QuickCheck - McErlang Integration Tutorial

by Hans Svensson

Prerequisites

Before trying to use the combination of QuickCheck and McErlang, we should make sure that we have all the necessary components installed.

Testing QuickCheck

We can use `eqc:start/0` to check whether QuickCheck is working. It will return either:

```
1> eqc:start().
Starting eqc version 1.194 (compiled at {{2010,5,18},{20,16,23}})
Licence reserved until {{2010,9,29},{12,0,34}}
<0.6756.4>
```

or:

```
2> eqc:start().
The QuickCheck server eqc is already running--no need to restart it.
ok
```

if it works as expected.

Testing McErlang

McErlang also has an application that we should be able to start, we could start it with either `mce_app:start/0` or the more formal `application:start(mcerlang)`.

```
3> mce_app:start().
ok
4> application:start(mcerlang).
{error,{already_started,mcerlang}}
```

Locating the integration code

For ultimate success, we also need some QuickCheck-modules to be prepared for running inside McErlang. If we can load the module `eqc_mcerlang` everything is probably in place.

```
5> code:ensure_loaded(eqc_mcerlang).
{module,eqc_mcerlang}
```

A first example

Let us use a (broken) implementation of parallel map as an illustrative example:

```
bad_pmap(Fun, List) ->
 Self = self(),
 [ spawn(fun() -> Self ! Fun(E) end) || E <- List],
 [ receive X -> X end || _ <- List].
```

A simple property for this function would be:

```
prop_bad_pmap() ->
  ?FORALL(
  {F,Input}, {function1(nat()),list(nat())},
  begin
   Res  = lists:map(F,Input),
   PRes = bad_pmap(F,Input),
   PRes == Res
  end).
```

where `function1/1` generates a random function (with the return type as indicated by its argument nat() in this case). So in effect we generate a random function, and apply it to a random list both using `lists:map` as a reference, and `bad_pmap`. The resulting lists should be equal. Let's try it:

```
6> c(eqc_mcerlang_tutorial).
./eqc_mcerlang_tutorial.erl:151: Warning: variable 'Res' is unused
{ok,eqc_mcerlang_tutorial}
7> eqc:quickcheck(eqc_mcerlang_tutorial:prop_bad_pmap()).
```

```
.....................................................................................................
OK, passed 100 tests
true
```

Wow, that worked nicely. However, as suggested by the function name, this particular implementation of `pmap` is broken, but our QuickCheck property does not seem able to find the error. The problem lies in the Erlang scheduler, which is (for such small programs) quite deterministic and thus the error is rarely exposed. One solution could be to use PULSE, and its random scheduling to run the tests, another to immediately turn to McErlang. Let us show both variants.

bad_pmap with PULSE

Using PULSE is fairly simple, we need to use the `?PULSE`-macro in our property, but the rest should be automatic.

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

prop_bad_pmap_pulse() ->
  ?FORALL(
  {F,Input}, {function1(nat()),list(nat())},
  ?PULSE(
  PRes,
  bad_pmap(F,Input),
  begin
   Res  = lists:map(F,Input),
   ?WHENFAIL(io:format("~p /= ~p\n",[Res,PRes]),
       PRes == Res)
  end)
 ).
```

`?PULSE(Result,Expression,Property)` takes three arguments, the first is the `Result` of running the `Expression` in PULSE and the last argument is the QuickCheck property. Ok, let us try it, we first need to compile the code using the PULSE parse-transform:

```
8> c(eqc_mcerlang_tutorial,[{parse_transform,pulse_instrument}]).
{ok,eqc_mcerlang_tutorial}
```

Then run it:

```
9> eqc:quickcheck(eqc_mcerlang_tutorial:prop_bad_pmap_pulse()).
...Failed! Reason:
{'EXIT',{application,pulse,not_running}}
After 4 tests.
{#Fun<eqc_gen.102.126639374>,[0]}
{20732,54965,27162}
false
```

What!? Aha, we forgot to start the PULSE-application, we start it and try again...

