Acta Cybernetica **21** (2014) 439–458.

Code Coverage Measurement Framework for Android Devices[*]

Ferenc Horváth[†], Szabolcs Bognár[†], Tamás Gergely[†], Róbert Rácz[†], Árpád Beszédes[†], and Vladimir Marinkovic[‡]

Abstract

Software testing is a very important activity in the software development life cycle. Numerous general black- and white-box techniques exist to achieve different goals and there are a lot of practices for different kinds of software. The testing of embedded systems, however, raises some very special constraints and requirements in software testing. Special solutions exist in this field, but there is no general testing methodology for embedded systems. One of the goals of the CIRENE project was to fill this gap and define a general testing methodology for embedded systems that could be specialized to different environments. The project included a pilot implementation of this methodology in a specific environment: an Android-based Digital TV receiver (Set-Top-Box).

In this pilot, we implemented method level code coverage measurement of Android applications. This was done by instrumenting the applications and creating a framework for the Android device that collected basic information from the instrumented applications and communicated it through the network towards a server where the data was finally processed. The resulting code coverage information was used for many purposes according to the methodology: test case selection and prioritization, traceability computation, dead code detection, etc.

The resulting methodology and toolset were reused in another project where we investigated whether the coverage information can be used to determine locations to be instrumented in order to collect relevant information about software usability.

In this paper, we introduce the pilot implementation and, as a proof-of-concept, present how the coverage results were used for different purposes.

Keywords: coverage, embedded, traceability, Android

[*]Part of this work was done in the Cross-border ICT Research Network (CIRENE) project (project number is `HUSRB1002/214/044`) supported by the **Hungary-Serbia IPA Cross-border Co-operation Programme**, co-financed by the European Union. This research was partially supported by the Hungarian national grant **GOP-1.1.1-11-2011-0006**.

[†]University of Szeged, Department of Software Engineering, E-mail: `{hferenc,bszabi,gertom,rrobi,beszedes}@inf.u-szeged.hu`

[‡]University of Novi Sad, Faculty of Technical Sciences, E-mail: `vladam@uns.ac.rs`

DOI: 10.14232/actacyb.21.3.2014.10

1 Introduction

Software testing is a very important quality assurance activity of the software development life cycle. With testing, the risk of a residing bug in the software can be reduced, and by reacting to the revealed defects, the quality of the software can be improved. Testing can be performed in various ways. For example, static testing includes the manual checking of documents and the automatic analysis of the source code without executing the software. During dynamic testing the software or a specific part of the software is executed. Many dynamic test design techniques exist, the two most well known groups among them are black-box and white-box techniques.

Black-box test design techniques concentrate on testing functionalities and requirements by systematically checking whether the software works as intended and produces the expected output for a specific input. The techniques take the software as a black box, examine "what" the program does without having any knowledge on the structure of the program, and they are not intrerested in the question "how?".

On the other hand, white-box testing examines the question "How does the program do that?", and tries to exhaustively examine the code from several aspects. This exhaustive examination is given by a so-called coverage criterion which defines the conditions to be fulfilled by the set of statement sequences executed during the tests. For example, 100% instruction coverage criterion is fulfilled if all instructions of the program are executed during the tests. Coverage measures give a feedback on the quality of the tests themselves.

The reliability of the test can be improved by combining black-box and white-box techniques. During the execution of test cases generated from the specifications using black-box techniques, white-box techniques can be used to measure how completely the actual implementation is checked. If necessary, reliability of the tests can be improved by generating new test cases for the code fragments not verified.

1.1 Specific problems with embedded system testing

Testing in embedded environments has special attributes and characteristics. Embedded systems are neither uniform nor general-purpose. Each embedded system has its own hardware and software configuration typically designed and optimized for a specific task, which affects the development activities on the specific system. Development, debugging, and testing are more difficult since different tools are required for different platforms. However, high product quality and testing that ensures this high quality is very important. Suppose that the software of a digital TV with play-from-USB capabilities fails to recover after opening a specific media file format and this bug can only be repaired by replacing the ROM of the TV. Once the TV sets are produced and sold, it might be impossible to correct this bug without spending a huge amount of money on logistic issues. Although there are some solutions aiming at the uniformisation of the software layers of embedded systems (e.g., the Android platform [12]), there has not been a uniform methodology for embedded systems testing.

1.2 The CIRENE project

One of the goals of the CIRENE project [19] was to define a general testing methodology for embedded systems that copes with the above mentioned specialities and whose parts can be implemented on specific systems. The methodology combines black-box tests responsible for the quality assessment of the system under test and white-box tests responsible for the quality assessment of the tests themselves. Using this methodology the reliability of the test results and the quality of the embedded system can be improved. As a proof-of-concept, the CIRENE project included a pilot implementation of the methodology for a specific, Android-based digital Set-Top-Box system. Although the proposed solution was developed for a specific embedded environment, it can be used for all Android-based embedded devices such as smart phones or tablets.

The coverage measurement toolchain plays an important role in the methodology (see Figure 1). Many coverage measurement tools (e.g., EMMA [28]) exist that are not specific but can be used on Android applications. However, these are applicable only during the early development phases as they are able to measure code coverage on the development platform side. This kind of testing omits to test the real environment and misses the hardware-software co-existence issues which can be essential in embedded systems. We are not aware of any common toolchain that measures code coverage directly on Android devices.

Our coverage measurement toolchain starts with the instrumentation of the application under test, which allows us to the measure code coverage of the given application during test execution. As the device of the pilot project runs the Java-based Android operation system, Java instrumentation techniques can be used. Then, the test cases are executed and the coverage information is collected. In the pilot implementation, the collection is split between the Android device and the used testing tool RT-Executor [24]: the service collects the information from the individual applications of the device, while the testing tool processes the information (through its plug-ins).

