TALLINN UNIVERSITY OF TECHNOLOGY
School of Information Technologies

Silver Schnur 155387IAPB

DEVELOPMENT OF AN AUTOMATED TESTING SYSTEM FOR SWI-PROLOG

Bachelor's thesis

Supervisor: Evelin Halling
MSc

Tallinn 2018

TALLINNA TEHNIKAÜLIKOOL
Infotehnoloogia teaduskond

Silver Schnur 155387IAPB

SWI-PROLOGI AUTOMAATTESTIMISSÜSTEEMI ARENDUS

Bakalaureusetöö

Juhendaja: Evelin Halling
MSc

Tallinn 2018

Author’s declaration of originality

I hereby certify that I am the sole author of this thesis. All the used materials, references to the literature and the work of others have been referred to. This thesis has not been presented for examination anywhere else.

Author: Silver Schnur

25.05.2018

Abstract

The purpose of this thesis is to develop a new SWI-Prolog tester, which would replace the existing one. The new tester should provide better feedback to students and help write more complex tests. The focus was placed on improving the way failed tests were detected, since the previous tester often could not discover them.

Thanks to the work detailed in this thesis a new tester was written, which uses a modified fork of the PlUnit testing framework. The framework was modified to output results in an easily parsable format and provide needed features to test writers. The created tester will be integrated into the automated testing system used by the Tallinn University of Technology.

This thesis is written in English and is 23 pages long, including 7 chapters, 5 figures and 2 tables.

Annotatsioon

Swi-Prologi automaattestimissüsteemi arendus

Antud lõputöö eesmärgiks on arendada uus SWI-Prologi tester, mis asendaks praeguse testri. Uus tester peaks andma paremat tagasisidet ja võimaldama testide kirjutajatel edastada testide kohta rohkem informatsiooni ning kirjutada keerulisemaid teste. Eriline rõhk tuvastamisel, millised testid kukkusid läbi ja miks, sest praegune tester ei suuda tihti seda tagastada.

Töö tulemusel valmis uus tester, mis kasutab modifitseeritud PlUniti raamistiku. Raamistiku laiendati, et see annaks lihtsamini töödeldavat tagasisidet ja pakuks testide kirjutamisel uusi võimalusi. Loodud tester integreeritakse Tallinna Tehnikaülikooli automaattestimise süsteemi.

Lõputöö on kirjutatud inglise keeles ning sisaldab teksti 23 leheküljel, 7 peatükki, 5 joonist ja 2 tabelit.

List of abbreviations and terms

SWI-Prolog	Dialect of Prolog taught in Tallinn University of Technology
TUT	Tallinn University of Technology
CSV	*Comma-separated values*
Git	A popular version control system
Moodle	Free open-source learning management system

Table of Contents

List of figures

List of tables

1 Introduction

When learning new programming languages, an important part of the process is writing code and verifying that it works. To help check that code written by a student is correct and performs its task correctly Tallinn University of Technology has developed an automated testing system, which is used by many courses. The system runs the students' solutions against tests written by the lecturers and returns the result via email and publishes the results on a webpage.

An important requirement of the testing system is the ability to validate code written in different languages. For that purpose, every supported language must have its own tester. The goal of this thesis is the development of one such tester, which will replace the currently existing SWI-Prolog one.

1.1 Overview of the automated testing system

The main components of the automated testing system are TUT's Git server, a Moodle based webpage, a mailing server, and the testers. First, an automatically graded project is created using the webpage. When the project has been created, the information necessary for testing, such as chosen tester and test files, is passed on to the rest of the system.

The testing is triggered by a Git hook in TUT's Git server. When a student pushes their commits to their repository, the system pulls the changes and runs the associated tester with the tests. Different projects are associated with different folders and the project is selected according to the files changed by the commit. When a commit touches files associated with multiple commits or multiple commits are pushed at the same time, the tester is rerun for each project and commit.

Finally, the results of the tests are published to the webpage and an email is sent to the student. On the webpage students can see the contents of the tested files and the associated grade. In addition to that, teachers can access additional information to help detect failures in testers.

1.2 Issues with the current tester

The main issue with the current SWI-Prolog tester is that it frequently fails to inform students of failed tests. The email composed by the tester should have 3 sections: the number of test passed, the names of the failed tests and a session ID, which is used to debug the automated testing system in case of failure.

