Cirene
Cross-border ICT Research Network

Hungary-Serbia
IPA Cross-border Co-operation Programme
Good neighbours creating common future

Test System Architecture

Abstract

This report is part of the CIRENE project that aims (amongst other things) the definition of a general embedded system testing methodology. The project incorporates a pilot implementation of the methodology in a selected environment. This document is part of this pilot implementation as it describes the architecture of our pilot testing system, which aims the testing of the selected Android-based Set Top Box (Digital TV decoder). Previously, a special methodology was defined in [1] for the pilot testing system and this document gives the overall architecture and the details how that special methodology should be implemented in the pilot environment.

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

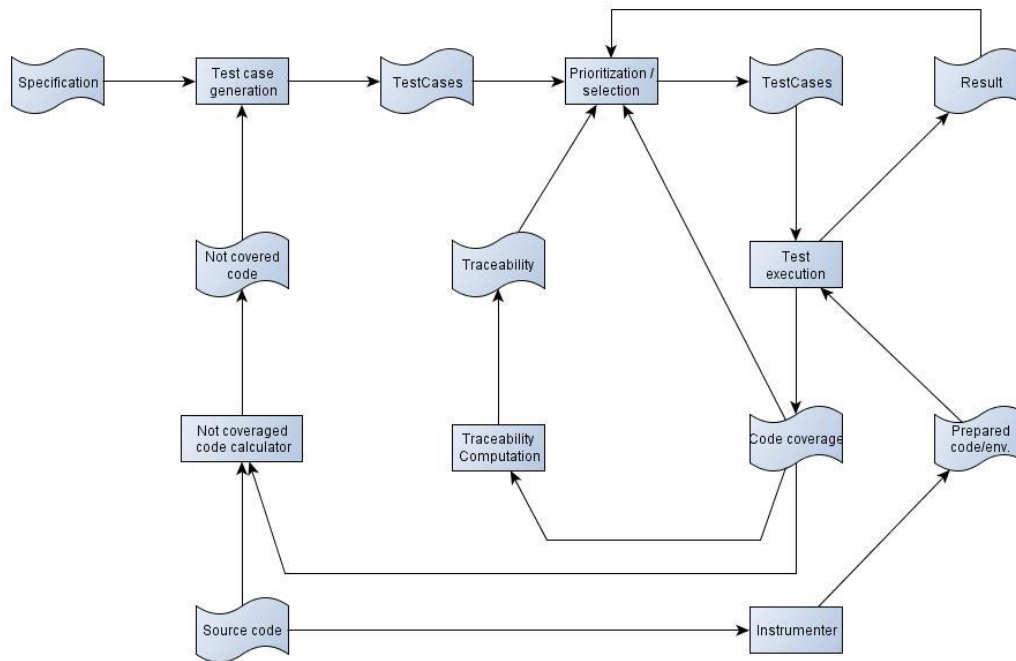
The project is co-financed by the European Union

Table of contents

Abstract 1
1 Overview 3
2 Prioritization/Selection 5
2.1 Test project 5
2.2 Selection 5
2.3 Prioritization 6
2.3.1 Prioritization based on longest traces of execution 7
2.3.2 Prioritization based on test cases that cover the most functions 7
2.3.3 Prioritization based on test cases that additionally cover the most functions 7
2.3.4 Prioritization based on test cases that have failed most times 8
2.3.5 Prioritization based on runtime information 8
2.3.6 Additional prioritization based on test cases that cover not yet covered functions 8
2.4 Cut 8
3 Not covered code calculator 10
4 Traceability computation 11
5 Instrumentation 12
6 Service application 14
7 Communication 15
7.1 Local File 15
7.2 Network connection 15
8 Test execution 17
9 References 19

1 Overview

In this section we give an overview of the embedded system testing methodology specialized for our pilot system.



The specialized embedded system testing methodology can be seen on the figure above. This methodology describes how the Android-based Set Top Box device (our pilot system under test) should be tested, and what information could be reused to improve testing. In this project the focus is on gathering and using the code coverage information. Computed coverage information can be used in test case selection and prioritization as well as to infer traceability links between different artifacts (like test cases, methods, requirements, etc.) or to determine code fragments that are not executed by the set of all test cases. Not covered code can help to define additional test cases (and improve test quality), or might be identified as “dead code”, and serves as the base of some refactoring for improved software quality.