The coverage information gathered with the help of the coverage framework can be utilized by many applications in the testing methodology. They can be used for selecting and prioritizing test cases for further test executions, or for helping to generate additional test cases if the coverage is not sufficient. It is also useful for dead code detection or traceability links computation.

The resulting methodology and toolset were reused in another project which aims usability testing on Android devices. In this project, we investigated whether the coverage information gathered by the described method can be used to determine locations in the code that must be “watched” during test executions in order to collect relevant information of the usability of the software. The long-term goal was to reduce the number of instrumentation points in the examined software which results in less performance decrease and, thus, supports aimed mass field testing.

In this paper, we introduce the pilot implementation, discuss our experiments conducted to examine the further use of the coverage results, and evaluate these experiments.

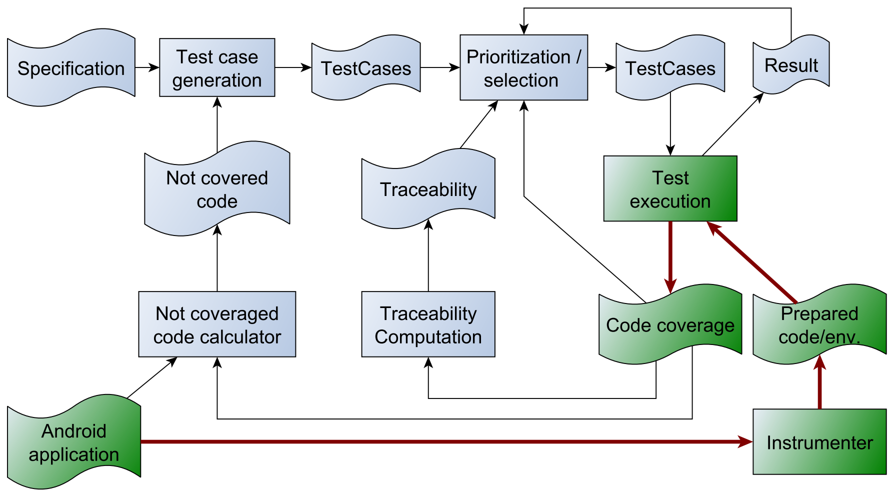


Figure 1: Coverage collection methodology on the Set-Top-Box

1.3 Paper structure

The rest of the paper is organized as follows. In Section 2 we give an overview on the related work. Section 3 presents the implementation of the coverage measurement framework. In Section 4 some use cases are shown to demonstrate the usefulness of coverage information. Finally, we summarize our achievements and introduce some possible future works in the last section.

2 Related Work

In the CIRENE project, one of our first tasks was to assess the state-of-the-art in embedded systems testing techniques with special attention to the combined use of black and white-box techniques. As a result of this task we presented a technical report [3] in which we report only a few number of combined testing techniques that have been specialized and implemented in the embedded environment.

Gotlieb and Petit [17] presented a path-based test case generation method. They used symbolic program execution and did not execute the software on the embedded device prior to the test case definitions. We use code coverage measurement of real executions to determine information that can be used in test case generation.

José et *al.* [9] defined a new coverage metric for embedded systems to indicate instructions that had no effect on the output of the program. Their implementation used source code instrumentation and worked for C programs at instruction level, and had a great influence on the performance of the program. Biswas et *al.* [4] also utilized C code instrumentation in embedded environment to gather profiling information for model-based test case prioritization. We use binary code instrumentation at method level, use traditional metric that indicates whether the method is executed during the test case or not, and our solution has a minimal overhead on execution time. The resulting coverage information can also be used for test case selection and prioritization.

Hazelwood and Klauser [18] worked on binary code instrumentation for ARM-based embedded systems. They reported the design, implementation and applications of the ARM port of the Pin, a dynamic binary rewriting framework. However, we are working with Android systems that hides the concrete hardware architecture but provides a Java-based one.

There are many solutions for Java code coverage measurement. For example, EMMA [28] provides a complete solution for tracing and reporting code coverage of Java applications. However, it is not concerning the specialities of Android or any embedded systems.

Most of the coverage measurement tools utilize code instrumentation. In Java-based systems, byte code instrumentation is more popular than source code instrumentation. There are many frameworks providing instrumenting functionalities (e.g., DiSL [21], InsECT [6, 26], jCello [27], and BCEL [2]) for Java. These are very similar to each other regarding their provided functionalities. We chose Javassist [7] to be our instrumentation framework in the pilot project.

Traceability links between requirements and source code are important in software development. Automatic methods for traceability link detection include information retrieval ([20, 1, 8]) and probabilistic feature location ([22]) and combined techniques ([11]). We used code coverage based feature location to retrieve traceability information.

3 Coverage Measurement Toolchain

The implemented coverage measurement toolchain consists of several parts. First, the applications selected for measurement have to be prepared. This process includes program instrumentation that inserts extra code into the application so that the application can produce the information necessary for tracing its execution path during the test executions. The modified applications and the environment that helps collecting the results must be installed on the device under test.

Next, tests are executed using this measurement environment and the prepared applications, and coverage information is produced. In general, test execution can be either manual or automated. In the current implementations, we use two different approaches for test automation.

Within the CIRENE pilot implementation *RT-Executor* [24] (a black-box test automation tool for multimedia devices testing) is used as the automation tool.

