Specialized Methodology

Abstract

This report is part of the CIRENE project that aims (amongst other things) the definition of a general embedded system testing methodology. The project incorporates a pilot implementation of the methodology in a selected environment. This document is part of this pilot implementation as it describes the specialized methodology that can be applied in the selected environment, which is an Android-based Set Top Box (Digital TV decoder). The special methodology omits some parts of the general methodology, and details some other parts of it. Test prioritization and selection, code coverage measurements, code instrumentation and other activities are included and detailed.

This work were done in the Cross-border ICT Research Network (CIRENE) project (project number is HUSRB/1002/214/044) supported by the **Hungary-Serbia IPA Cross-border Co-operation Programme**, co-financed by the European Union.

The project is co-financed by the European Union

Table of contents

Abstract 1
1 Overview 3
1.1 General methodology 3
1.2 Special methodology 5
2 Prioritization/Selection 7
2.1 Test project 7
2.2 Selection 7
2.3 Prioritization 8
2.4 Cut 9
3 Not covered code calculation 10
4 Traceability computation 11
5 Instrumentation 12
6 Service application 13
7 Test execution 14
8 References 15

1 Overview

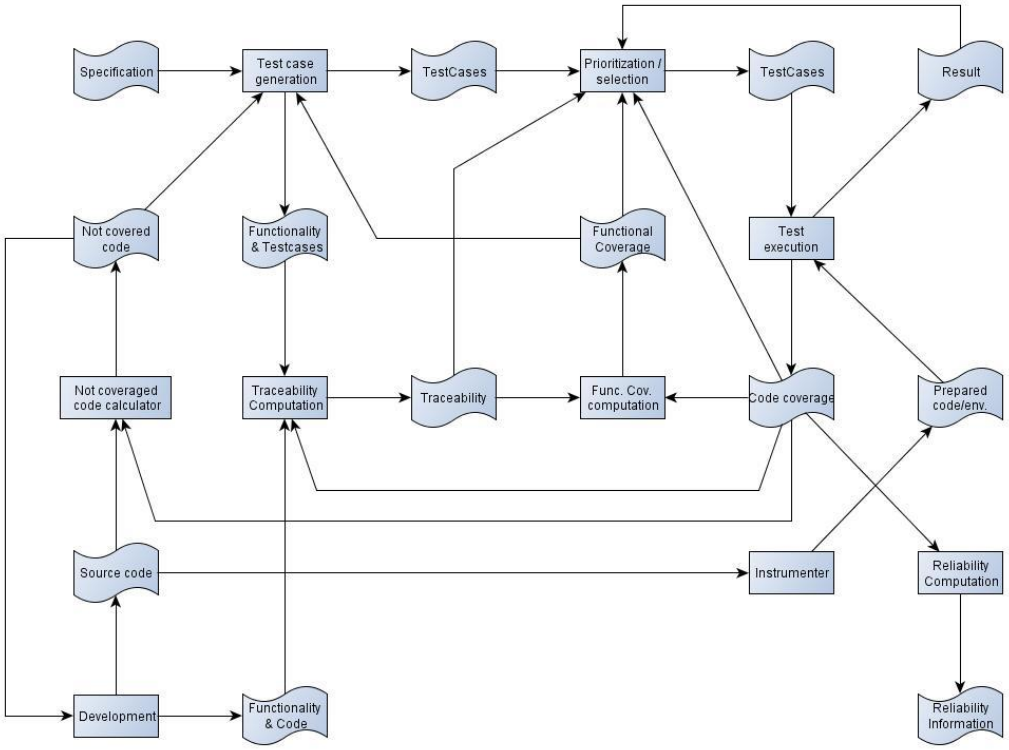
In the CIRENE project we would like to define an embedded system testing methodology, which is basically a Black-Box Testing (BBT) process that utilizes information collected by White-Box Testing (WBT) techniques. A pilot implementation of the proposed methodology on a specific embedded system is also the part of the project.

Prior to this document, a general methodology for functional testing of embedded systems [1] was defined, and a hardware-software analysis methodology [2] was created. Based on these previous documents, this one describes the methodology specialized for our target pilot system, an android-based Set-Top-Box device.

In the rest of this section, the general methodology is described then the specialized methodology is introduced. The remaining sections will describe different parts of the specialized methodology in details.