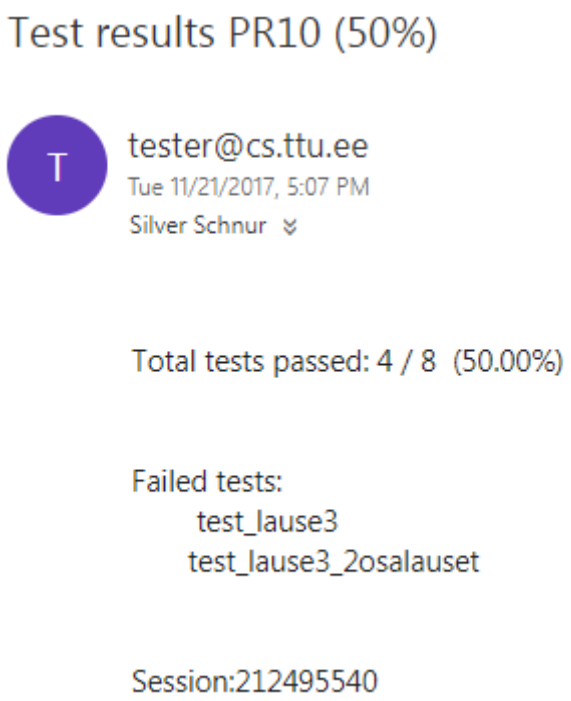

Figure 1. Example of an email sent by the tester

As seen on Figure 1, 4 out of the 8 tests failed, however the section which lists failed tests only has the names of 2 tests. This is bad, because the only ways to fix the failing tests are to either change random parts of your code or ask the lecturer or an assistant to check what went wrong. This leads to a lot of wasted time on both parts and a decrease of overall moral.

A secondary but also important problem is that the tester uses PlUnit which lacks several features, which could be used to improve the grading and understandability of assignments. The new tester should help alleviate the situation and add additional capabilities to bring it closer to the capabilities of the current Java tester.

2 Requirements for the new tester

The new SWI-Prolog tester should obviously fix the problem with silently failing tests. In addition, it would also be helpful if it output the tests which pass, since that may help narrow down any problems in the code. A smaller but still nice-to-have feature would be to include any syntax errors; however, this is less critical since students are not supposed to push completely untested code.

The new tester should also add support for features most missed by the test writers. The requested features are: descriptions, test specific timeouts, weights, and a dependency system. Descriptions would be more suited to conveying additional information about tests than overlong names. This is important since from the students' point of view the entire automated testing system is a black box.

Test specific timeouts are necessary to limit the amount of time the tester spends in an infinite loop. The automated testing system does set a timeout for every tester, but this is generally far longer than necessary for most assignments, since it needs to accommodate tests with significant runtimes. Having a shorter delay before a test is marked failed also means that useful results can be provided for tests which are run after the one which got stuck in a loop.

Weights and a dependency system can be used to create more interesting testing and grading scenarios. Instead of a flat one point per passed test, weights can be used to award more points for certain tests than others. Combined with a dependency system, which would automatically fail tests, when their prerequisite tests have not passed, a much bigger initiative can be given for completing harder parts of an assignment.

Finally, if practical, the new tester should allow reuse of the existing SWI-Prolog tests. That means the tester should use PlUnit or PlUnit syntax with the minimal amount of necessary changes. This is also important because the current test writers already have experience with PlUnit and it would lessen the burden of switching to the new tester.

3 Architecture of the new tester

The current tester is a Python script, which copies the tests and solution into a directory and runs the PlUnit based tests. It uses regular expressions to parse the output printed by the PlUnit predicate run_tests. Because only the most common cases are covered, the tests can silently fail when something unexpected happens. Most often it is because of the test body throwing an exception.

It was elected to keep the same overall architecture because of the frequent changes made to the automated testing system. While it is possible to write everything in SWI-Prolog without giving away features, either by having a separate script or a monolithic design, SWI-Prolog is obscure being the 34th language by popularity on the TIOBE index and therefore likely to impede further development of the automated testing system due to lack of proficiency in it [1]. In addition, the rest of the system is already written in Python.

