Embedded system functional testing methodology

Abstract

This report is part of the Cirene project that aims (amongst other things) the definition of a general embedded system testing methodology. In this report the overview of the methodology is given and different parts of it are described. The methodology incorporates white-box and black-box testing techniques in order to improve the quality and lower the overall cost of embedded systems testing. The core testing process of creating tests, executing tests, and using results in selecting tests to be re-executed is extended by supplementary modules like code instrumentation, coverage measurement, traceability computation, test case selection and prioritization, etc.

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

The project is co-financed by the European Union

1 Introduction

This document mainly targets the testing process of embedded systems, including black-box (functional) testing, white-box (structural) testing and their influence to each other. More precisely, black-box testing of embedded systems can be aided using white-box techniques in many ways. The base of this goal is to provide valuable information on the quality of the executed black box tests. The data acquired using white-box methods can be used not only for quality assessment but can also be used to improve the quality.

University of Novi Sad (UNS) has experience in embedded systems and their black-box testing, while University of Szeged (USZ) has experience in using white-box techniques and test optimization. Combining these knowledge, a complex testing process is defined for embedded systems which includes steps that utilizes data generated by white-box testing techniques to improve the functional testing quality and/or efficiency.

Prior to this document, a survey of embedded systems testing methods [5] was prepared in the CIRENE project. Based on that assessment of available technologies and solutions, this second document defines a general methodology for functional testing of embedded systems, and a third document will also be created to help choosing implementation parameters of the relevant parts of this methodology based on the hardware-software attributes of the target system the methodology is applied for.

2 Background

2.1 BB elements of the process

Black-box testing is performed without the knowledge about the structure of the object under test. During this process, only the interface of the object is known and used to send input information to the object, i.e. to stimulate the object. The response of the object, i.e. its output is then analyzed to see if the object responds as expected. Since the tester does not have information about internal structure of the object under test and therefore only observes how the object responds to the given set of inputs, black-box testing is said to concentrate on the question "What does the program do?" and tests whether it does works it should.

The black-box testing can be performed in two ways:

- Manual definition of test cases, in which the tests are pre-defined and executed the same way for all objects under test,
- Automatic definition of test cases, in which the testing system performs some test cases and makes decision about the remaining test cases based on the previous results.

The testing methodology consists of automated generation of test case scenario, automated test execution, which is controlled through the software application and finally test result reporting. Consequently, the proposed testing methodology enables fully automated testing and ensures thus reduction of the verification processing time and costs, compared to semi-automated or manual testing. Moreover, automated testing is more profound and ensures higher reliability due to the fact that it is performed without human intervention. The proposed automated testing methodology is carried out through defined test suites.

2.1.1 Test case generation

Test cases are generated in order to test the complete functionality of the object under test. Since the internal structure is not known, they are generated based on the specification of the functionality of the object.

Generation of test cases can be performed manually, i.e. by a test engineer. They analyze the specification of the object under test, establish a test plan which covers all aspects of the object's functionality and make the test suite based on the test plan. The test suite is executed "as is" for all objects under test. This approach is easy to implement, but lacks adaptation and provides a lot of unnecessary lag if some of the tests are not required for a particular object.

Automatic test case generation increases adaptive capabilities of the system. In this way of generating test cases, only a subset of test cases is actually executed on the object under test and based on the results of this basic set of test cases, a decision is made on which further tests need to be performed. This approach allows only the required tests to be executed and increases the overall usefulness of the test suite.

2.1.2 Functional coverage

The methodology for the black-box functional verification system consists of the following three stages:

- test suite setup,
- test suite execution,
- test result report.

Test suite setup stage defines strict instructions for the object under test that are performed during the test suite execution stage: generating the test sequences used in object's functionality inspection and setting the object in operational mode required for the inspection. Test suite execution stage produces the set of individual test case results that are further organized into the test report in the report generation stage.

Functional coverage is further increased by performing randomized test case generation, which does not execute identical tests for all objects under test, but changes them randomly. This allows better adaptation to the particular object under test.

2.2 WB elements of the process

The difference between white-box testing and black-box testing is that while black-box testing concentrates on the question "What does the program do?", and has no information about the structure of the software, white-box testing examines the "How does the program do that?" question, and tries to exhaustively examine the code from some aspects. This exhaustive examination is given by a so-called coverage criterion. There are two main types of white-box coverage criteria: instruction and branch coverage. Instruction coverage defines that program points should be executed during the tests. What a program point means is dependent on many factors like granularity (it can be sole instructions, basic blocks, methods, classes, modules, etc.). Branch coverage defines how different program paths should be executed or different decisions should be exercised during the tests. Of course, it is dependent on the definition of program point: on instruction level we can examine decisions, or even parts of the decisions (e.g. condition coverage); while on method level the call graph paths can be examined.

In this methodology we use the following coverage levels:

- ***Method level***: we can measure if all the methods were invoked during a test scenario.
- ***Instruction level***: we measure more - a method contains several instructions, but some of these may not be executed during test. In this level of coverage we measure the amount of executed instructions.

On the technical side, the coverage information somehow should be extracted from the test execution. We have many possibilities to do this.

2.2.1 Code instrumentation

In application level we can use a technique named instrumentation. As it was mentioned earlier, executable code is required as WB is a dynamic technique, thus the code gets executed during testing of the program to measure coverage. The executable code can be given in compiled, binary form or in source code. The main idea of the instrumentation is inserting instructions that output some information about the interesting points of the executed code. The information content and the interesting points are vary depending on the coverage level and criterion. For example, a simple method coverage requires only a binary "I was executed" information at the beginning of each methods, while condition coverage requires to output the value of all elementary condition of an executed decision, and the code providing this information needs to be inserted into all decisions (thus all decision points needs to be instrumented).