In the usability testing project we use a simplified tool in the testing process, which helps gathering and preparing the coverage information for the evaluation. The functions of this tool are based on the *Robotium* [16] framework. Robotium is an Android test automation framework that has full support for native and hybrid applications and makes it easy to write powerful and robust automatic black-box tests for Android applications.

During the execution of the test cases, the instrumented applications produce their traces which are collected, and coverage information is sent back to the automation tool.

Third, the coverage information resulted from the previous test executions is processed and used for different purposes, e.g., for test selection and prioritization, additional test case generation, traceability computation, and dead code detection.

In the rest of this section, we describe the technical details of the coverage measurement toolchain.

3.1 Preparation

In order to measure code coverage, we have to prepare the environment and/or the programs under test to produce the necessary information on the executed items of the program. In our case, the Android system uses the Dalvik virtual machine to execute the applications. Although modifying this virtual machine to produce the necessary information would result in a more extensive solution that would not require the individual preparation of the measured applications, we decided not to do so, as we assumed that modifying the VM itself had higher risks than modifying the individual applications. With individual preparation it is much easier to decide what to measure and at what level of details. So, we decided to individually prepare the applications to be measured. As we were interested in method level granularity, the methods of the applications were instrumented before test execution, and this instrumented version of the application was installed on the device. In addition, a service application serving as a communication interface between the tested applications and the network was also necessary to be present on the device.

3.1.1 Instrumentation

During the instrumentation process, extra instructions are inserted in the code of the application. These extra instructions provide additional functionality (e.g., logging necessary information) but they should not modify the original behaviour of the application. Instrumentation can be done on the source code or on the binary code.

In our pilot implementation, we are interested in method level code coverage measurement. It requires the instrumentation of each method inserting a code that logs the fact that the method is called. As our targets are Android applications usually available in binary form, we have chosen binary instrumentation.

Android is a Java-based system which in our case means that the applications are written in Java language and compiled to Java Bytecode before a further step creates the final Dalvik binary form of the Android application. The transformation from Java to Dalvik is reversible, so we can use Java tools to manipulate the

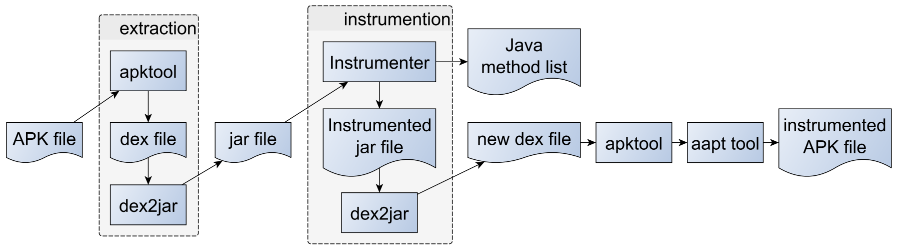


Figure 2: Instrumentation toolchain

program and instrument the necessary instructions. We used the `Javassist` [7] library for Java bytecode instrumentation, `apktool` [13] for unpacking and repacking the Android applications, the `dex2jar` [14] tool for converting between the Dalvik and the Java program representations, and `aapt` [15] tool for sign the application. The *Instrumentation toolchain* (see Figure 2) is the following:

- The Android binary form of the program needs to be instrumented. It is an `.apk` file (a special Java package, similar to the `.jar` files, but extended with other data to become executable).

- Using the `apktool` the `.apk` file is unpacked and `.dex` file is extracted. This `.dex` file is the main source package of the application, it contains its code in a special binary format. [15, 5]

- For all `.dex` files the `dex2jar` is used to convert them to `.jar` format.

- On the `.jar` files we can use the `JInstrumenter`. The `JInstrumenter` is our Java instrumentation tool based on the `Javassist` library [7].

  `JInstrumenter` first adds a new collector class with two responsibilities to the application. On the one hand, it contains a coverage array that holds the numbers indicating how many times the methods (or any other items that is to be measured) were executed. On the other hand, this class is responsible for the communication with the service layer of the measurement framework. Next, the `JInstrumenter` assigns a unique number as ID to each of the methods. This number indicates the method's place in the coverage array of the collector class. Then a single instruction is inserted in the beginning of all methods which updates the corresponding element of the coverage array on all executions of the method.

  The result of the instrumentation is a new `.jar` file with instrumented methods and another file with all the methods' names and IDs.

- The instrumented `.jar` files are converted to `.dex` files using the `dex2jar` tool again.

- Finally, the `.apk` file instrumented application is created by repacking the `.dex` files with the `apktool` and signing it with the `aapt` tool.

During the instrumentation, we give a name to each application. This name will uniquely identify the application in the measurement toolchain, so the service application can identify and separate the coverage information of different applications.

After the instrumentation, the application is ready for installation on the target device.

3.1.2 Service application

In our coverage measurement framework implementation it is necessary to have an application that is continuously running on the Android device in parallel with the program under test. During the test execution, this application is serving as a communication interface between the tested applications and the external tool collecting and processing the coverage data. On the one hand this is necessary because of the rights management of the Android systems. Using the network requires special rights from the application and it is much simplier and more controllable to give these rights to only a single application than to all of the tested applications. On the other hand, this solution provides a single interface to query the coverage data even if there are more applications tested and measured simultaneously.

In Android systems, there are two types of applications: "normal" and "service". Normal applications are active only when they are visible. They are destroyed when moved in the background, although their state can be preserved and restored on the next activation. Services are running in the background continuously and are not destroyed on closing. So, we had to implement this interface application as a service. It serves as a bridge between the Android applications under test and the "external world" as it can be seen on Figure 3. The tested applications are measuring their own coverage and the service queries these data on-demand. As the communication is usually initiated before the start and after the end of the test cases, this means no regular communication overhead in the system during the test case executions.