To measure code coverage, the tested software must be prepared in order go gather execution traces. In our pilot system it is done through code instrumentation prior to test execution.

The pilot project does not cover automatic test case generation. Although we use **MaTeLo** to support test case generation, we treat this step as a black box. We use Java bytecode instrumentation to insert trace generation functionality in the tested software. Test execution is done by the **RT-Executor**. During test execution, RT-Executor plugins collect coverage information from a server application on the Set Top Box device. Similar plugins perform test selection and prioritization.

In the following sections, the implementation steps of the specialized methodology are described in details.

2 Prioritization/Selection

Test selection and prioritization is implemented as an **RT-Executor** plugin. The plugin works on test project files. It reads the list of test cases, performs test selection and prioritization, and writes the new list in another test project file. In the next subsections, the technical details are presented.

2.1 Test project

We generate **BBT** project file with **RT-Executor** program. The file contains the runnable test cases. The first some line of the project file are constant which describe some attribute of the project. These lines are the following:

```
<RT-RK BlackBox Test Tool Project file>
<Project name: Projectname>
<Project database path: Results>
<Project HTML path: Results>
<Project update: No>
<Project repeat: No>-
```

Furthermore, the project file contains list of tests. The lines of the list like the next:

```
[\MISC_001_001\MISC_001_001.tst\]
```

This refers for the MISC_001_001 test case, which is in the RT-Executors Test folder, in the MISC_001_001 subfolder. Additional resources are placed here, too, like as reference pictures, data files.

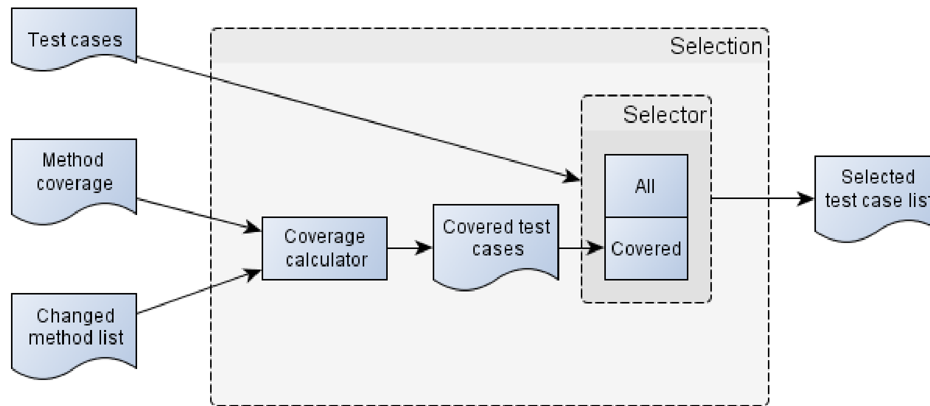
The generated file contains more test cases and defines their order. The program uses coverage, trace and runtime information when it generates the project file. The output has the proper format for **RT-Executor**.

2.2 Selection

The test case selection is based on the information stored in the criterion file which contains information on code coverage.

There are two options for selecting desired test cases: either the selection of all available test cases, or the selection of only those test cases which cover functions that have been changed since the previous executions.

The input for selection step is a text file, containing the test case list with all the available and necessary information.



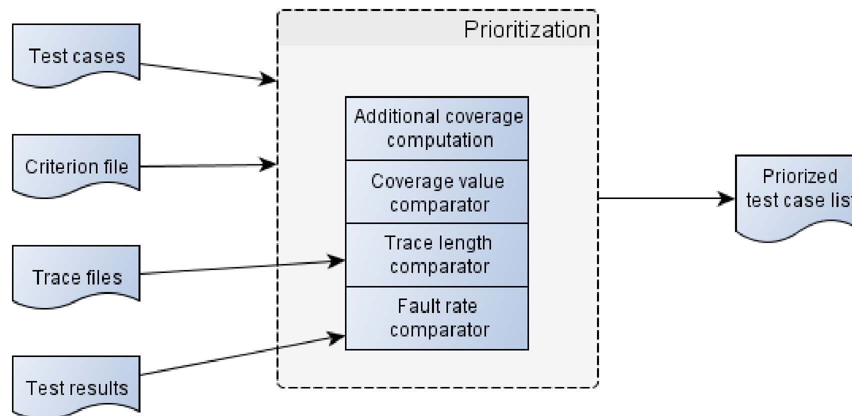
The above figure shows the methods of selection. The two methods are the *"All"* method which select all test case and the *"Covered"* method which selection of only those test cases which cover methods that have been changed since the previous executions. The *"Changed method list"* contains changed methods and the *"Method coverage"* is the list of relations between methods and test cases.