The instrumentation can be given in different levels. We can instrument the source code, the binary code or the middleware.

While instrumenting the source code of a program, we firs instrument the source code and compile it afterwards. This method is easy: we can find the proper lines, methods or instructions to insert the instrumentation code after or before their position. This is usually only static and manual, but there are some tools for parse the source code and insert instructions to the specified places, e.g. before every line or into the beginning of every methods. For example the GNU Compiler Collection (gcc) provides a number of instrumentation compilation options like the compilation option *-finstrument-functions* automatically inserts tracepoints at every entry and exit of user-compiled functions. Other gcc-options for instrumentation and coverage measurement are: *-ftest-coverage, -fprofile-arcs, -pg*. Automatic instrumentation is also possible using one of several source code parsing and transformation toolkits, available for various programming languages, for examples: *TXL, yacc* and *Javacc* which is used by the Java Instrumentation Engine (JIE).

Instrumentation of the binary code is a bit different, because it is hard to find the borders of the instructions or the methods if we want to trace back to source code, but it is not impossible, and in the other hand the optimization during compilation clears out the instrumentation code and the breakpoints from the code. We can measure the binary code directly for other types of coverage and information, e.g. tracing every access to memory. There are many devices to help binary instrumentations, like *Pin, DTrace, GDB, SystemTap, Frysk,* etc. Some of these are open source. There are two types of binary instrumentation: static and dynamic. Static instrumentation is prepared before executing the program, so we can plan and manage the instrumentation with tools that read executable binary code, insert instrumentation instructions and relocate instruction references. A static tracepoint insures that all its arguments remain available at runtime and won't be optimized out by the compiler. The frequently used tracepoints are most efficiently implemented with kernel tracepoints, the cost of calling a probe being that of a simple function call.

Dynamic instrumentation is adding trace points at runtime. Adding ad hoc tracepoints at runtime is not possible with kernel tracepoints, but may conveniently be achieved, albeit at the larger cost of a trap instruction. On some processors, replacing even a single byte of the instruction stream may lead

to inconsistencies between the memory and the instruction cache. The *utrace* in the kernel programming interface can address these issues and in general provide a more flexible and efficient mechanism to support debugging and tracing tools. *Ptrace* is a programming interface to interact with the process to debug and it can be used to monitor and control another process: receive notification of signals, traps and system calls; read and write the process memory and registers; and start, stop and single step the execution. To continue after hitting a breakpoint, single stepping inline (SSIL), is adequate if all threads are stopped. Otherwise, other running threads may miss the breakpoint.

2.2.2 Instrumenting the middleware

In the next level we can modify the middleware to collect information. The middleware lies between the hardware and the operating system, and it is built up from libraries and drivers. Through this layer we usually cannot acquire adequate information on the program executions.

One way to collect some information is making another layer in the system what sits over the middleware. We can find the *Universal Middleware Bridge* (UMB) [1] that can be used to solve seamless interoperability problems caused by the heterogeneity of several kinds of middleware. It stands above all devices and middleware and gives a common virtual domain. If we use the same idea and make a device or a program that gives a virtual extension of the middleware, than we can do almost anything: measure, count, make system calls, make redirections, etc.

Another approach is the generic or configurable middleware [2]: middleware that expose constructs and concepts to enable their modification and adaptation. Configurable middleware enables an application to select the actual components and specific run-time policies to address its requirements. Generic middleware extends the configurability concept. Middleware implementations have similar design, thus a specific distribution model may be built around canonical elements using a functionality-oriented approach; these elements are then adapted to conform to a specific distribution model during a personalization process. Several projects demonstrate that middleware functionalities can be described as a set of generic services, independent from any distribution model; and propose architectures based on a set of abstract interfaces. A personality is then defined as the combination of concrete modules that implement these interfaces and provide access to generic middleware services. However, modifying the middleware is always expensive in time, cost, complexity, and needs wide knowledge.

2.2.3 Modifying execution framework

In this level we can modify the execution framework or the virtual machine to send back information, log and necessary information about the code execution. We can do this by extend the code of the framework or the virtual machine, but for this, we need to have a huge experience in programming these components. The virtual machine is a software layer between the executable binary code and the operating systems. It is an environment in which special binary can be executed. Special binary is an intermediate language which is typically compiled from simple source code. The virtual machine runtime executes the special binary files, emulating the virtual machine instruction set by interpreting it. In virtual machine level we can use call trace which consists of information of called

method. Hardware emulators and simulators can also be treated as virtual machines. An advantage of them over using debugging interfaces is that while many data are generated through debugging interfaces that have to be filtered later, we can let the virtual machine to generate only the necessary information and saving communication bandwidth, memory and storage usage.

2.2.4 Debugging

In hardware level the instrumentation is not available, at least not in the way mentioned above. We need to have debug port in the hardware or a debugger device, which can communicate with the hardware in common ports. The debugger can read the code in the hardware and can insert breakpoints into it and can store additional code or contact to other devices which stores additional code. When the trap instruction is encountered, a software interrupt is generated. The additional instrumentation code may then be executed. After it, the original instruction content is restored.

In general, debuggers provide very detailed information on the program execution. This information is more than enough for tracing purposes. The task in this case is to collect runtime information from the embedded system through some debug interfaces, and then filter out irrelevant and extract the necessary information.

