Exercise (Thursday): Testing a Process Registry

In today's exercise we will develop tests for a process registration module that simulates the Erlang process registry—an example of an API with side effects. The code that we will be testing is in the module **Registry.hs**—**DO NOT READ THIS FILE!** Imagine instead that you are testing the *real* Erlang registry, implemented by around 1500 lines of C code in the Erlang virtual machine. Our goal is to discover—and specify—the behaviour of the registry by testing, treating the implementation as a black box.

You are provided with the module **StateModel.hs**, which contains the state modelling library described in the lecture. You are also provided with two *partial* models of the registry, in files **RegistryModel1.hs** and **RegistryModel2.hs**. **DO NOT READ RegistryModel2** until you have completed the positive testing exercises—it contains a solution to these exercises.

Positive Testing of the Registry

For the first part of this exercise session, we will work with module **RegistryModel1**, which contains the specification code presented in the first part of yesterday's lecture. We will be focusing on *positive testing*, that is, on tests which only perform "successful" calls to the code under test, which we interpret as calls that do not raise an exception.

Load **RegistryModel1** into ghci, and make sure you can test the main property with QuickCheck:

```
quickCheck prop_Registry
```

1. Is it really true that the tests now pass—or did I lie to you? If they fail, *strengthen the precondition of* **register** to avoid the failing cases, and check that the failing cases you found are now discarded by QuickCheck by repeating exactly the same test.
2. The code provided only tests **register—**we shall extend it to test unregister in a similar way. Add **Unregister** actions to the **arbitraryAction** generator. Run **prop_Registry**: your tests should fail. Modify *one function at a time*, testing **prop_Registry** after each change, until your tests pass again.
3. Once your tests pass, run a larger number of tests
   ```
   quickCheck . withMaxSuccess 10000 $ prop_Registry
   ```
   and study the statistics reported. The second table, the **Actions** table, is the interesting one—it shows what percentage of all calls to the API under test were made to each function. Why does this table appear as it does?

Adding Negative Testing with Postconditions

For the second part of this exercise, start from **RegisterModel2** (which you may now read). This file contains a solution to the exercises so far, extended to perform negative testing of **register** as discussed in the lecture.

4. Test **prop_Registry**. Do the tests now pass? If not, the model of **register** is still wrong. Correct it, so that the tests pass.
5. Add negative tests for **unregister** following a similar programme.

Crashing Processes

In practice, processes sometimes crash, and this might conceivably affect the operation of the registry. Thus we would like to test the registry with processes which may crash before or after they are registered, or even while they are in the registry. We can simulate crashes by *killing* processes at known points in a test case; to do so we will include calls to the operation **kill** (defined in **RegisterModel2**) in our tests.

6. Add an action to kill threads to your tests. Does this affect test results? If so, then we know the registry *does* treat living and dead threads differently. Extend the model so that tests pass again (thus discovering exactly how the registry behaves).

Debugging

The file **RegisterBuggy.hs** is an *almost* correct implementation of the registry. See if you can pin down the bug *without* looking at the code:

- First of all, manually, by calling the API directly in ghci, and inspecting the results visually. Don't spend too long on this—perform a "reasonable" set of tests, enough to persuade an average developer that the code seems to work. (If you find the bug at this stage, congratulations!)
- Secondly, run your state machine tests on **RegisterBuggy** instead of **Register** (change the import declaration). See if you can diagnose the bug from the results.

Still want more?

The file **FileSpec.hs** contains foreign import declarations for the C file I/O functions **fopen**, **fclose**, **fread**, **fwrite**, and **fseek**—along with wrapper functions making them easier to call, a simple unit test, and the skeleton of a state machine test. How do **fread**, **fwrite** and **fseek** behave, really? Can you extend the skeleton to a full state-machine test, that checks that **fread** returns the predicted sequence of bytes in every case?