2013 Third Eastern European Regional Conference on the Engineering of Computer Based Systems

Selection and Prioritization of Test Cases by Combining White-Box and Black-Box Testing Methods

Sandra Kukolj, Vladimir Marinković, Miroslav Popović
University of Novi Sad, Faculty of Technical Sciences
Novi Sad, Serbia
sandra.kukolj@rt-rk.com, vladimir.marinkovic@rt-rk.com, miroslav.popovic@rt-rk.com

Szabolcs Bognár
University of Szeged, Institute of Informatics, Department of Software Engineering
Szeged, Hungary
bszabi@inf.u-szeged.hu

Abstract—**In this paper, we present a methodology that combines both white-box and black-box testing, in order to improve testing quality for a given class of embedded systems. The goal of this methodology is generation of test cases for the new functional testing campaign based on the test coverage information from the previous testing campaign, in order to maximize the test coverage. Test coverage information is used for selection of proper test cases in order to improve the quality of testing and save available resources for testing. As an output, a set of test cases is produced. Generated test cases are processed by the test Executor application that decides whether results have passed or failed, based on the results of image grabbing, OCR text extraction, and comparison with expected text. The presented methodology is finally validated by means of a case-study targeting an Android device. The results of the case study are affirmative and they indicate that the proposed methodology is applicable for testing embedded systems of this kind.**

Keywords-embedded systems; quality assurance; black-box testing; white-box testing; code coverage

I. Introduction

Quality assurance, such as the software testing of embedded systems, is an important issue. Software testing is a diverse field that supports different software requirements, such as functional correctness, usability, robustness, efficiency, backward compatibility, etc. Black-box (BBT) testing is one approach for automated functional testing in TV and multimedia technology. The main flow of the process includes test case generation from the specification. Some, or all of them, are selected to be executed in a certain order. Some other testing techniques in systems for automatic functional testing of digital TV sets have been proposed in [1,2], as the best possible choice in the development of multimedia devices.

In this paper, a methodology that helps the testing of embedded systems software/hardware architecture is presented. Besides BBT, it uses white-box software testing technique. It can be used to evaluate and improve the quality of functional testing by determining untested parts of the embedded architecture. In that way, black-box testing of embedded systems is aided using white-box testing techniques. The goal is to provide valuable information of the executed black-box tests quality.Using a technique called code instrumentation, the *test coverage* (or briefly coverage) and *traceability* information are extracted from the execution *trace*.Traceability model is a list of methods that are being reached during the execution of the test case.

Execution of tests is done automatically by using a PC based application for control, development and execution of automated test cases. This automation increases the quality of the software under test. The outcome of this execution is a pass/fail decision for test cases set by the requirements. These results can be used to select or prioritize test cases during the next execution. Also, the set of simple test cases can be used to find uncovered paths and branches in executed software.One of many possible applications of this kind is presented in [3].

In the next section, some of the reasons and methods for test case selection are presented. Then, in the third section, each proposed criteria for prioritization is briefly described. Fourth section gives details of the described module implemented in the BBT plug-in (device driver). The fifth section presents an implementation of the presented module as a BBT device driver, and in the sixth section, the automated testing process of the STB (set-top box) is described. These consumer devices are employed in Digital Video Broadcasting Networks, and are mainly used for digital television reception and some interactive services [4,5,6]. Finally, in the seventh and eighth section, we describe the validation of the proposed system, along with the instrumentation process, in order to obtain coverage data.

II. Test Case Selection

Test case selection and prioritization make the testing process much quicker. If the proper test cases for the testing process can be selected, than less resources are required to perform testing that produces some of the following goals: covering the methods, the code, the methods, the branches, or reach a given level of any other type of test coverage. To select or prioritize test cases, some enumerable properties of them could be measured. Selecting a set of test cases can decrease the needs of testing, but also save time and money. There are two options for selecting desired test cases: either the selection of all available test cases, or the selection of only those test cases which cover methods that have been changed in the previous executions. The requests for all test cases, or only selected ones, are generated by the instrumentation tool, which represents another white-box testing module. The available input information with the test case list is in the form of text files, generated by the instrumentation tool. Naturally, there is no repetition of test cases in the resulting output, whereas two or more methods can be covered with the same test cases. The output is generated by the test executor application, as a project file that contains sorted test cases.

III. TEST CASESPRIORITIZATION

Prioritization of test cases is mostly based on the coverage information and it gives optimal usage (saving of available resources). The main goal is coverage of all methods in a STB's application with minimum number of test cases. Another goal is to reach high speed in this process which can be realized by executing a larger number of short test cases. The prioritization process provides a list of test cases that are ordered by previously selected priority criteria, i.e. test case properties.