The decision was made to output the results from the SWI-Prolog part of the tester using a CSV format to make it easier to parse for the Python script, which will deal with calculating the score based on test weights and composing the email. A more direct solution using PySWIP, which would allow directly running SWI-Prolog queries from the Python script, was considered. However, it was abandoned in favour of a simpler and more concrete interface, which would allow to modify the tester without any knowledge of Prolog.

To help debug failing tests the full output of standard error is included in the email sent to students. While this may help the students access information about tests which they are not supposed to, it was considered an acceptable risk due to how hard learning Prolog can be for someone used to more conventional programming languages. However, everything written to the standard output by submitted code can only be seen by the teachers.

Since an entry point is needed and all currently existing tests use files with names matching the format "test_pr[project number].pl" for it, the tester will search for files in the testing folder named according to that pattern. In case multiple files are found, they are both run and a separate grade is published for each file.

3.1 Unit testing framework

Unfortunately, due to the relative obscurity of Prolog the choice of available unit testing framework is limited. Mostly it seems that there exists one major framework for every dialect of Prolog. While it is possible to write code, which can run on multiple dialects (PlUnit itself can run on SICStus and YAP in addition to SWI-Prolog), most frameworks do not bother with it. The only other framework which runs on SWI-Prolog, Crisp, was too small and lacking in features to be of significant help. [2]

Due to these problems, the decision was made to keep using PlUnit and to extend it to cover the requested features. Unfortunately, it was impractical to extend PlUnit any other way than forking, since the features required significant changes in the framework to support them. Most notably implementing timeouts and the dependency system required changes in the way the tests are executed.

The desire to keep the usage of the added features similar to the existing ones also contributed to the argument for forking. Since the option lists used by the tests are type checked on runtime, it was impossible to add anything to them from outside the framework. In addition, the usage of PlUnit and its syntax it's necessary to add only an import statement for the new framework to make the old tests compatible with the new tester.

The fork also makes more use of marking tests as blocked. In PlUnit the only time a test is marked blocked is when that option is set on the test. In addition to that, the fork will also mark tests as blocked, when one of their prerequisites, such as setup or tests which it depends on, fail.

The name of the fork is EPlUnit, which stands for Extended PlUnit. It is unlikely that it will ever be fully merged back into PlUnit as some features, most notably weights, have little to no use outside of the context of graded assignments. The fork is meant to be installed in the SWI-Prolog libraries folder, so that all tests can use it.

4 Added Features

All the required new features mentioned in the requirements were added. Their intended usage, implementation details, and syntax will be demonstrated in the order of implementation complexity.

4.1 Description

The aim of a description is to provide a way to give more information about a test instead of having very long names for tests. The description does not affect the way a test functions and is included in the feedback given to students. The description must be a textual value – accepted types are atom, string, chars and codes.

```
:- use_module(library('eplunit')).
:- begin_tests(description).

test(just_description, description("This test will pass")) :-
    true.
test(with_other_options, [description("The expected value of A is 3"), true(A
=:= 3)]) :-
    A = 3.
test(no_description) :-
    true.
test(multi_line_description, description("multi
line
description")) :-
    true.

:- end_tests(description).
```

Figure 2. Example usage of description

As it can be seen in figure 2, the description can be used alone or with other test options. The description can also contain line breaks.

4.2 Weight

The objective of weights is to have different tests a bigger or smaller effect on the student's grade. The weight of the test has no impact on the way or order a test is executed. By default, the weight of a test is 1. Weight can be used to reduce a test's impact on the grade to zero. The value of a weight must be a nonnegative integer.

```
:- use_module(library('eplunit')).

:- begin_tests(weight).

test(weight_2, [weight(2)]) :- true.
test(weight_3, [weight(3)]) :- true.
test(weight_3_fail, [weight(3), true(A =:= 2)]) :-
    A = 3.
test(weight_0, [weight(0)]) :- true.

% test(weight_invalid, weight(1.5)) :- fail. % Valid weights are only
nonnegative integers
% test(weight_invalid, weight(-1)) :- fail. % Valid weights are only
nonnegative integers

:- end_tests(weight).
```

Figure 3. Example usage of weights

The commented-out test will cause a domain_error to be thrown when they are left in as the options passed to tests are checked at runtime. In its current state these tests would give a grade of 62.5% since the third test fails and the last one does not contribute to the total.