1.1 General methodology

The general methodology defined in this CIRENE project (overviewed in the next figure) utilizes different (BBT) process in order to test an embedded system, and uses White Box Testing (WBT) process to collect information and provide feedback to the BBT processes in order to enhance the quality of them. The methodology is general in the sense that it is not specific to any hardware or software configurations of embedded systems (although, naturally, the given embedded system has to fulfill some requirements before certain part of this methodology can be implemented).



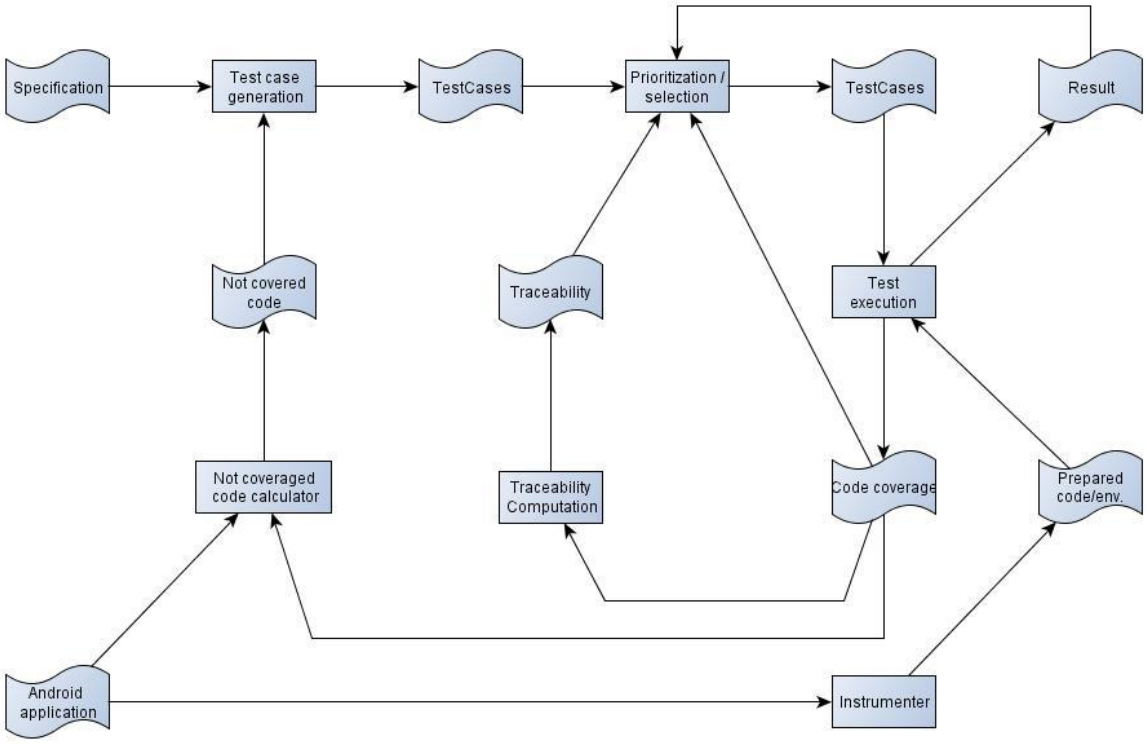
As we described in [1], BBT is testing only the functionalities of a system, while WBT explores the structure of it. BBT is interested in "what" the system does and WBT in "how" it does that. It is obvious that these two techniques complete each others. In our proposed methodology code coverage measurement (as a WBT technique) is used to collect information on the code.

In the proposed method test cases are first generated from the specification which also produces traceability links between various artifacts. Test selection and prioritization are then applied on the test cases, and the selected test cases are executed in the defined order. Test execution is done using a prepared executable that generates coverage data during execution. The test results and the coverage data can be used in the next selection/prioritization task. Code coverage can also be used for reliability computation, to determine not covered code, to enhance or confirm traceability information, or to compute functionality coverage. Functionality coverage can also be used for prioritization/selection. Additional test case generation can utilize information on not covered code and not covered functionalities.

The method is described in details in [1].

1.2 Special methodology

In the CIRENE project we perform a pilot application of the general methodology. It means that this general methodology is specialized for a given embedded device (an android-based Set-Top-Box), and some parts of the methodology are implemented. An overview of the specialized methodology can be seen in the next figure.