A. Prioritization based on longest traces of execution

One of several prioritization criteria is based on the information about trace lengths, which are given in the input file, for each test case. Namely, these traces represent the number of executions of different methods associated with a particular test case. During the execution of a specific test case, some methods can be called more than once, or not called at all. The aim of this criterion is to calculate the priority of test cases in descending order, where test cases that cover methods with longer traces should have higher priority. These traces represent the runtime method call, and in this context, the methods trace is the number of their calls within one test case. The term *trace length*, or just *trace*, is the number of method calls concerning one particular test case. So, test cases with longer traces are in fact the ones that have reached largest number of methods.

Trace length calculation is presented with formula (1), as the trace length from previous execution of test case subtracted from trace length of the last executed test case.

$$traceLen = cum\_trace\_len_n - um\_trace\_len_{n-1} \quad (1)$$

B. Prioritization based on coverage

This criterion prioritizes test cases on the basis of given method coverage information. First, the file with all the covered methods for each test case is read, and a map with this information is created. Then, for each test case, methods that are covered by a certain test case are found. The test case with the highest priority is located. The higher coverage test cases are placed on the top of the list, while the priorities for all other test cases are recalculated. This recalculation is done by observing methods that have not yet been covered. In the end, the structure with the resulting outputs is created.

C. Prioritization based on failing test cases

First, the file that holds all the pass/fail results for certain test cases is read. This file also contains the execution date for every test case. Then, these dates are used for calculation of the time interval in which test cases have failed most frequently. Test cases that have failed less frequently have lower priority than those that failed more often.

IV. AUTOMATED TESTING PROCESS

As already mentioned, BBT only checks the functionality of the system. There is no knowledge about the internal structure of the object under test, and the tester only observes how the object responds to the given set of inputs. The WBT method does the opposite. It tests the code and searches for errors or dead code. Relevant information about the source code can be collected. This helps BBT in making code check during functional testing or in finding out if additional test cases are necessary. On the other hand, this information could reduce the number of test cases needed to cover all the methods, or an order of test cases could be made such to select the most efficient ones first. Automating the test case generation process provides a means to ensure that test cases have been derived in a consistent and objective manner and that all system requirements have been covered [7].For testing purposes of the BBT, the Executor application is executed. It automatically defines test cases and executes them as input scripts. These special input scripts that contain previously generated set of test cases are used for making and executing test plans. The output is generated also as a set of tests, but with ordered test cases. Automatic test case generation increases adaptive capabilities of the system. In this way of test cases generation, only a subset of test cases is actually executed on the object under test. On the basis of the results, a decision is made on which tests need to be performed further. Since the internal system structure is not known, test cases are generated based on the specification of the functionality of the object, in order to test the complete functionality of the object under test.

V. BBT PLUG-IN TOOL

The BBT plug-in tool is realized as a device driver that plugs into the BBT system. The device driver module has its own API that enables interaction with the BBT application and simplifies the realization of the application module. The plug-in connects to the device and sends control messages. It makes the proper calculations and returns not yet covered methods names in a text file (coverage information).

Coverage information is necessary to perform any coverage-based testing tasks, like test case selection, test case prioritization, test case generation, etc. The trace, on which this coverage information is usually based on, may contain relevant information about the actual program execution, like method-entries or returns, line number of executed instructions, etc. Traces created by test cases show the connection between the code and the test case. The functional coverage connects test cases and the specification. It shows which test case refers to which functionality, defined in the specification. Code coverage gives the information of how much code has been reached by the test case during the execution.

The plug-in calculates the coverage for each test case, and then all the information is written into a file in the testing directory. Traceability value can be computed based on the connection between the functionalities, the test cases, and the coverage information. This means that we can establish traceability links between test cases and source code. Through these links and, the existing traceability links, we can define missing links between other elements (e.g. requirements and source code). Even if traceability exists between code and higher level elements, we can compute different type of coverage based on code coverage (e.g. compute requirements coverage if requirements-code traceability links are available).

VI. Code Instrumentation

A necessary step in coverage calculation is the instrumentation of the whole code before test execution. By instrumenting a certain program, it is possible to determine, among others, the number of decisions (branches), previously exercised with the given test case.

The used instrumenting technique enables automated instruction inserting that simultaneously generates trace and coverage information for the executed code. Having the DLL for the *instrumenter application*(a device that instruments the binary code), the coverage information is obtained after the execution of test cases. It prepares the code and the environment to send information during test execution by inserting feedback instructions into the source code or inserting interrupt instructions and additional code to the binary.