```
10> pulse:start().
ok
11> eqc:quickcheck(eqc_mcerlang_tutorial:prop_bad_pmap_pulse()).
..............Failed! After 16 tests.
{#Fun<eqc_gen.102.126639374>,[5,2]}
{5648,45400,40546}
[5,3] /= [3,5]
Shrinking.(1 times)
{#Fun<eqc_gen.102.126639374>,[5,0]}
{5648,45400,40546}
[5,3] /= [3,5]
false
```

The property fails, just as expected!

bad_pmap with McErlang

Using QuickCheck with McErlang is not much more complicated than using PULSE. There is a corresponding `?MCERLANG`-macro, which takes four arguments, and we need to invoke the McErlang compiler, but first the property:

```
-include_lib("eqc_mcerlang/include/mcerlang.hrl").

prop_bad_pmap_mcerlang() ->
  ?FORALL(
  {F,Input}, {function1(nat()),list(nat())},
  begin
   Res  = lists:map(F,Input),
```

```
    ?MCERLANG(
    [?MODULE],
    PRes,
    bad_pmap(F,Input),
    ?WHENFAIL(io:format("~p /= ~p\n",[Res,PRes]),
        PRes == Res))
  end).
```

`?MCERLANG(InsMods,Result,Expression,Property)` takes four arguments, the second is the `Result` of running the `Expression` in McErlang and the last argument is the QuickCheck property. The first argument, `InsMods` is a list of modules that should be compiled using McErlang. Unfortunately, we still need to make sure that they are in fact compiled with McErlang, but this is at least also checked before running the property. Let us illustrate that:

```
12> eqc:quickcheck(eqc_mcerlang_tutorial:prop_bad_pmap_mcerlang()).
Failed! After 1 tests.
{#Fun<eqc_gen.102.126639374>,[]}
ERROR: Running McErlang-property without using McErlang-compiler!
false
```

I.e. we need to compile the module using the McErlang compiler:

```
13> mce_erl_compile:file("eqc_mcerlang_tutorial.erl",[{outdir,"."}]).
Applying parse transform eqc_mce_prop_transform
{ok,[eqc_mcerlang_tutorial]}
```

Don't forget to load the result.

```
14> l(eqc_mcerlang_tutorial).
{module,eqc_mcerlang_tutorial}
```

Now we can try the property:

```
15> eqc:quickcheck(eqc_mcerlang_tutorial:prop_bad_pmap_mcerlang()).
Failed! After 1 tests.
{#Fun<eqc_gen.102.126639374>,[]}
McErlang-application is not running
false
```

What again!? Aha, there is also a McErlang-application, we start it and try again...

```
16> mce_app:start().
ok
17> eqc:quickcheck(eqc_mcerlang_tutorial:prop_bad_pmap_mcerlang()).
............Failed! After 14 tests.
{#Fun<eqc_gen.102.126639374>,[2,4]}
McErlang generated a Counter-example, it can be accessed via mce_app:get_res().
[0,1] /= [1,0]
Shrinking...(3 times)
{#Fun<eqc_gen.102.126639374>,[0,1]}
McErlang generated a Counter-example, it can be accessed via mce_app:get_res().
[2,3] /= [3,2]
false
```

The property fails, just as expected!

Process registry example

As a second more involved example, we use the infamous `proc_reg`-example. The code originates from an aborted attempt of implementing an optimized process registry. The implementation was never completed, and the final prototype contains some intricate concurrency errors that are hard to expose using QuickCheck alone. The optimized process registry is described in more detail in Finding Race Conditions in Erlang with QuickCheck and PULSE.

The API supported by `proc_reg`is: `reg(Name,Pid)` to register a pid, where(Name) to look up a pid, unreg(Name) to remove a registration, and send(Name,Msg) to send a message to a registered process. `proc_reg` stores the association between names and pids in Erlang Term Storage ("ETS tables")-hash tables. It also creates a monitor for each registered process, whose effect is to send proc_reg a "DOWN" message if the registered process crashes, so it can be removed from the registry. Two ETS table entries are created for each registration: a 'forward' entry that maps names to pids, and a 'reverse' entry that maps registered pids to the monitor reference.

The module `proc_reg_eqc` contains an `eqc_statem` state machine specification for the process registry. We wont explain it in detail here, instead we refer to QuickCheck training material and the paper mentioned above.

To test the concurrency aspects of the process registry, we use the state machine specification and the `parallel_commands`-functionality from QuickCheck. This functionality generate parallel test cases by parallelizing a suffix of an `eqc_statem` test case, separating it into two lists of commands of roughly equal length. We run tests by first running the prefix, then spawning two processes to run the two command-lists in parallel, and collecting their results, which will be non-deterministic depending on the actual parallel scheduling of events. We decide whether a test has passed, by attempting to construct a sequentialization of the test case which explains the results observed.

For the `proc_reg`-example a property using parallel test cases could look like:

proc_reg_eqc.erl