Messages are accepted from and sent to the external coverage measurement tools. The communication uses JSON [10] objects (type-value pairs) over the TCP/IP protocol. Implemented messages are:

NEWTC The testing tool sends this message to the service to sign that there is a new test case to be executed and asks it to perform the required actions.

ASK The testing tool sends this message to query the actual coverage information.

COVERAGE DATA The service sends this message to the testing tool in response to the **ASK** message. The message contains coverage information.

Internally, the service also uses JSON objects to communicate with the instrumented applications. Implemented signals are:

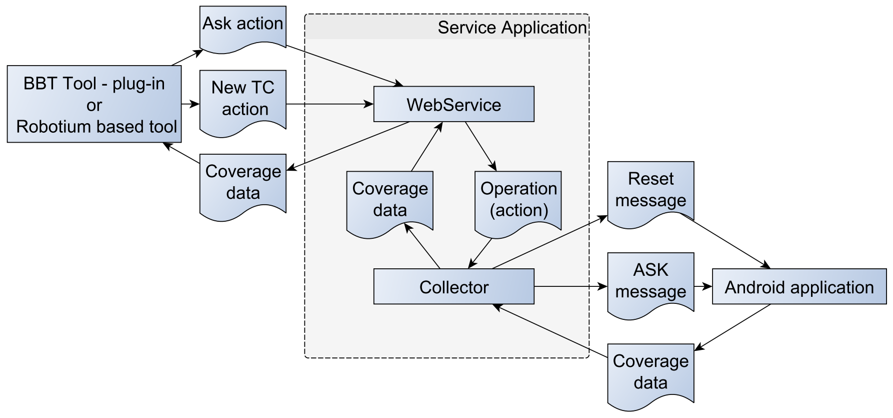


Figure 3: Service Layer

reset With this signal the service asks the apps to reset the stored coverage values.

ask The service sends this signal to query the actual coverage information.

coverage data The application sends this message to the service in response to the **ask** signal. The message contains coverage information.

3.1.3 Installation

To measure coverage on the Android system, two things need to be installed: the particular application we want to test and the common service application that collects coverage information from any instrumented application and provides a communication interface for querying the data from the device.

The service application needs to be installed on a device only once; this single entity can handle the communication of all tested applications.

The instrumented version of each application that is going to be measured must be installed on the Android device. The original version of such an application (if any) must be removed before the instrumented version can be installed. It is necessary because Android idetifies the applications by their special android-name and package, and our instrtumentation process does not change these attributes of the applications; it only inserts the appropriate instructions into the code. Our toolchain uses the `adb` tool (can be found in Android Development Kit) to remove and install packages.

3.2 Execution

During test execution, the Android device executes the program under test and the service application simultaneously. The program under test counts its own coverage information and sends this information when the service layer application asks for it. The coverage information can be queried from this service layer application through network connection.

We used two possible modes of test execution: manual and automated. Either mode is used, the service layer application must be started prior to the beginning of the execution of the test cases. It is done automatically by the instrumented applications if the service is not running already.

We implemented a simple query interface in Java for manual testing, a plug-in for the RT-Executor [24], and a simple set of functions for the Robotium [16]. The two automated frameworks use different yet somewhat similar approaches.

On one hand, we used the RT-Executor, which reads the test case scripts and executes the test cases. The client side of the measurement framework is contained in a plug-in of the automation tool, and this plug-in must be controlled from the test case itself. Thus, the test case scripts must be prepared in order to measure the code coverage of the executed applications.

The plug-in can indicate the beginning and the end of the particular test cases to the service, so the service can distinguish the test cases and separate the collected information. In order to measure the test case coverages individually, one instruction must be inserted in the beginning of the test script to reset the coverage values and one instruction must be inserted in the end instructing the plug-in to collect and store coverage information belonging to the test case.

During test execution the following steps are taken:

- The program under test (PUT) is started.
- The start of the program triggers the start of the measurement service if necessary. Then PUT connects to the service and registers itself by its unique name given to it in instrumentation process.
- The test automation system starts a test case. The test case forces the client of the automation system to send a **NEWTC** message to the service. The service sends a **reset** signal to PUT, which resets the coverage array in its collector class. The service returns the actual time to the client.
- The test automation system performs the test steps. PUT collects the coverage data.
- The test case ends. The client of the automation tool sends an **ASK** message to the service. The service sends an **ask** signal to PUT, which sends back the coverage data to the service. The service sends back the coverage data and the actual time to the client.
- The client calculates the necessary information from the coverage data and stores it in the local files. The stored data are: execution time, trace length, coverage value, lists of covered and not covered methods. Another plug-in decides if the test case was passed or failed and stores this information in other local files.

These steps are repeated during the whole test suite execution. At the end, the coverage information of all the executed test cases are stored in local files and are ready to be processed by different stages of the testing methodology.

On the other hand, we used the Robotium framework as a black-box test aiding tool, the Android testing API, and JUnit as the testing environment. Robotium provides useful functions to help accessing the graphical user interface layer of Android applications. This way it makes easy to write JUnit test cases which test any application without user interaction.

In this case, the Android framework executes the JUnit test cases like RT-Executor executes its test scripts. The client-side of the measurement framework is contained in a *TestHelper* class that controls data flow during test execution. Similar to the previous settings, this class must be controlled from the test case itself, so the test cases must also be prepared in order to measure code coverage.

The helper class works like the plug-in of the RT-Executor. Thus, the execution steps are very similar to those mentioned above except that only the coverage information is stored at the end.