2.3 Prioritization

Test cases can be prioritized in four ways:

- Additional coverage computation
- Coverage value comparator
- Trace length comparator
- Fault rate comparator

The following figure shows the inputs and output of the prioritization process.



2.3.1 Prioritization based on longest traces of execution

Test cases with longer traces have higher probability to cover larger number of different functions. To utilize this higher probability, this prioritization strategy assigns higher priorities to test cases with longer traces. This prioritization is a simpler, heuristic version of the next one. It can be misleading because of the loops in the methods that expand the trace length.

2.3.2 Prioritization based on test cases that cover the most functions

In this strategy, test cases covering more methods get higher priority. Test cases covering more methods have higher probability to cover faulty methods and/or to examine a more complex execution path of the software. However, it might happen that test cases executing the same group of methods have high rank and some other methods are covered by only low-priority test cases, which may result in low coverage value after cutting this list.

2.3.3 Prioritization based on test cases that additionally cover the most functions

Here, test case with largest coverage value is selected first, and priorities for all other test cases are recalculated based on the methods that have not been covered yet by previously selected test cases. The recalculation of the priorities of the remaining elements occurs after the selection of every new element. In this way we can select the minimal number of test cases that covers the most methods.

2.3.4 Prioritization based on test cases that have failed most times

Test cases that failed most times in a given period (usually the last some days or weeks) have higher priorities than those that failed less times (or did not fail at all in period). These test cases may refer to program parts that probably contain mistake or error.

2.3.5 Prioritization based on runtime information

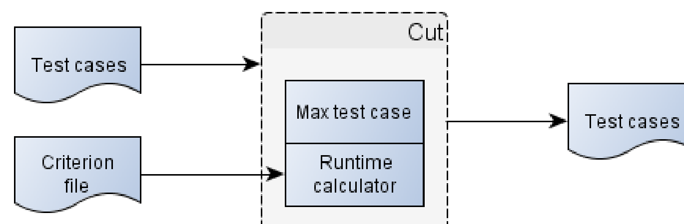
Here we sort the test cases using their execution time. Both ascending and descending orders are available based on the aims of the testing. We might prefer long execution times that probably (although not necessarily) implies that the test case covers larger part of the software. However, we can also prefer test cases with short execution times that probably exercise specific parts of the software.

2.3.6 Additional prioritization based on test cases that cover not yet covered functions

This is a special prioritization algorithm. It is similar to the additional one, but works on a higher level. Namely, when a test round is done it covers some methods and leaves the other methods uncovered. This algorithm takes this list of uncovered functions into account, and prefers test cases that cover this list more.

2.4 Cut

If we have a prioritized list of test cases, we can lessen the number of later executed test cases in a simple way: we cut the list. The remaining elements of the list can placed in the test project.



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

- If the time for testing is limited, and execution time data of the test cases is available, the prioritized list of test cases can be cut in a way that the cumulative execution time of the

remaining test cases does not exceed the time limit. The cut can be used with any prioritization strategies.

- If the customer fixes or maximizes the number of test cases, or any other case exists that gives a limit on the number of test cases, than we need to cut the list by choosing the given number of elements from the top of the list. We can use any of the prioritization mentioned in the previous section before cut the list.

The project is co-financed by the European Union

3 Not covered code calculator

The not covered code calculation is easy task because all necessary information will available after the test execution.

The first step of the not covered code calculation is that **Coverage plug-in** collects all unique method identifier from the instrumentation phase. **Coverage plug-in** gets the code coverage information after tests execution. Then it compares the coverage information and all method identifier lists. Those methods doesn't exist in the coverage information are not covered.