```
prop_parallel() ->
    ?FORALL(Repetitions,?SHRINK(1,[10]),
      ?FORALL(PCmds,parallel_commands(?MODULE),
     ?ALWAYS(Repetitions,
    begin
     {ok,Tabs} = proc_reg_tabs:start_link(),
     {ok,Server} = proc_reg:start_link(),
     {H,AB,Res} = run_parallel_commands(?MODULE,PCmds),
     kill_all_pids({H,AB}),
     cleanup(Tabs,Server),
     ?WHENFAIL(
     io:format("Sequential: ~p\nParallel: ~p\nRes: ~p\n",
         [H,AB,Res]),
     Res == ok)
    end))).
```

Where we invoke `parallel_commands(Module)` to generate a parallel test case, and `run_parallel_commands(Module,ParallelTest)` to actually run the test. Note that the test case includes code to start the process registry, as well as cleaning up afterwards. This is necessary to ensure that each test starts in a known state.

Running this property sometimes reveals the concurrency error, but it could take quite a few tests before the error is manifested. To compile the tutorial example, the module `erl_make` is provided. (Don't forget to compile itself first though!)

```
18> c(erl_make).
{ok,erl_make}
19> erl_make:make().
Making files for normal run
Adding ./ebin to code-path...true
20> eqc:quickcheck(proc_reg_eqc:prop_parallel()).
.....................................................................................
OK, passed 100 tests
true
21> eqc:quickcheck(eqc:numtests(1000,proc_reg_eqc:prop_parallel())).
.....................................................................................
.Failed! After 102 tests.
1
{[{set,{var,1},{call,proc_reg_eqc,unreg,[c]}},
  {set,{var,2},{call,proc_reg_eqc,spawn,[]}},
  {set,{var,3},{call,proc_reg,where,[c]}},
  {set,{var,4},{call,proc_reg_eqc,reg,[b,{var,2}]}},
  {set,{var,5},{call,proc_reg_eqc,reg,[a,{var,2}]}},
  {set,{var,6},{call,proc_reg_eqc,unreg,[d]}},
  {set,{var,7},{call,proc_reg,where,[c]}},
  {set,{var,8},{call,proc_reg,where,[c]}},
  {set,{var,9},{call,proc_reg_eqc,spawn,[]}},
  {set,{var,10},{call,proc_reg,where,[a]}},
  {set,{var,11},{call,proc_reg_eqc,spawn,[]}},
  {set,{var,12},{call,proc_reg_eqc,spawn,[]}},
  {set,{var,13},{call,proc_reg_eqc,reg,[c,{var,9}]}},
  {set,{var,14},{call,proc_reg_eqc,spawn,[]}},
  {set,{var,15},{call,proc_reg_eqc,reg,[d,{var,2}]}},
  {set,{var,16},{call,proc_reg_eqc,unreg,[c]}},
  {set,{var,17},{call,proc_reg_eqc,spawn,[]}},
  {set,{var,18},{call,proc_reg_eqc,kill,[{var,2}]}},
  {set,{var,19},{call,proc_reg_eqc,reg,[c,{var,12}]}},
  {set,{var,20},{call,proc_reg_eqc,kill,[{var,12}]}},
  {set,{var,21},{call,proc_reg_eqc,unreg,[c]}},
  {set,{var,22},{call,proc_reg,where,[a]}},
  {set,{var,23},{call,proc_reg_eqc,reg,[a,{var,2}]}},
  {set,{var,24},{call,proc_reg_eqc,kill,[{var,17}]}},
  {set,{var,25},{call,proc_reg_eqc,unreg,[c]}},
  {set,{var,26},{call,proc_reg_eqc,kill,[{var,14}]}},
  {set,{var,27},{call,proc_reg_eqc,reg,[c,{var,14}]}},
  {set,{var,28},{call,proc_reg_eqc,reg,[d,{var,9}]}},
  {set,{var,29},{call,proc_reg_eqc,kill,[{var,9}]}},
  {set,{var,30},{call,proc_reg_eqc,unreg,[d]}},
  {set,{var,31},{call,proc_reg_eqc,reg,[c,{var,2}]}},
  {set,{var,32},{call,proc_reg_eqc,unreg,[c]}},
```

