UNIT TESTING FRAMEWORK FOR MATLAB

M. Kvasnica, Ľ. Čirka

Institute of Information Engineering,
Automation and Mathematics,
Slovak University of Technology,
81237 Bratislava, Slovakia

Abstract

In this work we present a new framework for unit testing of programs in MATLAB. The framework can be used if the Test Driven Development (TDD) approach is employed in the development process. The framework allows the user to write and execute test files, which verify correct behavior of the tested code. In the TDD approach, each test should verify only a small portion of certain tested function, hence the tests are referred to as *unit tests*. Due to that one usually has a large number of unit tests when developing complex programs. Our framework therefore offers a user-friendly way of creating and managing multiple tests.

1 Introduction

Software development is a complex process especially when multiple developers are involved. Guaranteeing quality, correctness and coherence of all implemented modules is therefore of imminent importance. There are various coding techniques which could be employed in the development process. For instance, in the *extreme programming* approach several developers share the same computer and they split the job such that one coder writes the code and the other one watches over him. Then, after some period of time they exchange the responsibilities. It is without doubt that while this approach can minimize errors in the written code, it is done so on expenses in the man power. Moreover, it is difficult to verify that the currently worked on code is coherent with other modules of the whole software package. Therefore the extreme programming approach further evolved to include automated testing procedures. This approach, also known as the *test driven development* procedure, is based on a, possible large, number test files which represent design requirements put on the whole software. Successful execution of all tests then verifies the correct behavior of the overall code.

This testing is usually performed on 4 levels:

1. *unit testing*, where a minimal unit of the code is being tested;
2. *integration testing*, where possible defects in interactions between several units are tested;
3. *system testing*, where the whole code is integrated into the final product and the testing validates that all design criteria have been met;
4. *acceptance testing*, where all design goals are being validated.

Since the number of tests is usually very large (often in range of thousands of tests or even more), it wouldn't be possible to apply the TDD approach without a framework which is capable of managing and executing the individual test files. Therefore in this paper we present a new unit testing framework for the MATLAB environment. The package allows MATLAB users to easily write unit test files, execute them, compare their results and visualize all results by means of a web application.

This paper is structure as follows. In Section 2 we first review the basics characteristics of test driven development and unit testing. The toolbox itself will be presented in Section 3, before wrapping up the paper by conclusions.

2 Test Driven Development and Unit Testing

In the Test Driven Development (TDD) approach the code is created in a way such that firstly test code which validates certain design goals. The code itself is only written afterwards. The TDD approach therefore gives a rapid feedback whether the code fulfills specified design criteria. The main idea of TDD is the definition of the structure of a simple test language, which is transformed to a source code and subsequently compiled. The test itself is therefore the most important part of this approach.

The cycle of the test driven development framework can be described as follows:

- Creation of a test. Each design requirement has to be represented by one or more test files. The developer creating a respective test must be provided with clear design requirements.
- Running of the created test with subsequent check of the test results. This checks the correctness of the tested code if the test ran without errors.
- Modification of the code. If the test failed during its execution, the corresponding tested code has to be changed to make the test pass. During this phase only minimal changes to the code should be made.
- Code refactoring. The code can be further improved either in terms of functionality or performance-wise. Existing tests guarantee that no regression bugs are introduced during the process of refactoring.

If one wants to apply the TDD approach to software design, two basic rules should always be respected:

1. Newer add new functionality until existing tests all pass;
2. Individual tests should cover as many lines of the tested code as possible, i.e. the code coverage should be maximized.

Unit testing concerns with detailed testing of a relatively small piece of code. This fine granularity is very useful in creation of complex programs where individual functions are difficult to validate in real operation. Under unit testing we understand tools and methodology which serve to validate the correct functionality of the tested code. The testing process consists of three phases:

1. Creation of the test plan
    - state main ideas and plans
    - select the properties which should be tested
    - filter plans and ideas
2. Creation of the set of tests
    - propose a set of tests
    - filter out the tests
3. Examination of the test results in terms of
    - correctness (test passed, test failed)
    - performance, i.e. runtime of each test

In the next section we describe a MATLAB toolbox which implements the TDD approach in spirit of the above mentioned description.

3 Toolbox for unit testing

As already pointed out in the introduction, the toolbox allows users to write unit tests, execute them in an automated fashion, collect test results, compare them against expected results and visualize the total test run. In order to write suitable unit tests, the toolbox provides several functions which trigger an error message when the result of a tested function doesn't meet requirements. To illustrate this behavior, let's consider the following sample unit test, which validates the correctness of the implementation of a `sin` function:

```
1. function test_sin1
2.
3. a = sin(pi)
4. asserttolequal(a, 0)
5.
6. a = sin(pi/2)
7. asserttrue(a == 1)
```

Every such unit test starts with a `function` preamble, followed by the test name (in this case `test_sin1`). Then, on line 3 the tested function is executed for the selected value of the input argument `pi`. Subsequently, the actual outcome of the function is compared against the expected value by means of the `asserttolequal` function. This function, as its name already hints, expects that the first argument `a` is tolerance-equal to the second argument. In other words, the function will stop the execution if `a` differs from zero by more than a tolerance. By default, the tolerance is set to $1 \cdot 10^{-8}$. Another possibility is represented by the `asserttrue` function. This function will pass through if its argument is true. In other words, if the condition `a==1` is not fulfilled, the `asserttrue` function will stop the execution of the test with an error message.