Debugging can provide us the following information: names of variables, functions and types; where the data is stored (for a register variable, says which register it is kept in; for a non-register local variable, prints the stack-frame offset at which the variable is always stored); the name of a symbol which is stored at a specified address; the data type of an expression; a description of data type and expression type; a brief description of all types whose names match a regular expression; the list of all the variables local to a particular scope; the names and data types of all defined functions or whose names contain a match for regular expression; the names and data types of all variables that are declared outside of functions or whose names contain a match for regular expression; even the name of the source file, the directory containing it, the directory it was compiled in, its length in lines and which programming language it is written in, the executed code, its position in the binary, source code line number it was generated from, etc.

Many embedded systems (at least those version of the commercial ones that are dedicated for development) has hardware debugging interfaces and there are many debugging interfaces exist in different embedded systems. These interfaces are designed for debugging, but as mentioned above, can be used to retrieve the necessary data on program execution. From the test server we can get the serial number of executed instructions, we can query different variables (e.g. those storing the result of a decision). We can even influence the program execution by modifying some parameters if necessary.

Debugging is another possibility to collect information or just to supervise the code during running. We can use hardware devices, debuggers, to collect information about the embedded systems by inserting breakpoints in the code or in the executing system. The hardware debuggers provides a software tool where we can see the source or the binary code and follow the run. We can insert and erase breakpoints into the code, do the running step by step, enter and leave methods, etc. The hardware debugger handles the transfer of the new information to the device under test.

Debugging without a hardware debugger is difficult, because if the device under test not supports the debugging function or don't have debug port, than we need to find some solution. We can write macros which sends signals if a problem rises. Other way is to add a bunch of logging and find some way to force a dump but this would be more time consuming and less flexible. Anyway, this type of debugging is very tough and time-consuming. [3]

We can use also the software debugger provided by the IDE, but it works only in native level. This method is same as what we've seen at the hardware debugger: the IDE provides a screen where we can supervise and interact with the code. Unfortunately this method is unusable for embedded systems.

2.2.5 Hardware aided trace generation

The trace generation is an important part of the white box testing. To calculate traceability and coverage we need to follow the run of the program. Instrumentation and debugging can provide this following by inserted feedback points.

For hardware aided trace generation we use the debug ports of the devices. The embedded systems having a debug port also have a debug system which can communicate with additional debugger devices. These debugger devices have the ability to insert feedback points into the executed program and collect the information retrieved.

Other way is to modify the executing system, maybe the executing hardware (if we have control over the hardware), to send the current instructions to a port or a pin, from where we can read and filer them and build up the trace paths.

2.3 Connection points

2.3.1 Manual feedback on covered items

We have the chance to make manual review and feedback on the code after its execution when the trace is processed and the coverage information is generated. This is the basic method of checking the result of a testing phase. This way we can easily find not covered parts of the code, which can be dead code or just shows the need of new test cases. The result and the trace of the test cases give the opportunity to ascertain the possible locations of bugs. All of these tasks require further investigations, but coverage data provide a good starting point to them.

The dead code is the part of the source code that never runs. It can be a variable, a line, a method or an entire class. Using coverage measurements it is possible to detect this type of code-element: if a test suit finishes its run and the measured coverage data shows that executions have not covered all the elements of the code, than these uncovered parts need more attention (here test suit means the test set which covers all the functions, branches, states, classes, etc.). The source code elements that cannot be covered are important for the testers and developers for redesign and redevelop the code. In this process these uncovered elements are in focus. These unreached elements may be the results of the not adequate testing or the improper development. We can analyze these elements manually. After the functionalities of these are discovered, we can decide to write new tests to reach them or

drop them out from the code. Usual deficiency is to leave a branch of an assumption untested or the by-pass of the value-interval of the variables.

The possible location of the bugs can be derived from the result of the test cases. If a test fails then supposedly there is a fault in the trace of the test. This fault can be, for example, a bad evaluation of an assumption, a null value of a variable, even the lack of an instruction, and many other things.

2.3.2 Traceability

The traceability is an important property of the tested system. Traceability is the ability to link product documentation requirements back to stakeholders' rationales and forward to corresponding design artifacts, code, and test cases. Verify the history, location, implies the use of a unique piece of data which can be traced through the entire software flow of the program.

Traceability can be computed based on the connection between the functionalities, the test cases and the coverage information. We make the test cases from the specification, separately for each functionality. Thus we know which test case set refers to which functionality. During the execution we measure the coverage of the test cases, so we know how many and what lines, methods, etc. are reached by each test case. With this chain we can compute traceability links between test cases and source code, and through these and the existing traceability links, we can define missing links between other levels (e.g. requirements and source code). Or, if traceability exists between code and higher level elements, we can compute different coverage based on code coverage (e.g. compute requirements coverage if requirements-code traceability links are available).

2.3.3 Random testing

Random testing can quickly generate many tests, is easy to implement, scales to large software applications, and reveals software errors. But it tends to generate many tests that are illegal or that exercise the same parts of the code as other tests, thus limiting its effectiveness. Directed random testing is an approach to test generation that overcomes these limitations, by combining a bottom-up generation of tests with runtime guidance. A directed random test generator takes a collection of operations under test and generates new tests incrementally, by randomly selecting operations to apply and finding arguments from among previously-constructed tests. As soon as it generates a new test, the generator executes it, and the result determines whether the test is redundant, illegal, error-revealing, or useful for generating more tests. The technique outputs failing tests pointing to potential errors that should be corrected and passing tests that can be used for regression testing.