```
   {set,{var,33},{call,proc_reg_eqc,kill,[{var,11}]}},
   {set,{var,34},{call,proc_reg_eqc,reg,[d,{var,11}]}},
   {set,{var,35},{call,proc_reg_eqc,reg,[b,{var,9}]}},
   {set,{var,36},{call,proc_reg_eqc,unreg,[a]}}],
  [[{set,{var,37},{call,proc_reg,where,[d]}}],
   [{set,{var,38},{call,proc_reg_eqc,unreg,[d]}}],
   [{set,{var,39},{call,proc_reg_eqc,reg,[b,{var,2}]}},
    {set,{var,40},{call,proc_reg_eqc,reg,[b,{var,2}]}}],
   [{set,{var,41},{call,proc_reg,where,[c]}},
    {set,{var,42},{call,proc_reg_eqc,reg,[c,{var,12}]}},
    {set,{var,43},{call,proc_reg_eqc,reg,[b,{var,9}]}}]]]}
Sequential: [{{set,{var,1},{call,proc_reg_eqc,unreg,[c]}},
              {'EXIT',{badarg,[{proc_reg,unreg,1},
                               {proc_reg_eqc,unreg,1},
                               {eqc_statem,f511_0,5},
                               {eqc_statem,f724_0,5},
                               {eqc_statem,run_parallel_commands,2},
                               {proc_reg_eqc,'-prop_parallel/0-fun-3-',1},
                               {eqc,f692_0,4},
                               {eqc_gen,gen,3}]}}},
             {{set,{var,2},{call,proc_reg_eqc,spawn,[]}},<0.8903.4>},
             {{set,{var,3},{call,proc_reg,where,[c]}},undefined},
             {{set,{var,4},{call,proc_reg_eqc,reg,[b,<0.8903.4>]}},true},
             {{set,{var,5},{call,proc_reg_eqc,reg,[a,<0.8903.4>]}},true},
             {{set,{var,6},{call,proc_reg_eqc,unreg,[d]}},
              {'EXIT',{badarg,[{proc_reg,unreg,1},
                               {proc_reg_eqc,unreg,1},
                               {eqc_statem,f511_0,5},
                               {eqc_statem,f724_0,5},
                               {eqc_statem,run_parallel_commands,2},
                               {proc_reg_eqc,'-prop_parallel/0-fun-3-',1},
                               {eqc,f692_0,4},
                               {eqc_gen,gen,3}]}}},
             {{set,{var,7},{call,proc_reg,where,[c]}},undefined},
             {{set,{var,8},{call,proc_reg,where,[c]}},undefined},
             {{set,{var,9},{call,proc_reg_eqc,spawn,[]}},<0.8904.4>},
             {{set,{var,10},{call,proc_reg,where,[a]}},<0.8903.4>},
             {{set,{var,11},{call,proc_reg_eqc,spawn,[]}},<0.8905.4>},
             {{set,{var,12},{call,proc_reg_eqc,spawn,[]}},<0.8906.4>},
             {{set,{var,13},{call,proc_reg_eqc,reg,[c,<0.8904.4>]}},true},
             {{set,{var,14},{call,proc_reg_eqc,spawn,[]}},<0.8907.4>},
             {{set,{var,15},{call,proc_reg_eqc,reg,[d,<0.8903.4>]}},true},
             {{set,{var,16},{call,proc_reg_eqc,unreg,[c]}},true},
             {{set,{var,17},{call,proc_reg_eqc,spawn,[]}},<0.8908.4>},
             {{set,{var,18},{call,proc_reg_eqc,kill,[<0.8903.4>]}},ok},
             {{set,{var,19},{call,proc_reg_eqc,reg,[c,<0.8906.4>]}},true},
             {{set,{var,20},{call,proc_reg_eqc,kill,[<0.8906.4>]}},ok},
             {{set,{var,21},{call,proc_reg_eqc,unreg,[c]}},
              {'EXIT',{badarg,[{proc_reg,unreg,1},
                               {proc_reg_eqc,unreg,1},
                               {eqc_statem,f511_0,5},
                               {eqc_statem,f724_0,5},
                               {eqc_statem,run_parallel_commands,2},
                               {proc_reg_eqc,'-prop_parallel/0-fun-3-',1},
                               {eqc,f692_0,4},
                               {eqc_gen,gen,3}]}}},
             {{set,{var,22},{call,proc_reg,where,[a]}},undefined},
             {{set,{var,23},{call,proc_reg_eqc,reg,[a,<0.8903.4>]}},true},
             {{set,{var,24},{call,proc_reg_eqc,kill,[<0.8908.4>]}},ok},
             {{set,{var,25},{call,proc_reg_eqc,unreg,[c]}},
              {'EXIT',{badarg,[{proc_reg,unreg,1},
                               {proc_reg_eqc,unreg,1},
                               {eqc_statem,f511_0,5},
                               {eqc_statem,f724_0,5},
                               {eqc_statem,run_parallel_commands,2},
                               {proc_reg_eqc,'-prop_parallel/0-fun-3-',1},
                               {eqc,f692_0,4},
                               {eqc_gen,gen,3}]}}},
             {{set,{var,26},{call,proc_reg_eqc,kill,[<0.8907.4>]}},ok},
             {{set,{var,27},{call,proc_reg_eqc,reg,[c,<0.8907.4>]}},true},
             {{set,{var,28},{call,proc_reg_eqc,reg,[d,<0.8904.4>]}},true},
             {{set,{var,29},{call,proc_reg_eqc,kill,[<0.8904.4>]}},ok},
             {{set,{var,30},{call,proc_reg_eqc,unreg,[d]}},
              {'EXIT',{badarg,[{proc_reg,unreg,1},
                               {proc_reg_eqc,unreg,1},
                               {eqc_statem,f511_0,5},
                               {eqc_statem,f724_0,5},
                               {eqc_statem,run_parallel_commands,2},
                               {proc_reg_eqc,'-prop_parallel/0-fun-3-',1},
                               {eqc,f692_0,4},
                               {eqc_gen,gen,3}]}}},
             {{set,{var,31},{call,proc_reg_eqc,reg,[c,<0.8903.4>]}},true},
             {{set,{var,32},{call,proc_reg_eqc,unreg,[c]}},
              {'EXIT',{badarg,[{proc_reg,unreg,1},
                               {proc_reg_eqc,unreg,1},
```