More precisely, in the array **Coverage plug-in** gets from the service application, where the value is not null, the methods with the proper IDs are reached. **Coverage plug-in** can easily collect the reached methods from the entire testing process, and it compares this list to the whole method list generated during instrumentation.

4 Traceability computation

As we mentioned in [1], we use four artifacts: test cases, requirements, functionalities and methods of the program under test. We would ascertain the connections between test cases and methods, test cases and functionalities, test cases and requirements, and between methods and requirements. If the functionalities differ from the requirements, than we compute the link between methods and functionalities, too. Next we talk about these in details.

In the beginning of the testing process there is the test case generation. We mentioned that it will be made by hand, so when the test cases are made based on the specification we can record the relations between requirements and test cases into a file. The test cases originally need to have a unique ID and we can assign unique IDs to the requirements. In this way can make an easy-to-work-with file for our tools. This would provide the traceability of requirements and test cases.

The next type of traceability links are between the test cases and the methods of the software. This information is what we call code coverage in this project (as we use method level code coverage), and what is directly computed during test case execution using instrumentation techniques as it well be detailed in the next sections. The traceability information (code coverage data) is stored in trace files, one for each test case.

The two types of traceability information above can be achieved or computed directly. However, it might be interesting to see the relationship between the requirements and the source code. Namely, we might be interested in the methods implementing some specific requirements. It is not a simple task to answer this question correctly because a test case executes not only methods that are specific to the requirement the test case is derived from, but additional common or "utility" methods are also executed, which do not directly belong to the implementation of the requirement. Thus, based on the existing traceability links, some statistical or heuristic methods should be used to determine the relationship between requirements and methods.

If we define functionalities of the software, we can also compute traceability between functionalities and methods in the same way as between requirements and methods. To do this, we will need to define relationships between test cases and functionalities and/or between requirements and functionalities. Acquiring the later relationship is more probable if we have access to the functional specification and the developers of the SUT.

5 Instrumentation

The instrumenter application is a Java application which can inject special code into Java bytecode. We are using **javassist** to instrument the applications (selected from several other solutions). As the instrumentation is done on java **class** files, first we need to *gather* them. Using **apktool** we unpack the *apk* file from which we get the *dalvik bytecode* of the application (**classes.dex** file). We can extract the original **class** files from the *dalvik bytecode* with a converter application (**dex2jar**). Then we execute the instrumenter application with two parameters. The first parameter is the path of the jar file (which is the output of **dex2jar**). The second parameter is the project name which the instrumented application uses for identification when it registers itself in the service. An important thing is that each application should be given a unique name so that service application can manage them.

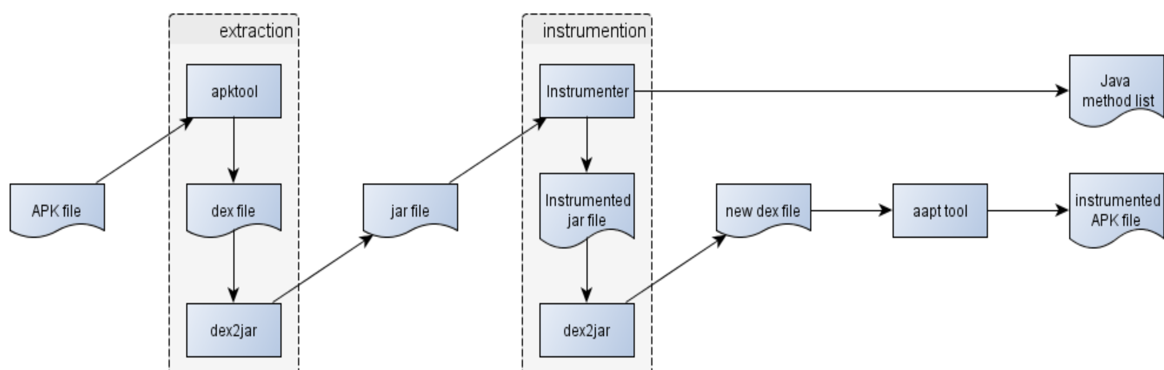
Executing the instrumenter application on these class files will insert additional lines into the bytecode. The additional code we want to add to the application the CoverageCollector class which has function is the communication to the service application. The CoverageCollector class contains the next array when *MethodNumber* is number of methods in application.

```
static int[] coverage = new int[MethodNumber];
```

The following instruction will be inserted to the beginning of every method:

```
CoverageCollector.coverage[MethodId]++;
```

The modified class files are then converted back to *dalvik bytecode* using the **dex2jar** application.