3.3 Processing the Data

As we mentioned above, the client side of the coverage measurement system is realized as a plug-in of the RT-Executor tool and as an extension to the Robotium framework.

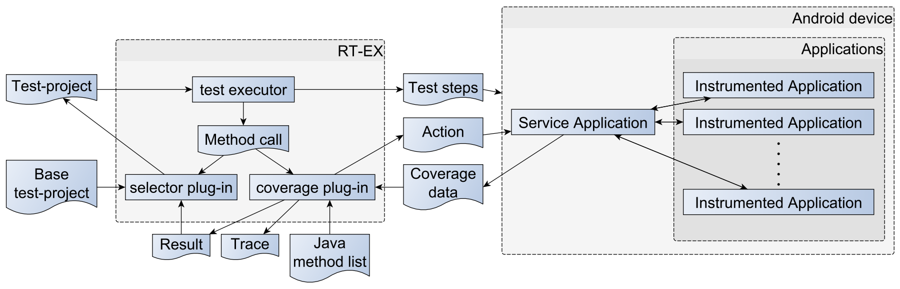


Figure 4: Test execution framework with coverage measurement

In the RT-Executor settings (Figure 4) the plug-in is controlled from the test cases. It indicates the beginning and the end of a test cases to the service layer application. The service replies to these messages by sending the valuable data back. When the measurement client indicates the start of a test case (by sending a **NEWTC** message to the service), the service replies with the current time which is stored by the client. At the end of a test case (when an **ASK** message is sent by the client), the service replies with the current time and the collected coverage information of the methods.

When the coverage data is received, the measurement client computes the execution time, trace length (the number of method calls), and the list of covered and not covered methods' IDs. Then, the client stores these data in a *result* file for further use. The client makes other files, the *trace* files, separately for each test case. Such a trace file stores the identifiers of the methods covered during the execution of the test case.

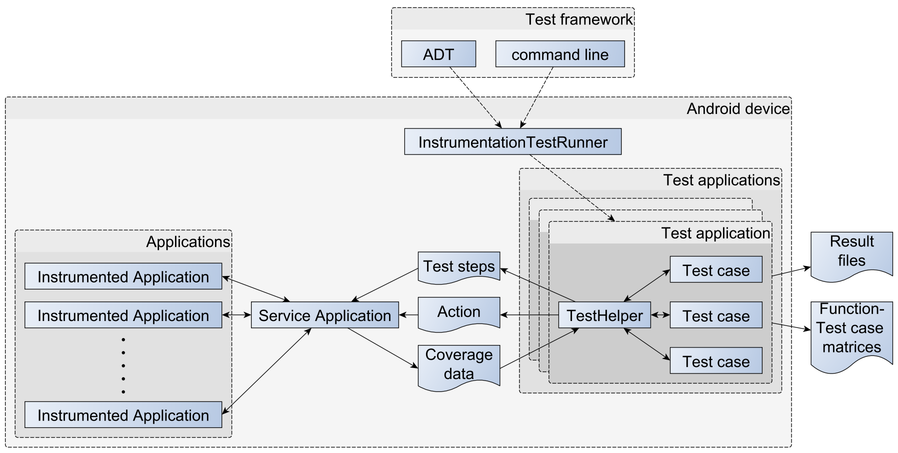


Figure 5: Robotium based test execution environment with the integrated *TestHelper*

In the Robotium settings (Figure 5) the communication between the service layer and the tested application is very similar to the RT-Executor based one. The difference is that the test cases are executed directly by the device and that instead of an external plugin, an internal test helper will communicate with the service application and will produce the coverage data.

As an alternative client, we implemented a simple standalone Java application that is able to connect to the measurement service. This client is able to visualize the code coverage information online, and is useful during the manual testing activities.

3.4 Applications on the Measurement Framework Results

The code coverage and other information collected during the test execution can be used in various ways. In the pilot project, we implemented some of the possible applications. These implementations process the data files locally stored by the client plug-in.

3.4.1 Test Case Selection and Prioritization

Test case selection defines a subset of a test suite based on some properties of the test cases. Test case prioritization is a process that sorts the test suite elements according to their properties [29]. A prioritized list of test cases can be cut at some points resulting in a kind of selection.

Code coverage data can be used for test case selection and prioritization. We implemented some selection and prioritization algorithms as a plug-in of the RT-Executor, which utilizes the code coverage information collected by the measurement framework:

- A change-based selection algorithm that used the list of changed methods and the code coverage information to select the test cases that covered some of the changed methods.
- Two well-known coverage-based prioritization algorithms: one that prefers test cases covering more methods; and another that aims at higher overall method coverage with less test cases.
- A simple prioritization that used the trace length of the test cases.

3.4.2 Not Covered Code

Not covered code plays an important role in program verification. There are two possible reasons for a code part not being covered by any test case executions. The test suite can simply omit its test case, in which case we have to define some new test cases executing the missed code. It can also happen that the not covered code cannot be executed by any test cases, which means that the code is dead. In the latter case, the code can be dropped from the codebase.

In our pilot implementation, automatic test case generation is not implemented. We simply calculate the lists of methods covered and not covered during the tests. These lists can be used by the testers and the developers to examine the methods in question and generate new test cases to cover the methods, or to simply eliminate the methods from the code.

3.4.3 Traceability Calculation

Traceability links between different software development artifacts play a very important role in the change management processes. For example, traceability information can be used to estimate the required resources to perform a specific change or to select the test cases related to the change of the specification. Relationship exists between different types of development artifacts. Some of them can simply be recorded when the artifact is created, some of them must be determined later.