```
                              {eqc_statem,f511_0,5},
                              {eqc_statem,f724_0,5},
                              {eqc_statem,run_parallel_commands,2},
                              {proc_reg_eqc,'-prop_parallel/0-fun-3-',1},
                              {eqc,f692_0,4},
                              {eqc_gen,gen,3}]}}},
             {{set,{var,33},{call,proc_reg_eqc,kill,[<0.8905.4>]}},ok},
             {{set,{var,34},{call,proc_reg_eqc,reg,[d,<0.8905.4>]}},true},
             {{set,{var,35},{call,proc_reg_eqc,reg,[b,<0.8904.4>]}},true},
             {{set,{var,36},{call,proc_reg_eqc,unreg,[a]}},
              {'EXIT',{badarg,[{proc_reg,unreg,1},
                              {proc_reg_eqc,unreg,1},
                              {eqc_statem,f511_0,5},
                              {eqc_statem,f724_0,5},
                              {eqc_statem,run_parallel_commands,2},
                              {proc_reg_eqc,'-prop_parallel/0-fun-3-',1},
                              {eqc,f692_0,4},
                              {eqc_gen,gen,3}]}}}]
Parallel: [[{set,{var,37},{call,proc_reg,where,[d]},undefined}],
           [{set,
                {var,38},
                {call,proc_reg_eqc,unreg,[d]},
                {'EXIT',
                    {badarg,
                        [{proc_reg,unreg,1},
                         {proc_reg_eqc,unreg,1},
                         {eqc_run_parallel_commands,run_child,3}]}}}],
           [{set,{var,39},{call,proc_reg_eqc,reg,[b,<0.8903.4>]},true},
            {set,{var,40},{call,proc_reg_eqc,reg,[b,<0.8903.4>]},true}],
           [{set,{var,41},{call,proc_reg,where,[c]},undefined},
            {set,{var,42},{call,proc_reg_eqc,reg,[c,<0.8906.4>]},true},
            {set,
                {var,43},
                {call,proc_reg_eqc,reg,[b,<0.8904.4>]},
                {'EXIT',
                    {badarg,
                        [{proc_reg,reg,2},
                         {proc_reg_eqc,reg,2},
                         {eqc_run_parallel_commands,run_child,3}]}}}]]
Res: no_possible_interleaving
Shrinking.............(14 times)
10
{[{set,{var,2},{call,proc_reg_eqc,spawn,[]}},
  {set,{var,9},{call,proc_reg_eqc,spawn,[]}},
  {set,{var,18},{call,proc_reg_eqc,kill,[{var,2}]}}],
 [[{set,{var,39},{call,proc_reg_eqc,reg,[b,{var,2}]}},
   {set,{var,40},{call,proc_reg_eqc,reg,[b,{var,2}]}}],
  [{set,{var,43},{call,proc_reg_eqc,reg,[b,{var,9}]}}]]}
Sequential: [{{set,{var,2},{call,proc_reg_eqc,spawn,[]}},<0.15983.4>},
             {{set,{var,9},{call,proc_reg_eqc,spawn,[]}},<0.15984.4>},
             {{set,{var,18},{call,proc_reg_eqc,kill,[<0.15983.4>]}},ok}]
Parallel: [[{set,{var,39},{call,proc_reg_eqc,reg,[b,<0.15983.4>]},true},
            {set,{var,40},{call,proc_reg_eqc,reg,[b,<0.15983.4>]},true}],
           [{set,
                {var,43},
                {call,proc_reg_eqc,reg,[b,<0.15984.4>]},
                {'EXIT',
                    {badarg,
                        [{proc_reg,reg,2},
                         {proc_reg_eqc,reg,2},
                         {eqc_run_parallel_commands,run_child,3}]}}}]]
Res: no_possible_interleaving
false
```

Using McErlang

As we saw in the `pmap`-example, using McErlang to run (part of) a QuickCheck property required only a slight modification to the property and a re-compilation. However, for more intricate QuickCheck properties there also need to be McErlang-compiled versions of some QuickCheck library functions. Luckily, `parallel_commands` and `run_parallel_commands` are among the available functions (the same goes for `commands` and `run_commands` for normal non-parallel state machine testing). See the `eqc_mcerlang`-documentation for a list of supported functionality.

A version of the parallel property for `proc_reg` using McErlang could look like:

proc_reg_eqc.erl