Another test generation technique that follows this is branch-directed test generation technique. Branch-directed generation is a test generation technique whose goal is to generate test inputs that cover branches left uncovered by a previous input generation technique. A branch-directed generator takes an input set (e.g. one produced by a previous run of directed random testing) and outputs a new input set containing inputs that cover branches left uncovered by the old set. Its generation strategy is to select inputs from the old set that reach but do not follow a particular branch in the code, and to make small, targeted modifications to the inputs, guided by a dynamic data flow analysis of values that impact the branch condition. [4]

Traceability in testing could be used for directing random testing. If we have a feedback from executed test cases, i.e. we can compute coverage based on code coverage, than we can assume that uncovered paths and branches could be reached by generating different inputs then those that led the test cases to currently covered paths. In other words, when we stick to a single state in DUT model, all covered paths from it have some inputs that causes transitions once generated. So, the union of all these inputs leads to already covered path. All the inputs that are not in this union (the reminder) could lead to uncovered path or branch, or possibly be redundant. As these inputs (new test cases) are added incrementally, we can easily find out if the test is redundant or illegal (not covering new path) and then exclude it from test scenario, and proceed with adding next input (next test case).

2.3.4 Prioritization/selection

Test case prioritization and selection 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 the same goal (e.g. to cover the functions, the code, the methods, the branches or reach a given level of any other type of coverage). In other hand, if we know which test cases have the responsibility for which part of the program, than after change the program we can test only the changed part. In the same way, if we have the connections between the different parts of the program, we can set up the impact sets of the different parts and after change we can select the proper test cases for testing only the changed and the impacted parts.

To prioritize / select test cases we might measure some enumerable property of them. Selecting a set of test cases is the key to lessen the needs of testing, save time and money. Prioritizing test cases means the optimal usage or save of the available resources. In the previous section we mentioned some sort of prioritization and selection which can be used to make an order in the test cases and help us selecting the necessary ones. Here comes some example, but there are many others. The measurable values, the available devices and the budget make the only limit.

- The first example property is the speed of the test case: how much time does it take a test from launch to return. With minimal modification even the test case itself can calculate this value for us, but we can prepare the environment to measure it. We can then select test cases that fit into a time frame, or execute the shortest ones first.

- The second property is derived from the result of the run of a test. We can count the fault rate of the test cases and we can prioritize by this attribute. The decision can be to exercise the most times failed test cases first, and leave those that always passed to the end of testing. We can add some locality: count fails from the last given period only (e.g. half year, or since last release).

- Complex tests can also be preferred utilizing the assumption that a more complex test exercises a larger part of the system under test. The complexity of the test might be estimated by the number and size of its input.

- With the complexity of a test the needs of resources can be grow, too. If we can measure the needs of resources than we can make an ascending or a descending list by it and we can decide if we want to use the less resource or try to reach the limits in resources.

- In this paper we mentioned the coverage criterion earlier. The coverage criterion is the best way to prioritize the test cases. Coverage gives many type of valuable information about the tests and the different coverage types are clearly connects to different properties. For example coverage can give the reached instructions, basic blocks, methods, classes, modules, etc. It can also define different program paths that executed or different decisions that exercised during the tests. Using this information to make an order of the test cases is the best choice. The easiest way to select or prioritize by the coverage information is the greedy method.

- We can utilize coverage in some other ways. If source code changes information are present, test selection can be done in a way that only those test cases are selected that cover the changed code parts. We can combine this with some prioritization: choose specific test cases first that cover the least code besides the changed parts.

- To start with the greedy way, we can select the first X piece from the list. These have the largest (or lowest) coverage or the largest (or lowest) values of the measured property. We can assume that the selected set will cover the largest part of the code, the functionalities, or methods, based on the type of the prioritization.

- If we study the traces of the test cases, than we can make selection to reach all different traces. If we make an order by the length of the traces, than we can chose the longest from all traces. Or we can choose the smallest set that covers all the traces. With this we can suppose a good coverage on the code.

- But we can select not only from a prioritized list of the test cases. Before make the ordering we can select test cases for other reasons, e.g. to cover a selected function, classes, objects or functionalities. But for this we also might use some information about them. In the first case we can discover the connections between the test cases and the source code elements, the functions and the traces.

If properties are not available for some reasons, than the usual way to select test cases is the random selection. With this we can choose without any further knowledge and usually we can reach a good coverage on the code or the functions.