We implemented a traceability calculator that computes the correlation between the requirements and the methods. The correlation computation is based on two binary matrices: the pre-defined relationship matrix between the requirements and the test cases and the matrix between the test cases and the methods (code coverage). From these matrices a binary vector can be assigned to each requirement and method representing whether the test cases assigned to the elements of this vector have relationship to the given requirement or method. If a requirement-method pair is assigned with high correlation (i.e., their assigned binary test case vectors are highly correlated), we can assume that the required functionality is implemented in the method. To calculate the correlation of these binary vectors we implemented three different well-known methods: the Pearson's product-moment coefficient [25], the Kendall's correlation coefficient [25], and a Manhattan distance based method where the similarity coefficient was defined as

$$S_M(a,b) = \frac{1}{1 + \sum_{i=1}^{n} |a_i - b_i|}. \tag{1}$$

The use of the information that is extracted from the results of the correlation computational processes can be diverse. For example, it can be used to assess the number of methods to be changed if the particular requirement changes. Additionally, as we observed during our usability testing project, if we define functionalities closely related to the usage of UI elements, then it can indicate the relations between these graphical elements and the parts of the code-behind.

4 Usage and Evaluation

In this section, we present and evaluate some use cases to demonstrate the usability of the measurement toolchain.

4.1 Additional Test Case Generation

In the pilot project our target embedded hardware was an Android-based Set-Top-Box. We had this device with different pre-installed applications and test cases for some of these apps. Considering the available resources we decided to test our methodology and implementation on a media settings application. After executing the tests of this application with coverage measurement, we found that the pre-defined tests covered only 54% of the methods. We examined the methods and defined new test cases. Although the source code of this application was not available, based on the not covered method names and the GUI, we were able to define new test cases that raised the proportion of covered methods to 69%. This is still far from the required 100% method level coverage, but shows that the feedback on code coverage can be used to improve the quality of the test suite.

4.2 Traceability Calculation

We made two experiments with the framework using it for traceability calculation.

First, in the CIRENE pilot project a simple implementation that is able to determine the correlation between the code segments and the requirements was made.

We did not conduct detailed experimentation in this topic, but we did test the tool. Instead of the requirements, we defined 12 functionalities performed by three media applications (players) on our target Set-Top-Box device. Then, we assigned these functionalities to 15 complex black-box test cases of the media applications and executed the test cases with coverage measurement. The traceability tool computed correlations between the 12 functionalities and 608 methods, and was able to separate the methods relevant in implementing a functionality from the not relevant methods.

In the experiment connected to the usability testing project our direct goal was to investigate whether the coverage information could be used to determine a small set of program locations to be instrumented in order to collect relevant information for usability analysis. The main idea was that by reducing the number of instrumentation points needed for comprehensive usability testing we would be able to minimize the possible negative effects on the performance of the application under test and, therefore, analysing complex applications would become easier. We conducted an experiment involving 10 small to medium sized Android applications (see Table 1). Test cases were created for the applications each one modelling some typical complex usage sessions. Next, we defined some functionalities for each application. This measurement aimed to verify that automatic methods are able to uncover relevant traceability links, and to evaluate the efficiency of different correlation computation methods in indicating traceability links between the artefacts.

Table 1: List of applications used for experiments

Application	Classes	Methods	Functionalities	Test cases
A_0	134	671	4	13
A_1	144	1083	4	25
A_2	303	1675	5	12
A_3	545	2565	9	12
A_4	812	3897	5	14
A_5	861	6760	5	11
A_6	1257	9619	5	12
A_7	1519	11854	5	11
A_8	1537	11166	5	15
A_9	4247	24747	5	12

In order to evaluate the results of the three different computations we examined the functionalities and methods of the applications and created *reference* links between them by manually classifying the methods of each application and connecting them to the functionalities. First the functionalities were determined by usage scenarios. Next, we used some kind of semantic similarity: words semantically connected to the determined functionalities and usual Android UI element name fragments were searched for in the class and method names. Functionalities were initially assigned with the matching elements. Then this initial classification was refined manually by examining each program element and looking for hidden or false *reference* links.

For evaluating the traceability calculation methods and comparing them to our manual method, we used the *precision*, *recall*, and *F-measure* metrics [23]. The first step of assessing these metrics was to compare the manually determined *reference* links to the function-method traceability links that were selected by the different correlation based traceability calculation methods. The comparison of the *reference* and the computed links classified each traceability link as true or false positive, and each lack of link as true or false negative records [23] for a calculation method. Based on this classification of links, the three metric values were computed for each traceability calculation method and for all applications.

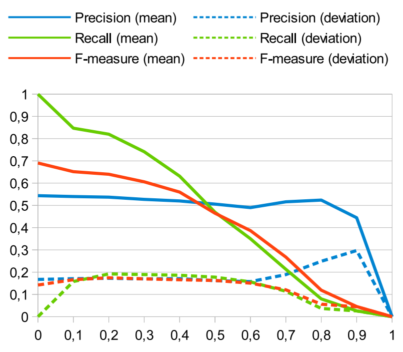
All of the used correlation computation methods assigns a real value within an interval ($[-1,1]$ or $[0,1]$) to a functionality-method pair, but the existence of the link is a binary information. To evaluate the methods we had to define some thresholds to convert real values into true and false values. As the different methods give different numbers, we could not use the same value for all the three ones. Thus, we checked the precision, recall and F-measure values of different threshold values for each methods and computed averages for all applications. The results are shown in Figure 6.