Usually, a complex software package is tested by means of a high number of tests. These individual test files are best to be divided into subdirectories, following the directory structure of the developed software package. Once such an organization of unit tests is available, the toolbox allows to execute all tests in a respective directory by using the following command:

```
D = testdrive('/path/to/directory/')
```

Once such a command is executed, the toolbox first searches for all tests contained in the specified path. All subdirectories located in the root folder are searched recursively. Once all test files are located, the toolbox executes them automatically. The test can either pass, or fail. A test can fail due to several reasons. Either its run was aborted by one of the assert functions, which indicates a misbehavior of the tested function; or the tested function alone failed due to an error in its implementation. The latter case is even more serious, as it can indicate, for instance, a typo in the code, or an unhandled exception.

If a given test passes, the toolbox provides another measure to verify the correctness. Specifically, it can compare the outputs the function has printed to the screen versus expected outcomes stored in a text file. The expected outcomes have to be stored in a file with the `.out` extension, whereas the file has to have the same name as the test itself. For illustration, consider the following unit test:

```
1. function test_roots1
2.
3. roots([1 2 3])
```

Once such a test is executed and the function `roots` is implemented correctly, it should print the following lines of the screen:

```
ans =

  -1.0000 + 1.4142i
  -1.0000 - 1.4142i
```

The user can then create the file `test_roots1.out` and put an expected output there, for instance:

```
answer =

  -1.0000 + 1.4142i
  -1.0000 - 1.4142i
```

Notice that in this case the two outcomes differ only in the name of the output variable (`ans` vs. `answer`). In this case the toolbox will denote the test as `wrong`, because it has passed without any MATLAB errors, but its outcome doesn't correlate to the expected outcome.

When the `D = testdrive(...)` command is executed, it runs the tests found in respective directories. For instance:

```
Looking into "testy/"... found 6 test(s)
Running testy/test1
Running testy/test2
Running testy/test3
Running testy/test4
Error using ==> test4 at 4
user error
Running testy/test5
Error using ==> test5 at 5
output is wrong
Running testy/test6
```

Here we see how the toolbox searched the `testy` directory and found 6 tests in total there. Subsequently all these six tests have been executed. As can be seen, all tests passed successfully, except of tests no. 4 and 5, which have been aborted by an error. The overall test run is then characterized by means of the build object `D`:

```
RESULTS:
 OK    testy/test1
 OK    testy/test2
 SKP   testy/test3
 ERR   testy/test4
 EXF   testy/test5
 WRO   testy/test6
```

As the printout says, tests no. 1 and 2 finished with the `OK` status, which means that the tests didn't generate any errors. Test no. 3, on the other hand, was skipped on user's behalf. Test no. 4 ended up with an error, while test no. 5 also failed, but this time the failure was expected (the result code `EXF` stands for *expected failure*). Finally, test no. 6 didn't produce any errors, but its actual output to the screen was different from the expected output stored in the respective `.out` file.

When the whole series of tests is executed, following the TDD approach it is necessary to fix the functions, whose tests have failed. Once the code is fixed, the tests have to be re-run. In

order to minimize the effort, the toolbox allows to run again only those tests which previously failed. To do so, the user can filter out the results according to respective result codes. For instance to select only those tests which failed with an `ERR` code, one could use

```
N = find(D, 'status', 'ERR')
```

which will create a new build object `N` representing only failed tests. One can then re-run these tests by calling

```
D = run(N)
```

One can also use negation of statements. For instance to select all tests with return codes different from `OK`, one can use

```
N = find(D, 'status', '~OK')
```

to select such tests. This filtering procedure helps the developer to focus only on a subpart of the whole pile of tests, hence minimizing the computational load needed to execute them.

4 Conclusion

In this paper we presented a new MATLAB toolbox which can be used for unit testing. The toolbox allows users to create, manage, and execute a pile of tests which verify the correctness of the tested code. Furthermore, the toolbox allows to verify the actual outcomes of the tests against expected outcomes. The toolbox is available upon mail request directly from the authors.

Acknowledgment

The authors are pleased to acknowledge the financial support of the Scientific Grant Agency of the Slovak Republic under grants No. 1/3081/06 and 1/4055/07 within the framework of the European Social Fund (PhD Students for Modern Industrial Automation in SR, JPD 3 2005/NP1-047, No 13120200115).

Ing. Michal Kvasnica
Institute of Information Engineering, Automation, and Mathematics
FCFT STU in Bratislava, Radlinského 9, SK-812 37 Bratislava, Slovakia
e-mail: michal.kvasnica@stuba.sk

Ing. Ľuboš Čirka
Institute of Information Engineering, Automation, and Mathematics
FCFT STU in Bratislava, Radlinského 9, SK-812 37 Bratislava, Slovakia
e-mail: lubos.cirka@stuba.sk