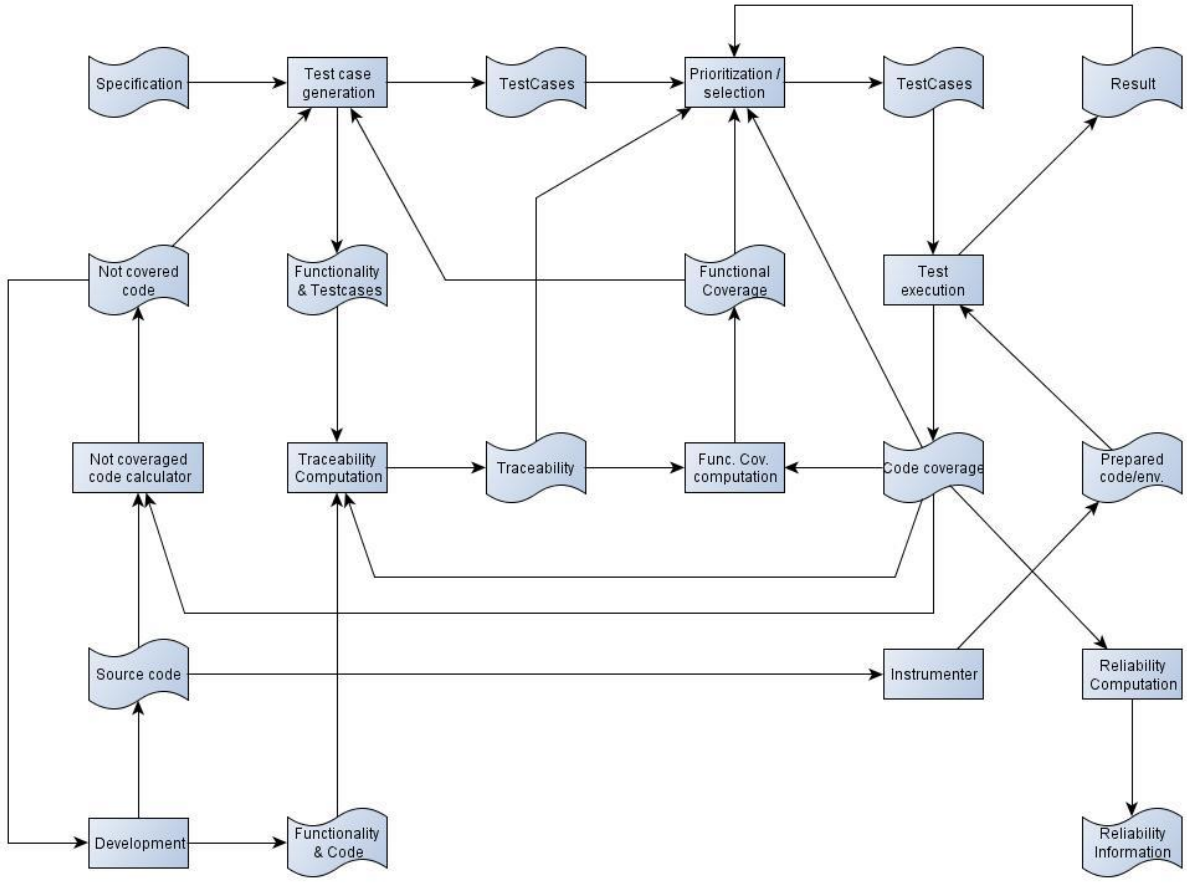
Another common reason of the test case selection is to select the tests that failed earlier to recheck the repaired functions, classes, methods or other source code elements.

2.3.5 Reliability model

The reliability provides an estimation of the level of business risk and the likelihood of potential application failures and defects that the application will experience when placed in operation. The

reliability model helps us to calculate these values using low level, directly measurable information. Such information can be the distribution of bugs in the code modules which can be obtained using test execution results, code coverage and traceability information. But other measures, e.g. static code metrics (such as code complexity) can also be used in a reliability model.

3 Testing process (and technical details of the components)



The figure above shows different parts of our white box aided black box testing process.

As a starting point test cases are made from the specification. The test cases are ordered and executed. On the other hand, the source code is instrumented or the executing environment is modified in order to prepare it to information collection. Tests are executed in this modified environment, which results in not only the “normal” pass-fail data but also in code coverage. Pass/fail results and code coverage are usually reused in subsequent test execution tasks by selecting and/or prioritizing test cases.

But code coverage can be utilized in other ways too. If the source code and covered code are compared, one can easily determine the uncovered code parts. The cause of the fact that some code parts are not covered requires manual investigation, which has two possible outcomes. The first is that the code fragment can be reached within the software and there is a lack of proper test cases that exercise this code fragment. In this situation new test cases can be generated exploiting the (not covered fragments of the) source code. In the second case, the not covered code parts are unreachable, thus they can be removed by the developers.

Code coverage can also be used to compute higher level coverage. If the development provides us traceability links between functionality and source code, and the test case generation step also provides functionality and test case links, then (based on these traceability information) functional coverage computation becomes available. This functional coverage then can be used for new test cases generation (in order to cover more functionality) or in test case selection or prioritization.

Code coverage can also be used for traceability links computation. If the development provides us traceability links between functionality and source code but traceability links between functionality and test cases are missing, or vice versa, code coverage can be used to determine the missing links. E.g. if traceability links between functionalities and test cases exist, and we also have the coverage information that what particular source code fragments are exercised by a given test case, we can link functionality to source code by linking each functionality to all code parts that are exercised during a test case execution of a test case linked to the given functionality. (In real situations, this coupling must be a little bit harder and may require some heuristics.)

Code coverage can also be used to derive the reliability of the tests and the code itself.

3.1 Explanation of the different boxes on the figure

Test case generation: refers to a method that produces test cases out of specification automatically. It has *the not covered code* metric and the *functional coverage* metric as additional inputs to generate new test cases if some are needed. We can extract from this method an associative map that shows the connections between the functionalities and the test cases.

Prioritization/Selection: It is a method that helps us to lessen the required time and the number of test cases for an exhaustive testing. This process gets the inputs from former results to select the failed tests for control testing; traceability, code coverage and functional coverage to select test cases to reach any kind of coverage criterion. This process gives a set of test cases as output.

Test execution: a process that runs the prepared environment or the instrumented code in the system, executes the test cases and collects coverage information from it automatically.