The above figure shows the instrumentation method.

During the execution of the instrumented application, it is the responsibility of the coverage collector class to connect to the service application, to collect coverage information and send it to the service upon a proper query message.

The instrumenter has a csv output too. This file contains the signature of all methods and their identification number. First line of file is:

```
ProjectName;NAME
```

where NAME is project name which we gave to the instrumenter application with parameter.

Other lines' structure is:

```
ID;HEADER;CLASS
```

where *ID* is the unique identification number for method, *HEADER* is method signature and *CLASS* is the path of the class which contains the method. E.g.:

```
2;public float add(float, float);com.coverageTest.Calculator
```

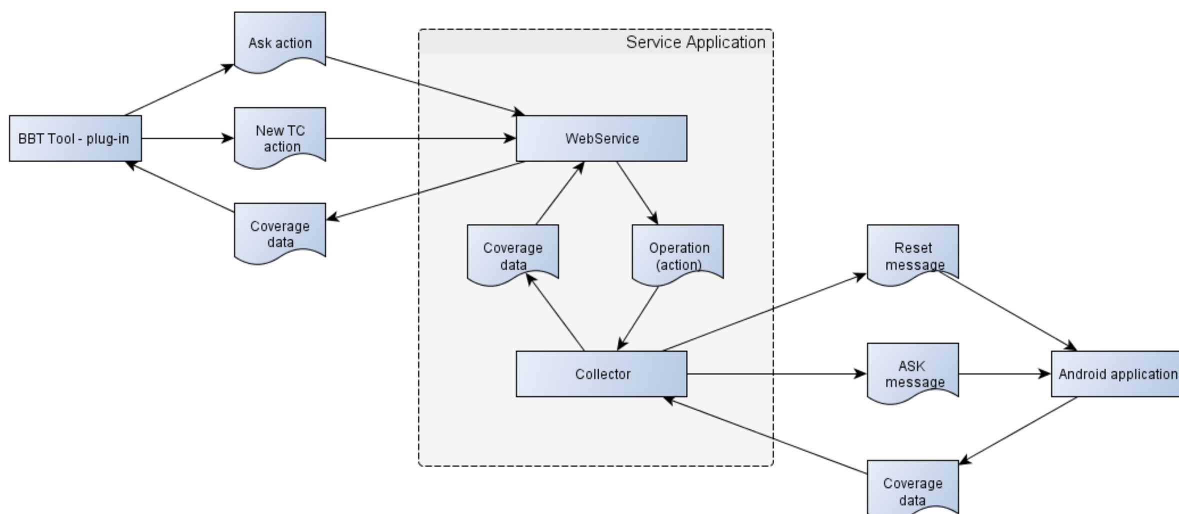
where the *com.coverageTest.Calculator* class contains the *public float add(float, float)* method.

6 Service application

The Service layer of an Android system is an application component representing either an application's desire to perform a long-running operation while not interacting with the user or to supply functionality for other applications to use. Our coverage service accumulates the coverage information and when it takes a request from a client then it passes them to it.

The coverage service application has to be installed already when the instrumented applications are executed. When the instrumented application is starting, it registers itself in the coverage service by sending its project name to the service which uniquely identifies the application in the future. The coverage service asks coverage information when it takes an "ask", "new test case" or action from the plugin. The service preserves information after instrumented application is terminated.

In details, the service sends the actual timestamp in milliseconds to the client when gets the "new test case" action. During test execution it collects the information from the methods into an array. When it gets the "end test case" action, sends the array and the actual timestamp again. The further calculations of the properties are made in the **RT-Executor** plugin. The next figure shows the structure of service application.