4.3 Timeout

Adding timeouts to tests required more significant changes to PlUnit than the previous two features. It was somewhat surprising that PlUnit did not support setting timeouts to tests, especially considering that SWI-Prolog has a call_with_time_limit predicate. The feature was implemented by wrapping every test execution point with that predicate and have EPlUnit catch the thrown time_limit_exceeded exception. By default, every test has a time limit of 3 seconds. That can be changed by calling the set_test_options predicate before executring the tests.

```
:- use_module(library('eplunit')).
% :- set_test_options([timeout(5)]). % Un comment this line to increase the
default timeout
:- begin_tests(timeout).

test(pass, timeout(0.1)) :- pass
test(pass2, [timeout(0.1), true(A =:= 3)]) :- A = 3.
test(fail, [timeout(0.1)]) :- sleep(1).
test(timeout_by_default) :- sleep(4).
test(timeout_even_with_increased_limit) :- sleep(6).
% test(impossible_timeout, [timeout(0)]) :- true. % Domain error
% test(impossible_timeout2, [timeout(-1)]) :- fail. % Domain error

:- end_tests(timeout).
```

Figure 4. Example usage of timeout

The first two test always pass and the third always fails. The fourth test will timeout with the default time limit, however, by uncommenting the second line it will pass. The fifth test will fail even with the extended time limit. The last two commented-out tests show that setting the timeout to a nonpositive value causes a domain error, since that makes no sense. The same applies to the default time limit.

4.4 Dependency system

The dependency system required the most work, because it affects the order tests are executed and can even block tests from executing. It is used to have a more defined boundary between grades for more advanced version of the same task and enforce constraints such as: tests belonging to the block A must be passed before tests in block B can be attempted.

The dependency system is composed of two parts: the first part is run before any tests and it determines the order in which test blocks should be executed. The second part verifies that none of the tests belonging in the prerequisite test blocks have failed. If any prerequisite tests have failed or been blocked all the tests which belong in the current test block will be marked as blocked.

In order for the dependency system to work, the dependency graph between different blocks must be a directed acyclic graph. If a cycle is detected in the dependency graph, then a invalid_dependency_graph exception is thrown and the testing is aborted. If a test block that is marked as a dependency is not found, then an existance_error is thrown and testing is also aborted.

```
:- use_module(library('eplunit')).

:- begin_tests(final, [dependson([basic, advanced])]).
test(final_blocked) :- true.
:- end_tests(final).

:- begin_tests(advanced, [dependson([basic])]).
test(advanced_fail) :- fail.
:- end_tests(advanced).

:- begin_tests(basic).
test(basic_success) :- true.
:- end_tests(basic).

% fails due to cyclic dependency
% :- begin_tests(loop, [dependson([loop])]).
% test(advanced_success) :- true.
% :- end_tests(loop).

% fails due to nonexistent dependency
% :- begin_tests(invalid_dependency, [dependson([invalid])]).
% test(advanced_success) :- true.
% :- end_tests(invalid_dependency).
```

Figure 5. Example usage of the dependency system

Due to the dependency system, the testing blocks are actually executed in the reverse order. Since test advanced_fail will fail, test final_blocked will be marked as blocked instead of passed. Uncommenting either of the commented-out test blocks will cause an exception to be thrown at run time, due to the previously described constraints.

5 Testing

While all added features have a test to demonstrate their usage, the main focus of testing is on finding out cases tests which might silently fail. For that purpose, a test setup was created to run the author's Git repository from the course "Logic Programming" against the same course's tests. Unfortunately, due to the possibility that the tests will be reused next year, neither the tests nor the authors solutions will be published.

The test was first will be ran with the current tester to establish a baseline and then with the new tester to compare the results. The test tracks the number of times a project's tests were run and the amount of passed, failed, blocked, and silently failed tests. The used Git repository has 81 commits, out of which 71 contain files covered by the assigned 12 automatically graded projects.

In both cases, the total number of run tests and the number of passed tests should be identical, since the same set of tests is used. It is expected that the number of silently failed tests will decrease and the number of failed and blocked tests will increase. The goal to reach zero silently failed tests.

In the following tables the project column records the name of the folder for each project. The second column shows how many tests were associated with each project. The runs column shows how many times all the tests associated with the project were run. The last four columns show how many tests across all runs passed, failed, were blocked or silently failed. The second to last row shows the sum of all the preceding numbers in the same column. Total number of test run is the sum of all passed, failed, blocked and silently failed tests.