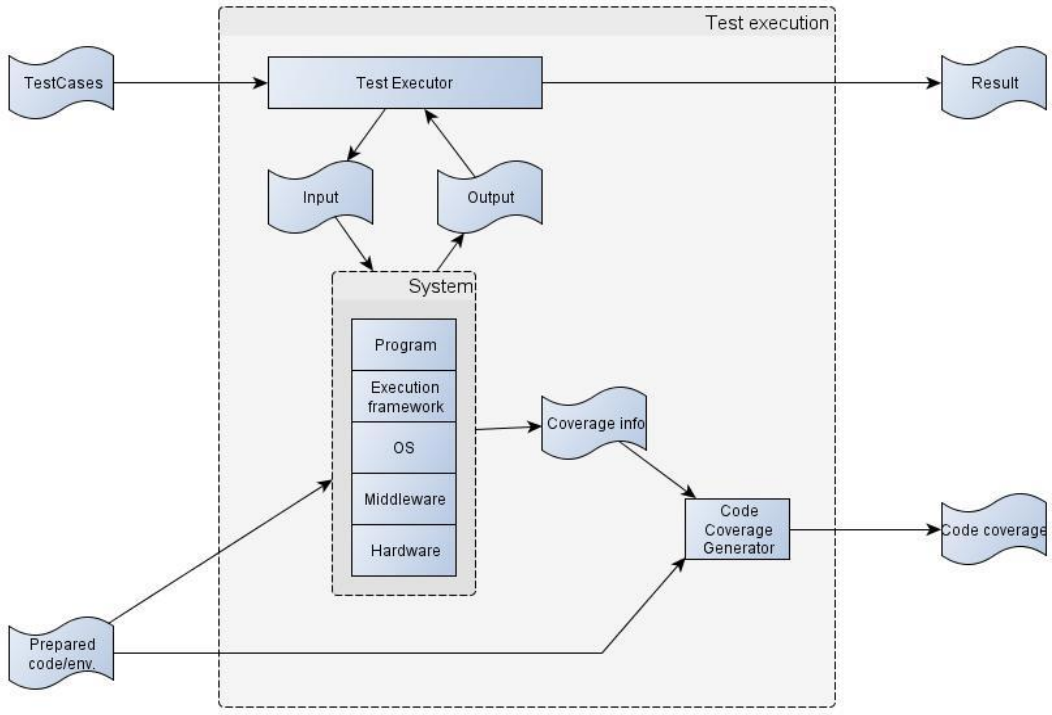
Not covered code calculator: it gets the source code and the code coverage information and ascertains if there are any part of the source code, that not have covered by test cases.

Traceability computation: a process which calculates the source code elements that was reached by a test case during its execution. It can derive the expected trace from the two maps, the 'Functionality & Test cases' and the 'Functionality & Code' maps, and combined this information with coverage it can specify the really reached trace. Than we can calculate the difference between the two traces to get information for new test case generation.

Functional coverage computation: it can ascertain the covered functionalities of a test case from the test case's traceability and coverage information. It helps us to select the proper test cases for testing a chosen functionality or gives information about not covered functions to generate new test cases.

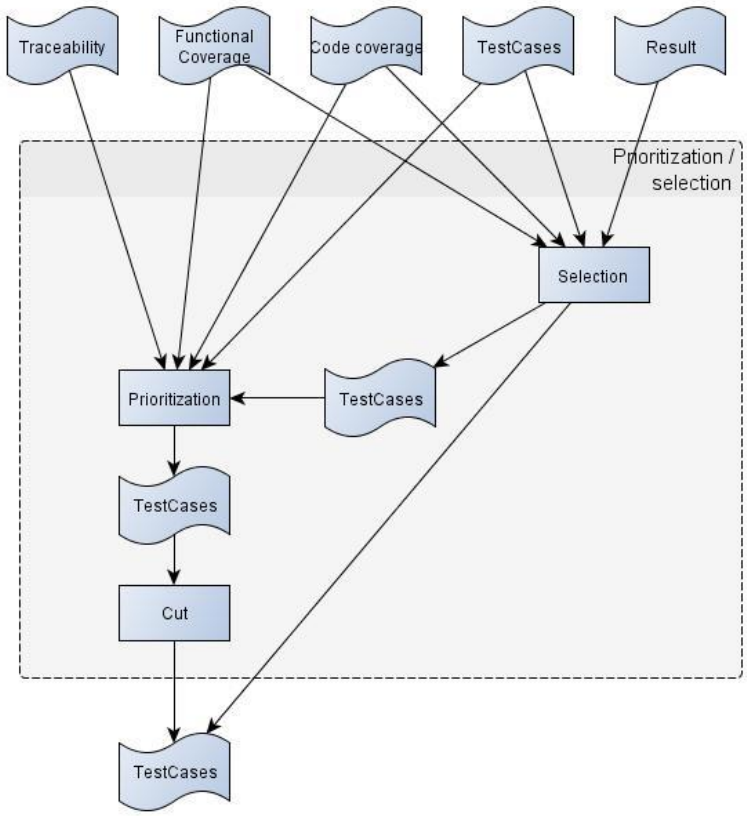
Instrumenter: a device that instruments the source code, the binary code or modify the executing framework. It prepares the code or the environment to send information during test execution by inserting feedback instructions into the source code or inserting interrupt instructions and additional code to the binary. The execution framework can also be modified by this technique to send back information during the execution.

Reliability computation: a process that computes reliability of the software under test using low level easily measurable information, e.g. code coverage.



This figure shows the inner structure of the test execution part of the previous figure.

The test case executor takes the test cases to be executed as input. Then it executes the program on the system with the proper input parameters. (Note that this execution can be either manual or automatic.) The system includes the prepared environment (instrumented program code, modified execution framework, attached hardware debugger, etc.). There are two outcomes of program execution. The “normal” output of the program is passed back to and processed by the test executor that decides on the result (pass/fail). The “extra” output of the prepared system is the traces produced during program execution. These traces are processed according to the prepared environment and code coverage data is generated.