By comparing the curves, we can observe that precision first slightly improves as the treshold grows, then it suddenly drops. Although completeness cannot be totally ignored, for our purposes less noise (fewer false positives) in the generated data is more important than completeness. Thus, we have chosen to select tresholds where the precision is maximal before its drop down. It resulted in 0.8, 0.3, and 0.1 treshold values for methods Pearson, Manhattan, and Kendall, respectively.

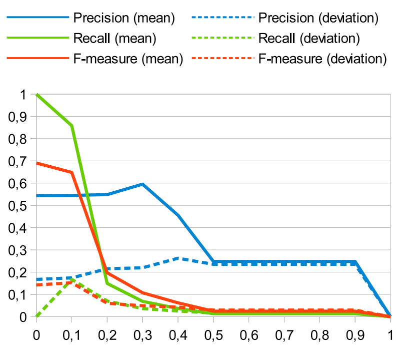
Table 2 shows the precision, recall, F-measure values of the three computation methods using the previously defined threshold values. As can be seen, in half of the cases the Pearson method produces the smallest set, and in four cases of them this is the best choice according to the precision. Manhattan and Kendall methods produce the same smallest sets in three cases and each of them produce the smallest set individually in one case. However, the precision for these sets is always the best among the three methods.

These results show that any of these three methods can be effectively used for calculating traceability between source code and functionalities of a software. For these 10 applications, the Pearson method seems to be slightly better than the other two, but the results are not convincing. Which is the best is probably depending on some other characteristics of the software.

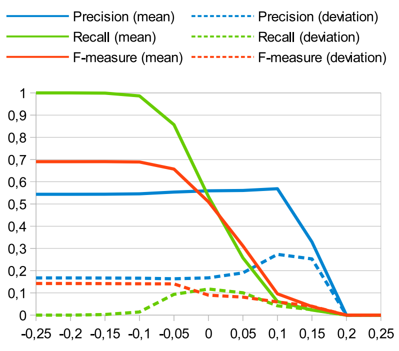
Based on these results we can say that the investigated methods that infer traceability links from code coverage data can be used to identify program points whose inspection provide relevant information for usability testing. The effectiveness of these methods are comparable to the manual traceability link detection. Therefore, it is possible to use them to support the usability testing of large sized Android applications.



(a) Pearson



(b) Manhattan



(c) Kendall

Figure 6: Precision, recall, and F-measure values at different thresholds for the three methods. (X axis: treshold; Y axis: metric value.)

5 Conclusions and Future Work

In this paper, we presented a methodology for method level code coverage measurement on Android-based embedded systems. Although there were more solutions allowing the measure of the code coverage of Android applications on the developers' computers, no common methods were known to us that performed coverage measurement on the devices. We also reported the implementation of this methodology on a digital Set-Top-Box running Android. The coverage measurement was integrated in the test automation process of this device allowing the use of the collected coverage data in different applications like test case selection and prioritization of the automated tests, or additional test case generation. We also presented an application of the framework. Using the produced coverage data we performed experiments with three traceability computation methods.

Table 2: Precision (P), recall (R), F-measure (F) values for applications and computation methods

Application	(a) Pearson 0.8			(b) Manhattan 0.3			(c) Kendall 0.1		
	P	R	F	P	R	F	P	R	F
A_0	0.78	0.13	0.22	**0.79**	0.10	0.17	**0.79**	0.10	0.17
A_1	**0.27**	0.06	0.10	**0.27**	0.08	0.12	0.00	0.00	0.00
A_2	0.35	0.07	0.10	**0.76**	0.04	0.07	**0.76**	0.04	0.07
A_3	0.07	0.01	0.02	**0.28**	0.13	0.16	**0.28**	0.13	0.16
A_4	**0.54**	0.07	0.12	0.46	0.02	0.04	0.46	0.02	0.04
A_5	**0.63**	0.05	0.06	0.54	0.03	0.05	0.54	0.03	0.05
A_6	0.81	0.09	0.14	**0.86**	0.10	0.16	**0.86**	0.10	0.16
A_7	0.44	0.13	0.12	**0.66**	0.09	0.11	**0.66**	0.09	0.11
A_8	0.83	0.10	0.17	**0.84**	0.08	0.14	0.84	0.08	0.14
A_9	**0.52**	0.08	0.19	0.50	0.03	0.05	0.50	0.03	0.05
Average	0.52	0.08	0.12	0.60	0.07	0.11	0.57	0.06	0.10
Deviation	0.25	0.04	0.06	0.22	0.04	0.05	0.27	0.04	0.06

There are many improvement possibilities of this work. Regarding the implementation of code coverage measurement on Android devices, we wish to examine if the granularity of tracing could be fined to sub-method level (e.g., to basic block or instruction levels) without significantly affecting the runtime behaviour of the applications. This would allow us to extract instruction and branch level coverages that would result in more reliable tests. In addition, we are thinking of improving the instrumentation in order to build dynamic call trees for further use. The current implementation (simple coverage measurement) does not deal with timing, threads and exception handling, which are necessary for building the more detailed call trees. It would also be interesting to help the integration of this coverage measurement in commonly used continuous integration and test execution tools.

Furthermore, we are examining the use of the resulting coverage data. There are other ways code coverage and computed traceability information can be used in usability testing, for example to partially automate collecting data and to establish usability models. The implemented code coverage measurement and the testing process that utilizes this information are a good base for measuring the effect of using coverage measurement data on the efficiency and reliability of testing.

References

[1] Antoniol, G., Canfora, G., Casazza, G., De Lucia, A., and Merlo, E. Recovering traceability links between code and documentation. *Software Engineering, IEEE Transactions on*, 28(10):970–983, Oct 2002.

[2] Apache Commons. BCEL homepage. `http://commons.apache.org/proper/commons-bcel/`, June 2013.