```
-define(INSMODS,[proc_reg, proc_reg_tabs, proc_reg_eqc]).

prop_parallel_mcerlang() ->
    ?FORALL(PCmds,parallel_commands(?MODULE),
    ?MCERLANG(
    ?INSMODS,
    {H,AB,Res},
```

```
    begin
     proc_reg_tabs:start_link(),
     proc_reg:start_link(),
     run_parallel_commands(?MODULE,PCmds)
    end,
    ?WHENFAIL(
    io:format("Sequential: ~p\nParallel: ~p\nRes: ~p\n",
        [H,AB,Res]),
    Res == ok)
   )).
```

We should note that the code part of this property is in fact simpler than the non-McErlang version of the property. The reason is that we do not need to clean up afterwards, each McErlang run starts in a fresh environment. To compile it using the provided `erl_make`-module, we use `erl_make:make_mcerlang()`.

```
22> erl_make:make_mcerlang().
Building example for McErlang
Applying parse transform eqc_mce_prop_transform
Adding ./ebin to code-path...true
```

Again, we need to ensure that the McErlang-application is started. The application fills two purposes, the first is to keep a history of run tests, and the second is to keep a state of parameters to use during model checking. For a detailed set of options, please refer to the documentation of `mce_app`. The most often used options are perhaps limiting the size of the state space table used by McErlang, and limiting the time each invocation of McErlang can maximally use. Here we limit the state space to 256 MB, and we don't allow a model checking run to take more than 10 seconds, before running the property:

```
23> mce_app:set_table_bound(256).
ok
24> mce_app:set_time_limit(10).
ok
25> eqc:quickcheck(proc_reg_eqc:prop_parallel_mcerlang()).
.................................Failed! After 38 tests.
{[{set,{var,1},{call,proc_reg_eqc,unreg,[b]}},
  {set,{var,2},{call,proc_reg_eqc,spawn,[]}},
  {set,{var,3},{call,proc_reg_eqc,reg,[c,{var,2}]}},
  {set,{var,4},{call,proc_reg,where,[d]}},
  {set,{var,5},{call,proc_reg_eqc,reg,[c,{var,2}]}},
  {set,{var,6},{call,proc_reg_eqc,reg,[b,{var,2}]}},
  {set,{var,7},{call,proc_reg,where,[a]}}],
 [[{set,{var,8},{call,proc_reg_eqc,reg,[d,{var,2}]}},
   {set,{var,9},{call,proc_reg_eqc,reg,[c,{var,2}]}},
   {set,{var,10},{call,proc_reg_eqc,reg,[a,{var,2}]}}],
  [{set,{var,11},{call,proc_reg,where,[a]}},
   {set,{var,12},{call,proc_reg_eqc,kill,[{var,2}]}},
   {set,{var,13},{call,proc_reg_eqc,spawn,[]}},
   {set,{var,14},{call,proc_reg_eqc,reg,[c,{var,2}]}},
   {set,{var,15},{call,proc_reg,where,[b]}}]]}
McErlang generated a Counter-example, it can be accessed via mce_app:get_res().
Sequential: [{{set,{var,1},{call,proc_reg_eqc,unreg,[b]}},
              {'EXIT',{badarg,[]}}},
             {{set,{var,2},{call,proc_reg_eqc,spawn,[]}},
              {pid,'node0@hanssv-ituniv',4}},
             {{set,{var,3},
                   {call,proc_reg_eqc,reg,[c,{pid,'node0@hanssv-ituniv',4}]}},
              true},
             {{set,{var,4},{call,proc_reg,where,[d]}},undefined},
             {{set,{var,5},
                   {call,proc_reg_eqc,reg,[c,{pid,'node0@hanssv-ituniv',4}]}},
              {'EXIT',{badarg,[]}}},
             {{set,{var,6},
                   {call,proc_reg_eqc,reg,[b,{pid,'node0@hanssv-ituniv',4}]}},
              true},
             {{set,{var,7},{call,proc_reg,where,[a]}},undefined}]
Parallel: [[{set,{var,8},
                 {call,proc_reg_eqc,reg,[d,{pid,'node0@hanssv-ituniv',4}]},
                 true},
            {set,{var,9},
                 {call,proc_reg_eqc,reg,[c,{pid,'node0@hanssv-ituniv',4}]},
                 true},
            {set,{var,10},
                 {call,proc_reg_eqc,reg,[a,{pid,'node0@hanssv-ituniv',4}]},
                 true}],
           [{set,{var,11},{call,proc_reg,where,[a]},undefined},
            {set,{var,12},
                 {call,proc_reg_eqc,kill,[{pid,'node0@hanssv-ituniv',4}]},
                 ok},
            {set,{var,13},
                 {call,proc_reg_eqc,spawn,[]},
                 {pid,'node0@hanssv-ituniv',11}},
```

```
            {set,{var,14},
                 {call,proc_reg_eqc,reg,[c,{pid,'node0@hanssv-ituniv',4}]},
                 {'EXIT',{badarg,[]}}},
            {set,{var,15},{call,proc_reg,where,[b]},undefined}]]
Res: no_possible_interleaving
Shrinking.........(9 times)
{[{set,{var,2},{call,proc_reg_eqc,spawn,[]}},
  {set,{var,3},{call,proc_reg_eqc,reg,[c,{var,2}]}},
  {set,{var,12},{call,proc_reg_eqc,kill,[{var,2}]}}],
 [[{set,{var,9},{call,proc_reg_eqc,reg,[c,{var,2}]}}],
  [{set,{var,14},{call,proc_reg_eqc,reg,[c,{var,2}]}}]]]
McErlang generated a Counter-example, it can be accessed via mce_app:get_res().
Sequential: [{{set,{var,2},{call,proc_reg_eqc,spawn,[]}},
              {pid,'node0@hanssv-ituniv',4}},
             {{set,{var,3},
                   {call,proc_reg_eqc,reg,[c,{pid,'node0@hanssv-ituniv',4}]}},
              true},
             {{set,{var,12},
                   {call,proc_reg_eqc,kill,[{pid,'node0@hanssv-ituniv',4}]}},
              ok}]
Parallel: [[{set,{var,9},
                 {call,proc_reg_eqc,reg,[c,{pid,'node0@hanssv-ituniv',4}]},
                 {'EXIT',{badarg,[]}}}],
           [{set,{var,14},
                 {call,proc_reg_eqc,reg,[c,{pid,'node0@hanssv-ituniv',4}]},
                 true}]]
Res: no_possible_interleaving
false
```

The property fails, and we get (after some time) a small counterexample.