This figure shows the test prioritization/selection method.

For a good prioritization and selection, we might use the following information:

- The traceability of the test cases shows the connection between the code and the test case for us.
- The functional coverage tells the connection between the specification and the test cases: shows that which test case refer to which functionality; the functionalities are defined in the specification.
- Code coverage gives the information of how much code does the test case reached during execution.
- The results show if a test case failed or passed.

The main input and output also a set of test cases, but a different set. The input contains all the generated test cases and the output contains the selected ones.

There are two ways of the process:

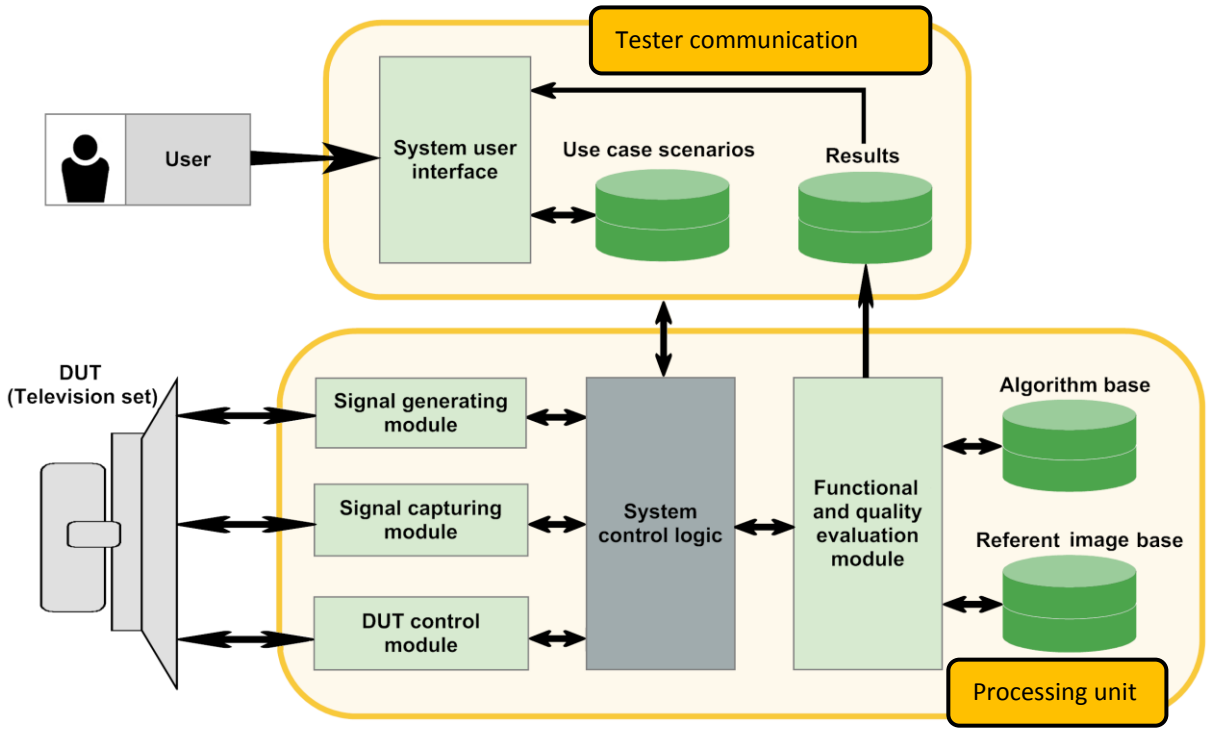
First, we can select from the full list of test cases using some binary properties of the test cases. For example, test cases cover some functions (i.e. executing the test case calls the function at least once), and in a new version of the software some functions must be changed since the previous version. Now we can select only those test cases that cover the changed functions, as it is obvious

that these are the test cases that potentially detect the errors introduced in the changed functions. Furthermore, we can add those test cases to the selection that were failed during the last execution to perform re-testing (confirmation testing). Or we can add test cases that cover not yet covered code to the selection.

Second, we can prioritize the set of test cases (either the full set or a pre-selected set of test cases) based on some measured property of the test cases and select from the ordered set to reach a good outcome of some previously defined criterion. E.g. we can measure the execution time of the test case or the number of functions covered by the test. The criterion can be the coverage of all functions or the coverage of some or all instructions, methods, classes, to perform as many test cases within a time frame as many possible, etc.

3.2 BB testing parts

The proposed system for black-box functional verification includes a complete testing framework.



This figure shows the architecture of BB testing environment.

The system for automated functional failure inspection of device under test (DUT) consists of two main blocks:

- a processing unit (PU), and
- a tester communication unit (TCU).

The TCU primarily interacts with a tester (a person who defines and executes tests). It consists of a system user interface (SUI), intended for specifying the use-case scenario of a DUT, starting the

verification process and examining the final verification results. The TCU also includes a use-case scenario database (UCSD), as a collection of versatile test cases.

The PU performs the overall functional failure detection and controls the DUT. Particularly, the PU consists of a generator unit (GU) with two basic functionalities:

- to generate a test input that will be processed within the DUT to produce the output, based on which the functionality of the DUT will be estimated (Signal generating module - SGM), and
- to acquire from the DUT an outgoing value processed by the DUT (Signal capturing module - SCM).