7 Communication

7.1 Local File

We use different files for communication between applications. These files are:

- **Method list**: this is a csv file which contains header and path of all methods. It is formed when we are instrumenting an application. The **BBT tool** asks this file as a parameter.
- **Coverage plug-in result**: this is a csv file and contains number of method calls, percentage value of coverage, execution time and result (PASS, FAIL) for every executed test case. **Coverage plug-in** creates this file when the test system finished the execution.
- **Coverage plug-in trace**: this is a csv file. Information of the executed methods is found in this file. **Coverage plug-in** creates this file when the test system finished the execution.

7.2 Network connection

The communication in the pilot testing system during test execution happens on two levels. The first level is between the instrumented application and the coverage service application on the Set Top Box device. This communication is very simple. The service application can send two different kinds of messages:

- **Ask**: the message forces the instrumented application to send coverage information to the service.
- **Reset**: the message forces the instrumented application to reset the coverage information, thus treat all functions of the software as not executed.

The second level is between the device and the computer executing the tests. The connection is TCP/IP-based and JSON objects are used to communicate between the coverage service and test execution clients. (The connection via LAN or Wi-Fi is requirement of communication.) The messages can be the following:

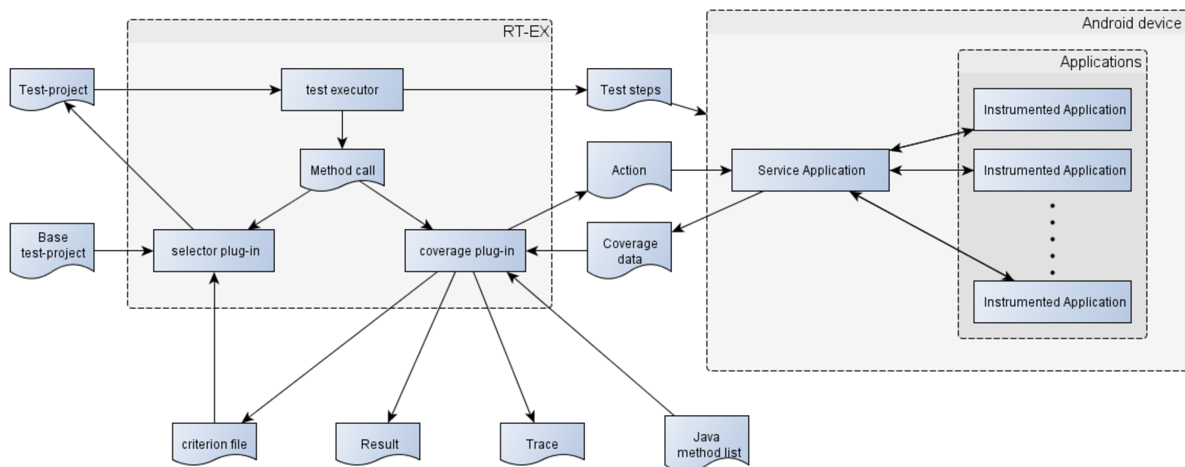
- Client sends to service: {“action” : “*ACTION*”}, where *ACTION* is selected from next list:
    - **NEWTC**: A new test case is going to be executed. The service sends **Ask** and **Reset** messages to instrumented application.
    - **ASK**: Query actual coverage information. The service sends an **Ask** message to the instrumented application and the returned information is sent to the client.

- Service sends to client: {“timestamp”: “*TIME”*, “data”: {“*NAME1*”*:* [*LIST*], “*NAME2*”*:* [*LIST*]}}, where *TIME* is the current time, NAMEx are *“project name”* of the instrumented application and *LIST* are the coverage information. Description of *LIST* is [*M1*, *M2*, *M3*, …] where *Mx* are numbers denoting that the corresponding method was called *Mx* times, and their positions in the list corresponds to the method. Position number of the method equals the method ID from **Method list file.**

The service application uses a web service which waits for requests and when receives one it process that. We are using **restlet** java library with the web service.