Project	Tests in project	Runs	Passed	Failed	Blocked	Silently failed
PR01	2	1	2	0	0	0
PR02	24	1	24	0	0	0
PR03	10	3	30	0	0	0
PR04	8	2	8	0	0	8
PR05	28	15	387	2	0	31
PR06	3	10	16	14	0	0
PR07	24	2	48	0	0	0
PR08	8	18	90	6	0	48
PR09	4	12	25	17	0	6
PR10	8	7	41	8	0	7
PR11	4	4	6	0	0	10
PR12	3	3	7	2	0	0
Total:	127	78	684	49	0	110
Total number of tests run:			843			

Table 1. Test results of the current SWI-Prolog tester

As seen in table 1, more than two thirds of failed tests failed silently, which was a significant obstacle for students, since they had no idea about which part they should focus on to improve their grade.

Project	Tests in project	Runs	Passed	Failed	Blocked	Silently failed
PR01	2	1	2	0	0	0
PR02	24	1	24	0	0	0
PR03	10	3	30	0	0	0
PR04	8	2	8	8	0	0
PR05	28	15	387	33	0	0
PR06	3	10	16	14	0	0
PR07	24	2	48	0	0	0
PR08	8	18	90	54	0	0
PR09	4	12	25	23	0	0
PR10	8	7	41	15	0	0
PR11	4	4	6	6	4	0
PR12	3	3	7	2	0	0
Total:	127	78	684	155	4	0
Total number of tests run:			843			

Table 2. Test results of the new SWI-Prolog tester

Compared to the results of the current tester, the new one shows significant improvement in reporting failing tests. The number of silently failing tests has dropped to zero, which means the new tester accomplishes its main goal of providing better feedback for failing tests.

6 Unresolved issues

There are some topics which are not covered by this thesis, yet could benefit from more attention in the future. The most beneficial of which is backporting some of the new features of EPlUnit into PlUnit. While weights certainly have limited value outside of the designed system, at least test specific timeouts can help design time sensitive systems. In addition, getting necessary features merged back into PlUnit would mean less overhead in keeping the developed tester up to date with new SWI-Prolog releases.

Another problem is the fact that the developed tester has little protection against students committing code which could print out test inputs or tests. The current design protects against only the simplest of such attacks – using write or print predicates to write information to standard output. There are currently no protections against printing writing anything to standard error, since everything written there is included in the email, which is sent to students, to make debugging easier.

Finally, there some PlUnit features which do make sense or work well in the context of this tester. Most notable of those is the fixme option on tests and test boxes. It is meant to be used for tests, which are expected to fail but will not hamper the ability to execute other tests [3]. While it may sound useful, it is usually expected that the students code passes all tests. The main reason to avoid using fixme when writing tests is that the way it is implemented means that tests marked with it provide poor feedback in case of failure.

7 Summary

TUT uses an automate testing system, which lets programming courses create automatically graded assignments. This helps conserve time by avoiding most of the manual checking of assignments. The system depends on different testers to provide the necessary testing functionality for taught programming languages. The SWI-Prolog tester currently in use had been identified as a particularly lacking in feedback and features.

According to the work laid out in this thesis, a new tester was designed and implemented. Based on a test with real world data it provides better feedback to students than the current one. In addition, several new features were implemented which will help in writing more advanced tests and testing scenarios. Switching to the new tester should be easy, since it uses the same syntax as the previous tester. Existing tests only need to import the new testing framework to use the benefits of the new system.

The new tester should be integrated into the automated testing system with little difficulty and should be in use by the autumn semester of 2018.

References

[1] “TIOBE Index,” [Online]. Available: https://www.tiobe.com/tiobe-index/. [Accessed 23 05 2018].

[2] D. Feng, “Investigating Prolog Based Unit Test,” Leuven, 2012.

[3] J. Wielemaker, “Prolog Unit Tests,” [Online]. Available: http://www.swi-prolog.org/pldoc/doc_for?object=section(%27packages/plunit.html%27). [Accessed 24 5 2018].

Appendix 1 – Reference to the public repository

The developed tester, EPlUnit, and tests have been published in a public Github repository. It is accessible from https://github.com/thunkk/swipl-tester.