Testing Techniques

Assignment 2

Sasja Gillissen, Martin Huijben, Martijn Terpstra

November 20, 2016

1 Automated Test Execution

In our previous assignment we developed the following use cases

Category	Input	Expected Output
Exiting	`exit`	Nothing
Function Assignment	`f(x):= x + 1 ; f(7)`	`8`
Builtin Functions (1)	`sqrt(ln(log(100)))`	`0.83`
Builtin Functions (2)	`floor(0.5) + ceil(0.5) + round(0.5)`	`2`
Variable Assignment	`k = 3 ; k`	`3`
Builtin Variables	`e * pi`	`8.5397342226735656`
Expression (1)	`(((0 + 1) * 2) - 3) / 4`	`-0.2500000000000000`
Expression (2)	`1 / 0`	Error
Equality (1)	`3 = 3`	`0`
Equality (2)	`2 = 1`	`1`
Equality (3)	`2 = 3`	`-1`

1.1 Describe, design, and develop an automated test execution environment for your SUT.

Our SUT is deterministic so for any given input we can expect the same output. As such we can simply test each use case by supplying input for each test case and seeing if the output matches.

For each use case we create 3 files with the post fixes "_input.txt" "_output.txt" and "_errors.txt". The first is the input we give the SUT, the other two being the expected outputs. These are split in two since UNIX shells output to both STDOUT (normal output) and STDERR(errors). Some of our use cases except the SUT to throw error output so this is included.

For the actual test we have written a simple bash script that.

- For each test pipes the input into our SUT and writes the output and errors to a file

- Compares the outputs with the expected output using the “diff” command
- If the outputs match simply echo “[OK]”
- If the outputs don’t match, show a diff of the outputs outlining what was different.

The SUT passes all tests if all test cases return “[OK][OK]”

1.2 Implement the test cases that you developed for manual testing, as scripts, programs, text files, . . . , so that they can be executed with your test execution environment.

The full code, including the use cases can be found at `https://github.com/Witik/CommandlineCalculator`

1.3 Test your SUT with the automated test scripts and analyze the results.

This is the final output of our test script

```
exit:                 [OK][OK]
function_assignment:  [OK][OK]
builtin_function1:    [OK][OK]
builtin_function2:    [OK][OK]
variable_assignment:  [OK][OK]
builtin_variables:    [OK][OK]
expression1:          [OK][OK]
expression2:          [OK][OK]
equality1:            [OK][OK]
equality2:            [OK][OK]
equality3:            [OK][OK]
```

The result is unsurprising, each test is successful. Since we had previously defined all use cases in detail making correct tests turned out to be easy.

1.4 Evaluate your automated test execution environment.

Since we had previously made very concrete test cases and our program can be communicated with from the command line automation was easy. Once our script could test the first use case, the remaining ten use cases could be added to the test suite in a matter minutes.

The testing suite was kept simple, allowing to to be developed in a lunch break. It would most likely have taken longer to look for a tool, learn it and set it up. The full testing environment could be setup in a fraction of the time it took us to get TorXakis working.

Furthermore because of its simplicity it was trivial to add use cases. Which means that if our SUT would later gain more features we could easily add more use cases to test for.

2 Model-Based Testing

2.1 Modeling Investigation Investigate and study (the behaviour of) your SUT; investigate and argue which parts of your SUT, which interfaces, and which functionality might be tested with TorXakis.

2.2 MBT Modeling Make a model for your SUT in the TorXakis modelling language txs.

2.3 MBT Test Environment Develop a test environment or test architecture which will be used for model-based testing of your SUT.

2.4 MBT Testing Use TorXakis to generate tests, and execute them on your SUT.

2.5 Deliverable Give the models, code, adapters, etc. in such a way that we can run it; provide a 'README'.