file to use it, and display the result of a new analysis with the new weights... which tests `read` much more often.

It's also possible to assign a higher priority to some of the transitions, so that the weight assignment tries to test them more often. For example, uncomment the following code

```
priority(unlocked,_,{call,_,lock,_}) ->
10;
priority(_,_,_) ->
1.
```

which assigns `lock` transitions a much higher priority, and run `automate\_weights` again. You will find the weights assigned result in a very different distribution of transitions!

We hope that working through this example has helped you familiarize yourself with `eqc_fsm`. Good luck with writing specifications of your own!