This DLL serves as an addition for the BBT system for automated test execution. Executable code is necessary for the presented instrumentation method, as a part of WBT. The code gets executed during program testing in order to measure the test coverage, which is calculated in percentage, according to formula (2), where *rm* represents the number of reached methods during the execution, and *am* represents the number of all the methods:

$$coverage = (rm/am) \times 100 \qquad (2)$$

This coverage gives information about how many lines and which lines are covered, or which methods (their IDs), etc. Additional lines provide the collection of the number of instruction/method calls.In Android systems, we need a server application that collects the information and communicates with the BBT device. Another addition is the DLL for the BBT device called the *CoverageClient* that needs to be included into every test case. It is a communication endpoint that makes the computations.

VII. Validation

A python script is written for the execution of a convenient calculator application, which is controlled using special macros for automation purposes. This application is used for testing the prioritization of test cases. The numbers are generated randomly and therefore the actual results cannot be compared with the expected ones. Currently calculated test script result (extracted in the file) is compared with the file generated from the OCR. The image is grabbed using a single channel device called NAViC (Network Attached Video Capture). It captures uncompressed images directly from the digital STB and DTV.

The tester communication unit (TCU) consists of system user interface, use-case scenarios database (as a collection of various test cases), and of course, the results. The system user interface is intended for specifying the use-case scenario of a DUT (Device Under Test, in this case, the Television Set), starting the testing process and examining the final testing results. The other side of the system is the processing unit (PU) that performs the functional failure detection and controls the DUT.A test case is only a calculation for raandomly generated numbers, while two or more test cases make up a sequence of calculations. Each test case executes one operation on two randomly generated operands. The coverage percentage is also generated, and it is accumulated after every other test case is executed, what can then be called as a sequence of calculations, therefore a formula for calculating cumulative coverage percentage is given as:

$$cum\_cov = \frac{\bigcup_{n=1}^{N=10} cov\_met\_num}{\bigcup_{n=1}^{N=10} met\_num} \times 100 \qquad (3)$$

This formula shows that cumulative coverage is acquired by division of all covered methods (both covered and uncovered methods).Because coverage is cumulative, the percentage value of the coverage, as well as the covered method number stay the same after testing is performed with a randomly chosen operation that has already been exercised. Table I represents the first testing campaign, while Table II is the second testing campaign. Table I gives an example of which methods have been covered by the presented test cases and which methods have been changed, but also what is the number of covered methods for each case of 30in total, as well as the execution runtime, trace length and outcome of the testing process. This table also gives an insight of test case description, the total number of covered methods and coverage for each test case.

Table II describes all four prioritization criteria for selection of all test cases with cumulative coverage for every criterion, based on which it is possible to reduce the number of test cases based on the same maximum coverage achieved (cutting the list of test cases from the second test case that reaches maximum coverage), and with this the execution runtime is reduced.

VIII. Conclusion

In this paper, a methodology of test case selection and prioritization in testing embedded systems is presented, combining white-box and black-box testing techniques. The execution environment provides coverage information, which can then be used to improve the testing quality.

TABLE I. TEST CASE DESCRIPTION AND RESULTS

First Testing Campaign							
Test Case ID	*Test Case Description*	*Covered Method Number*	*Covered Methods IDs*	*Changed Methods*	*Trace Len.*	*Execution Runtime [ms]*	*Test Outcome*
Sub_1	873-530=343	14/30	M1,…,M13,M14	M4,M5,M8	14	6511	PASS
Sub	957-649=308	15/30	M1,…,M13,M14	M4,M5,M8	14	7654	PASS
Mul_2	616*855=526.68	15/30	M1,…,M13,M15,M18	M5,M18	15	6970	PASS
Mul_1	424*838=355.31	15/30	M1,…,M13,M15,M18	M5,M18	15	5714	FAIL
Mul	43*29=1247	15/30	M1,…,M13,M15,M18	M5,M18	15	7003	FAIL
Div_1	158/278=0.57	14/30	M1,…,M13,M16	M12,M10	14	7367	PASS
Div	428/80=5.35	14/30	M1,…,M13,M16	M12,M10	14	7421	PASS
Add_2	605+915=1.520	14/30	M1,…,M13,M17	M2,M8,M12	14	6007	FAIL
Add_1	739+320=1059	14/30	M1,…,M13,M17	M2,M8,M12	14	6556	PASS
Add	862+74=936	14/30	M1,…,M13,M17	M2,M8,M12	14	6684	PASS