[3] Beszédes, Árpád, Gergely, Tamás, Papp, István, Marinković, Vladimir, and Zlokolica, Vladimir. Survey on testing embedded systems. Technical report, Department of Software Engineering, University of Szeged, and Faculty of Technical Sciences, University of Novi Sad, 2012.

[4] Biswas, Swarnendu, Mall, Rajib, Satpathy, Manoranjan, and Sukumaran, Srihari. A model-based regression test selection approach for embedded applications. *SIGSOFT Softw. Eng. Notes*, 34(4):1–9, July 2009.

[5] Bornstein, Dan. Presentation of Dalvik VM internals, 2008.

[6] Chawla, Anil and Orso, Alessandro. A generic instrumentation framework for collecting dynamic information. In *Online Proc. of the ISSTA Workshop on Empirical Research in Software Testing (WERST 2004)*, 2004.

[7] Chiba, Shigeru. Javassist homepage. `http://www.csg.ci.i.u-tokyo.ac.jp/~chiba/javassist/`, May 2013.

[8] Cleland-Huang, Jane, Czauderna, Adam, Gibiec, Marek, and Emenecker, John. A machine learning approach for tracing regulatory codes to product specific requirements. In *Proceedings of the 32Nd ACM/IEEE International Conference on Software Engineering - Volume 1*, ICSE '10, pages 155–164, New York, NY, USA, 2010. ACM.

[9] Costa, José C., Devadas, Srinivas, and Monteiro, José C. Observability analysis of embedded software for coverage-directed validation. In *Proceedings of the International Conference on Computer Aided Design*, pages 27–32, 2000.

[10] Developers. JSON. `http://www.json.org/`, June 2013.

[11] Eaddy, M., Aho, A.V., Antoniol, G., and Gueheneuc, Y.-G. Cerberus: Tracing requirements to source code using information retrieval, dynamic analysis, and program analysis. In *Program Comprehension, 2008. ICPC 2008. The 16th IEEE International Conference on*, pages 53–62, June 2008.

[12] Google. Android homepage. `https://www.android.com/`, June 2013.

[13] Google. apktool homepage. `https://code.google.com/p/android-apktool/`, May 2013.

[14] Google. dex2jar. `https://code.google.com/p/dex2jar/`, May 2013.

[15] Google Android Developers. Building and running an android application. `http://developer.android.com/tools/building/index.html`, May 2013.

[16] Google Code. Robotium homepage. `https://code.google.com/p/robotium/`, March 2014.

[17] Gotlieb, Arnaud and Petit, Matthieu. Path-oriented random testing. In *Proceedings of the 1st international workshop on Random testing*, RT '06, pages 28–35, New York, NY, USA, 2006. ACM.

[18] Hazelwood, Kim and Klauser, Artur. A dynamic binary instrumentation engine for the arm architecture. In *Proceedings of the 2006 International Conference on Compilers, Architecture and Synthesis for Embedded Systems*, CASES '06, pages 261–270, New York, NY, USA, 2006. ACM.

[19] Kukolj, Sandra, Marinković, Vladimir, Popović, Miroslav, and Bognár, Szabolcs. Selection and prioritization of test cases by combining white-box and black-box testing methods. In *Proceedings of the 3^{rd} Eastern European Regional Conference on the Engineering of Computer Based Systems (ECBS-EERC 2013)*, 2013.

[20] Marcus, A. and Maletic, J.I. Recovering documentation-to-source-code traceability links using latent semantic indexing. In *Software Engineering, 2003. Proceedings. 25th International Conference on*, pages 125–135, May 2003.

[21] Marek, Lukáš, Zheng, Yudi, Ansaloni, Danilo, Sarimbekov, Aibek, Binder, Walter, Tůma, Petr, and Qi, Zhengwei. Java bytecode instrumentation made easy: The disl framework for dynamic program analysis. In Jhala, Ranjit and Igarashi, Atsushi, editors, *Programming Languages and Systems*, volume 7705 of *Lecture Notes in Computer Science*, pages 256–263. Springer Berlin Heidelberg, 2012.

[22] Poshyvanyk, D., Gueheneuc, Y.-G., Marcus, A., Antoniol, G., and Rajlich, V. Feature location using probabilistic ranking of methods based on execution scenarios and information retrieval. *Software Engineering, IEEE Transactions on*, 33(6):420–432, June 2007.

[23] Powers, David MW. Evaluation: from precision, recall and f-measure to roc, informedness, markedness & correlation. *Journal of Machine Learning Technologies*, 2(1):37–63, 2011.

[24] RT-RK Institute. RT-Executor. `http://bbt.rt-rk.com/software/rt-executor/`, May 2013.

[25] Salkind, N.J. *Encyclopedia of measurement and statistics*. Number 1. k. in Encyclopedia of Measurement and Statistics. SAGE Publications, 2007.

[26] Seesing, A. and Orso, A. InsECTJ: A Generic Instrumentation Framework for Collecting Dynamic Information within Eclipse. In *Proceedings of the Eclipse Technology eXchange (eTX) Workshop at OOPSLA 2005*, pages 49–53, San Diego, CA, USA, october 2005.

[27] Slife, Derek and Chesney, Mark. jCello. `http://jcello.sourceforge.net/`, June 2013.

[28] Vlad Roubtsov. EMMA: a free java code coverage tool. `http://emma.sourceforge.net/`, June 2013.

[29] Yoo, S. and Harman, M. Regression testing minimization, selection and prioritization: a survey. *Software Testing, Verification and Reliability*, 22(2):67–120, 2012.