As it shows, we start the process with test case generation from specification using the MaTeLo tool [3]. Generated test cases are selected and prioritized based on previous execution results, code coverage and traceability information. (At the first time when this information is not available the trivial “select all” and “prioritize by creation order” methods are used). The next phase is the Test execution where all selected test cases are executed at least once in the order defined by the prioritization. During execution besides the pass/fail data we measure code coverage on the test cases. We use a special, instrumented executable version of the application to produce coverage information. Then we calculate traceability information from coverage data and use both the traceability and the coverage for the next prioritization and selection steps. From code coverage,

we also compute not covered code, which information is used in additional test case generation in order to raise code coverage of the test suite.

To measure the code coverage, we have to prepare programs under test. We instrument instructions in their code to generate the necessary information required to produce execution traces and coverage data. In the specific environment android applications are targeted by instrumentation, thus, we use Java byte code instrumentation technique to produce method level code coverage information.

2 Prioritization/Selection

Test case prioritization is a process that sorts the test cases according to some properties of them. The goal of prioritization is to define more important tests and execute them first. Test case selection is a process that chooses elements from the test suite based on one or more properties of the individual test cases or on the set of selected test cases. For different situations different properties can be used.

Usually the outputs of both processes are lists, but for the selection process it is a shorter list that contains only the selected items, and for the prioritization process it is an ordered list. Some algorithms can be used on the prioritized list to select some more important elements from it. We can cut the list according to some constraints, e.g. on the number of test cases or their expected execution time.

In our pilot implementation, prioritization and selection is used as well as the cut of the prioritized list. As our target testing system includes a test execution framework that works with test projects, we must be able to manipulate these test project descriptions.

2.1 Test project

We use the RT-Executor (RT-Ex) program for BBT device in this project. The RT-Ex uses special files for making and executing test plans. This file is the project file which is an executable file for the RT-Ex and contains the references to the test cases that need to be run during a testing sequence.

We select the test cases from a test suit based on some properties of them, like runtime, coverage or traceability. We make an ordered list from the selected ones and this prepared list will be the content of the project file.

2.2 Selection

We select from the test cases based on the information stored in the so-called “criterion file”. By comparing and combining the properties of the test cases we can select from them based on different aspects.

The criterion file stores the trace length, the coverage value and the execution time of all the test cases. This file generated during the first testing session when all the test cases are executed. We use this file as the container of all necessary information from the test cases.

In this project we use the coverage, trace and runtime information for base of selection. We give the opportunity to select the test cases with the largest coverage data, the longest trace path, the largest additional coverage data, or those which failed the most times. Test cases related to changed methods can also be selected.

We realize two cases of test selection:

1) **Select all the test cases.** It applied in the beginning of the testing process for information collection. It would be also necessary if the prioritization have the main role in the testing.
2) **Selecting those test cases which are related to changed program elements.** When a change occurs in a new program version, some of the test cases will exercise the changed part, and some will not. The assumption is that those test cases not executing changed parts of the program will probably produce the same result as before. So, selecting test cases exercise the changed code has only a low risk of miss some failures, but can speed up the test execution. Code coverage data can be used to perform this selection.

2.3 Prioritization

Test case prioritization is a process that sorts test cases based on some properties of them. In tis project we use mostly the coverage information, but almost any properties of the test cases could be used. A goal can be to cover all the methods in an Android application with minimal number of test cases. Other goal can be to reach high speed in testing process which can be effectuated by executing a number of short test cases.

With the prioritization process we provide a list of the test cases in which we have an order by the selected property (trace length, coverage data, additional coverage data, and fault rate) and from which we can cut or select the proper test cases for a good testing set.

In the pilot implementation prioritizations based on the following properties were used:

1) **Trace length:** We sort the test cases based on the trace length they walk through during their execution. The longest trace has the highest priority. The idea is that test cases with long traces may reach larger portions of the code. This is a simple heuristic trying to approach the result of the next selection.
2) **Number of covered methods:** We prioritize based on the number of methods the test case exercises during its execution. The more method is called during the execution of the test case, the highest priority that test case haw. The idea is the same as in the previous case, but

it is more precise as trace length does not always imply more executed methods (e.g. a loop calling a single method multiple times can produce very long trace).

3) **Number of additively covered methods compared the previously selected set:** If we select one test case, then we can tell what methods will be covered by it. If we would like to cover the most methods with the least number of test cases, we cannot simply choose the next test case based on simply the number of methods it covered. In this case we need to maximize the number of additionally covered methods. So we need to select the new test case by comparing its set of covered methods to the set of methods covered by the previously selected test cases. The test case that adds the most methods to this set is selected.
4) **Number of faults:** It utilizes the observations that some test cases (e.g. those that exercise the more complex parts of the software) tend to fail more frequently. Thus, these test cases should be checked first as these have the highest probability of failure. This prioritization will give a higher rank to the test cases that has failed more times recently.