TABLE II. TEST CASES BEFORE AND AFTER PRIORITIZATION

Second Testing Campaign									
Test Cases Before Prioritization		*Longest Traces*		*Most Coverage*		*Most Additional Coverage*		*Most Failing Tests*	
Test Case	*Cumul. Coverage [%]*	Test Case	Cumul. Coverage [%]	Test Case	*Cumul. Coverage [%]*	Test Case	*Cumul.* Coverage [%]	Test Case	*Cumul.* Coverage [%]
Sub_1	**46.66**	**Mul_2**	**50.00**	**Mul_2**	**50.00**	**Mul_2**	**50.00**	**Mul_1**	**50.00**
Sub	**46.66**	**Mul_1**	**50.00**	**Mul_1**	**50.00**	**Sub_1**	**53.33**	**Mul**	**50.00**
Mul_2	**53.33**	**Mul**	**50.00**	**Mul**	**50.00**	**Div_1**	**56.66**	**Add_2**	**53.33**
Mul_1	**53.33**	**Sub_1**	**53.33**	**Sub_1**	**53.33**	**Add_2**	**60.00**	**Sub_1**	**56.66**
Mul	**53.33**	**Sub**	**53.33**	**Sub**	**53.33**	Mul_1	60.00	**Sub**	**56.66**
Div_1	**56.66**	**Div_1**	**56.66**	**Div_1**	**56.66**	Sub	60.00	**Mul_2**	**56.66**
Div	**56.66**	**Div**	**56.66**	**Div**	**56.66**	Div	60.00	**Div_1**	**60.00**
Add_2	**60.00**	**Add_2**	**60.00**	**Add_2**	**60.00**	Add_1	60.00	Div	60.00
Add_1	60.00	Add_1	60.00	Add_1	60.00	Add	60.00	Add_1	60.00
Add	60.00	Add	60.00	Add	60.00	Mul	60.00	Add	60.00

More specifically, coverage data is used to select appropriate test cases and sort them in required order, based on the given specification, i.e. coverage criteria. Developers and testers use code coverage to ensure that all or significant part of all statements in a program have been executed at least once during the testing process. This approach allows only the required tests to be executed and increases the overall usefulness of the test suite. Selection of test cases is done so that available resources can be optimized, and it represents a new approach in this field of work. This approach makes it possible to get the same coverage, but with less test cases. The goal was to find the connection between requirements, test cases and methods. The advantage is that after a particular prioritization criterion, redundant test cases are discarded for next execution. Another advantage is the fact that after the reduction of test cases has been done, coverage percentage stays the same. Future work could be to automate test case generation based on the requirements specification.

ACKNOWLEDGMENT

This paper was partially supported by the IPA project CIRENE, Serbian Ministry of Edu. and Science, Grants TR 32031 and III 44009, and Vojvodina Provincial sec. of Science and Tech. Dev., Grant 114-451-2434/2011-03. The authors would like to thank colleague Róbert Rácz for his contribution.

REFERENCES

[1] A. N. Rau, "Automated test System for Digital TV Receivers", *IEEE Int. Conference on Consumer Electronics*, pp. 228-229., 2000.

[2] Rama, R Alujas, F Tarres, "Fast and robust graphic character verification system for TV sets",*In Eighth International Workshop on Image Analysis for Multimedia Interactive Services 19*, 2007.

[3] D. Marijan, V. Zlokolica, N. Teslić, V. Peković, T. Teckan, "Automatic Functional TV Set Failure Detection System", *IEEE Transactions on Consumer Electronics*, Vol. 56, No. 1, February 2010.

[4] S. Pekowsky and R. Jaeger, "The Set-Top Box as Multimedia Terminal", *IEEE Transactions on Consumer Electronics*, Vol. 44, No. 3, pp. 833 - 840., Aug. 1998.

[5] T. Tekcan, V. Zlokolica, V. Peković, N. Teslić, M. Gündüzalp, "User-driven automatic test-case generation for DTV/STB reliable functional verification", *IEEE Transactions on Consumer Electronics*, Vol. 58, No. 2, pp. 587 - 595., May 2012.

[6] F. Lonczewski, R. Jaeger, "An extensible set-top-box architecture for interactive and broadcast services offering sophisticated user guidance", *IEEE International Conference on Multimedia*, 2000.

[7] S. J. Cunning, J. W. Rozenblit, "Automatic Test Case Generation from Requirements Specifications for Real-time Embedded Systems", *IEEE SMC '99 Conference Proceedings*, Vol. 5, pp. 784-789, 1999.