In addition, the PU consists of a DUT control module (DCM) that manages the entire control over the DUT. The communication between the DCM and the DUT (i.e. it's TV Processing Unit - TPU) must be implemented. PU also comprises a functionality evaluation unit (FEU), which performs entire functional failure analysis. FEU analyses the output acquired from the DUT in order to determine the impairment induced by DUT. The analysis is performed employing the algorithm from the algorithm database (AD). In the algorithm, the output (acquired from the DUT) is compared to the reference result obtained from the reference image database (RID), which is considered to represent perfect device functionality. As an output, the algorithm provides the quantitative measurement of similarity between the two results. In conjunction with an adaptive threshold, this value is used in determining the acceptability of the device functionality. After determining the similarity level, the PU stores the similarity estimation result into the result database (RD), which is further used in the DUT failure analysis.

Finally, the PU comprises a system control logic unit (SCLU), which constitutes the substantial part of the PU. It manages the whole process of failure detection: At startup, SCLU gets from the TCU (i.e. the UCSD) the test case sequences upon which it initiates SGM to generate a corresponding test input for DUT. Through DCM, SCLU directs the DUT operations, and at DUT output, through the SCM, captures the output. It transfers the acquired result to FEU for similarity analysis. Also, SCLU provides the TCU (i.e. the SUI) with similarity estimation results that are expressed both quantitatively (percentage of defect) and binary (acceptable/not acceptable functionality).

3.3 WB testing parts

The white-box parts have their proper places in the proposed testing process, but technically they are not so easy to implement properly. First, the program compilation and/or execution toolchain must be extended with instrumenting capability. This means that an instrumenter must be inserted in this toolchain. It can be a source code instrumenter at the beginning of the compilation toolchain, a modified compiler that generates instrumented binaries, a binary instrumenter, or a modified execution environment that generates proper instructions into the code just before execution. This instrumenter prepares the program to produce traces during its executions. Traces may contain various information on the execution, like method entries/exits, actual decision outcomes, source code line number of executed instructions, etc. Usually, coverage granularity has influence on the

information generated into the trace. The trace must be processed to gather code coverage information

Code coverage then can be reported as a percentile data showing the quality of the tests, or can then be used in many ways in other parts of the proposed testing process.

3.4 Connections between BB and WB

In this section we describe how the black-box and white-box techniques can be combined, i.e. how can code coverage help black-box testing.

To gather proper coverage information, the test execution toolchain must use prepared test execution environment.

The output of the coverage measurement is a list of covered items for each test case. This list can be used for test selection and prioritization. The step is performed before test execution, and based on the "coverage matrix" (and other data such as changed code elements) it first determines a set of relevant test cases, then prioritizes these test cases. After prioritization, the list can be cut again at some positions based on some restriction of test resources. The modules performing these steps are separate modules that manipulate the original set of test cases, but they must be transparent to the other parts of the black-box execution toolchain.

Coverage data can also be used to produce missing traceability links between functionality and program code. Based on the links between functionality and test cases and coverage-provided relations between test cases and source code elements, some algorithms must be implemented that can choose relevant links. Choosing relevant links is important, because when a test case is executed, many common and general code parts are also exercised beside those that implement the given functionalities. This is a separate module that not directly influences the black-box test toolchain.

If functionality-to-code traceability links are determined from other sources, code coverage can be used to calculate functional coverage. This is a separate module helping the evaluation of the completeness of the tests.

List of covered items can also be directly used by the testers and developers. Testers might define new test cases to execute not covered code parts. This task might be at least partially automated using different test generation techniques. Developers can use coverage information to detect unreachable code parts in the software. This is a manual work, and will affect black-box testing only indirectly.

Code coverage data can also be used for reliability computations. The reliability model is a separate module in the proposed test process, which has many inputs including code coverage, traceability, test case data, etc. Thus, it computes the reliability of the tests and the system using data from white-box and black-box testing.

4 Summary

In this document a process for embedded systems testing is defined. The process includes both black-box and white-box testing techniques and their possible combinations.

The main flow of the process is a usual black-box testing process: test cases are generated from the specification, some of them are selected to be executed in a certain order, then execution produces pass/fail results that can be used to select/prioritize test cases in the next execution. The primary attachment point of the white-box techniques is test execution, where a modified execution environment produces code coverage information. Coverage data can be directly or indirectly used to improve the quality of testing. It can be used to select more appropriate test cases, or to order selected test cases according to some coverage criteria. Indirect usages include the contribution to traceability link computations, unreachable code detection and can be used to drive the definition of new test cases.

The project is co-financed by the European Union

5 References

[1] Kyeong-Deok Moon, Young-Hee Lee, and Chang-Eun Lee, Young-Sung Son: *Design of a Universal Middleware Bridge for Device Interoperability in Heterogeneous Home Network Middleware* In IEEE Transactions on Consumer Electronics, Vol. 51, No. 1, FEBRUARY 2005.

[2] Jérôme Hugues, Laurent Pautet, Fabrice Kordon: *Contributions to middleware architectures to prototype distribution infrastructures* In IEEE Rapid Systems Prototyping, JUNE 2003.

[3] http://blogs.msdn.com/b/ntdebugging/archive/2009/05/21/debugging-without-a-debugger.aspx; last visited in 2012. 08. 08.

[4] Carlos Pacheco: Design Directed Random Testing, Massachusetts Institute of Technology, June 2009.

[5] Árpád Beszédes, Tamás Gergely, Kornél Muhi, Róbert Rácz, Csaba Nagy, István Siket, Gergő Balogh, Péter Varga, Miroslav Popovic, István Papp, Jelena Kovacevic, Vladimir Marinkovic. Survey on Testing Embedded Systems. Technical report. 2012.