The output of the selection method is an ordered list of the test cases. From this point, we can cut the list, we can select from it, or we can just use it as a test suit.

2.4 Cut

If we have a prioritized list of test cases, we can lessen the number of later executed test cases in a simple way: we cut the list. This will ensure us that even there are some constraints of test execution the most important test cases are executed.

In the CIRENE project we apply the next two type of cut:

- **Fixed number of test cases:** If the customer fixes or maximizes (or there are any other case exists that limits) the number of test cases that can be executed, than we need to cut the list at the proper place. After applying any of the previously mentioned prioritizations, we can simply select the top some elements from the sorted list.
- **Execution time-based cut:** If we have a limited time for test execution, we can cut the list of prioritized test cases according to their cumulative expected execution time. Simply summarize the measured execution time of the test cases (available from the previous executions) and cut the list when the time limit is reached.

3 Not covered code calculation

This process is important for a good testing, because if any parts of the code are not covered during the execution of the whole test suite than we can be sure that some parts of the program are not tested. In this case we have to overview and probably modify the code or the test set.

In our project we chose the method level of information collection, so we need to know the method list of the program under test (PUT) and the trace path of all the test cases. We make a list of all the methods in the PUT, than make other lists for every test case from which methods did they reached. Finally we need to sum all the reached methods and compare them to the full method list.

In our process the full method list is gathered during the instrumentation of the application, and our service application follows up the traces that the particular test case walks through. These are sent to the plug-in, which makes the proper calculations and gives back the not covered methods.

With this information we can try to make new test cases for the program to cover the not yet covered parts. If not possible to make new test cases, than the not covered parts are surely dead codes.

4 Traceability computation

Traceability links show the connections between different parts and levels of the software development process. It can show which test cases related to which requirements; which methods realizes which requirements; which methods related the most to particular test cases; and many other relations.

For example, code coverage is also some kind of traceability information, as it shows the connection between methods and the test cases executing them. In our pilot implementation we take into account two other artifacts: requirements and functionalities of the program under test. Traceability between test cases and requirements, test cases and functionalities can be derived together with the creation of black-box test cases. Using these known traceability links, we can compute missing direct links between the source code elements and the requirements and/or functionalities of the software. These links can later be used to compute functional or requirements coverage and determine the need for additional test cases.

The previously mention tool MaTeLo is able to track traceability between specification elements and test cases, and using test execution results, it can calculate Functional coverage. However, in this pilot we do not plan to compute functional coverage.

5 Instrumentation

The instrumentation is a process where we insert extra instructions into the code of the program which instructions can provide the measurement of the coverage and other values.

As in our pilot implementation we aimed method level coverage measurement, the instrumentation will insert necessary instructions at the beginning and at the end of all methods. These instructions will send signals to the service that collects the signals from the methods. Thus, we can ascertain which methods were reached during the particular test cases and also during the whole testing process.

This information is used to calculate traceability, trace length and coverage.

6 Service application

In Android systems there are applications running in the background, continuously or when they have been called. These are the Services. In our test system implementation it is necessary and important to have an application that is continuously running, parallel with the program under test during the test execution, and collect data from the PUTs and provide an interface to the processes trying to acquire coverage data. The implemented service application communicates with the PUT via network communication, collects the information during testing and sends the collected data at the proper time to the test executor.

We are able to test and measure more than one application in the same time.

7 Test execution

Test execution is made automatically by the RT-Executor. We prepare the project file showed in the beginning of this document with the test cases we would like to execute. In preparation we instrument the application we would like to test and load up it to the STB. When the test project is executed we run the RT-Executor tool which executes the test cases one by one on the instrumented application. We collect the generated information and we use it to enhance further tests.

We created a plug-in to RT-Executor tool which connects to the device, sends control messages and retrieves coverage information from it, and saves this information.

The RT-Executor tool can control the device by a remote controller simulator. The program sends control signal to set top box and controls functionalities of the device.

8 References

[1] Embedded systems functional testing methodology. CIRENE project technical report. 2012.

[2] Embedded system software and hardware analysis methodology. CIRENE project technical report. 2012.

[3] http://www.all4tec.net/index.php/All4tec/matelo-product.html