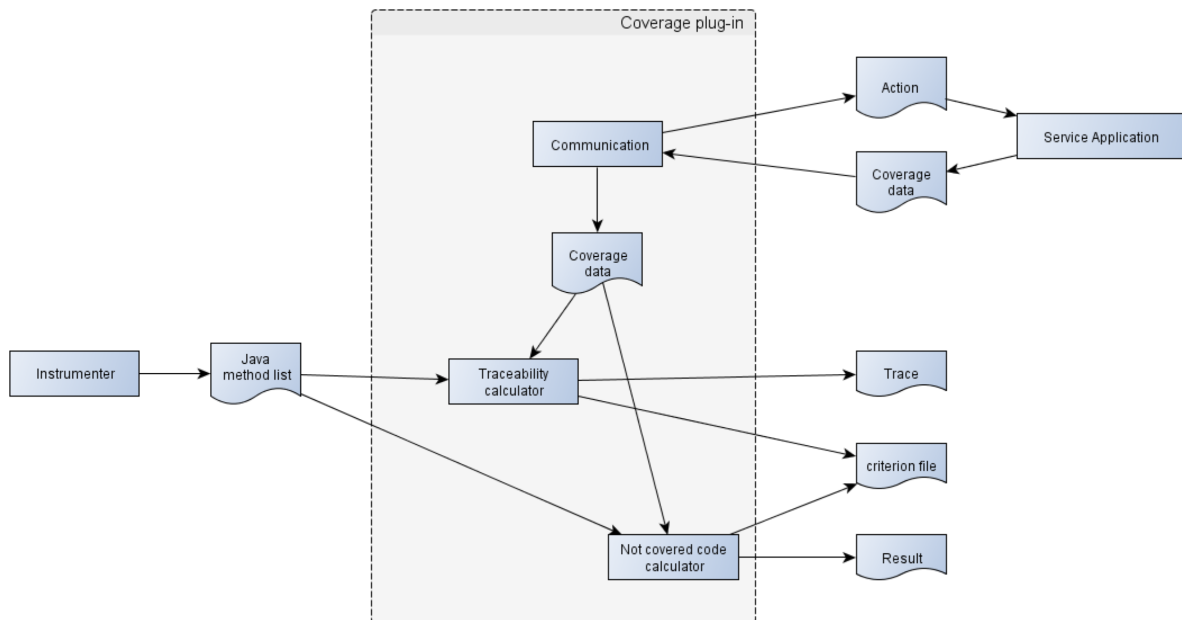
8 Test execution

When we execute the test project we are running the **RT-Executor** tool which executes the test cases one by one. We created a plugin to **RT-Executor** tool which connects to device via TCP/IP network, sends control messages and retrieves coverage information. The plugin sends "new test case" message when a test case starts. When a test case finished running the plugin asks coverage information and writes it to a file in the directory of the test case. The following figure shows the architecture and data flow of the test execution of the pilot testing system.



The system has two main parts. The first is the *RT-Executor* which includes the test executor and different plug-ins. The test executor performs automatic test case execution by sending *test steps* to the set top box device through its remote controlling interface. The test executor controls the *selector* and *coverage plug-in* through method calls. The coverage plug-in is responsible for the communication with the server application on the Android device, and for processing the acquired coverage data. The input of coverage plug-in are *java method list* and *coverage data* (from the STB device), its output are the *result, trace* and *criterion file*. The selector plug-in performs test case selection, prioritization and cut. The inputs of *selector plug-in* are the *test cases* and the *criterion file* and its output is a *test project* that can be loaded by the test executor. The second part is *android device* which contains the *service application* and the *instrumented applications*.

The next figure shows the structure of the coverage plug-in.



The coverage plug-in contains three parts: *communication*, *traceability calculator* and *not covered code calculator*. The *communication* component connects to the service application and gets *coverage data* what it passes on *traceability* and *not covered code* calculators. Inputs of the traceability calculator are *java method list* from *instrumenter* and *coverage data* from *communication component*. Outputs of the *traceability calculator* are *trace file* and the *criterion file* (which is only partially processed). Inputs of the not covered code calculator are *java method list* from *instrumenter* and *coverage data* from *communication component*. Outputs of the not covered code calculator are *result file* and a part of *criterion file*.

The project is co-financed by the European Union

9 References

[1] Specialized Methodology. CIRENE project technical report. 2012.