A Strategy to Perform Coverage Testing of Mobile Applications*

M. E. Delamaro
Centro Universitário Eurípides de Marília
Marília, Sao Paulo, Brazil
delamaro@fundanet.br

A. M. R. Vincenzi
Instituto de Informática
Universidade Federal de Goiás
Goiânia, Goiás, Brazil
auri@inf.ufg.br

J. C. Maldonado
Instituto de Ciências Matemáticas e de Computaçao
Universidade de Sao Paulo
Sao Carlos, Sao Paulo, Brazil
jcmaldon@icmc.usp.br

ABSTRACT

The development of wireless application has recently received more attention due to the increment in the number and in the power of mobile devices such as PDA's and cellular phones. Different methods and techniques have been developed to ease the design and development of applications for these kind of devices. Also, different languages have been proposed to provide support for such platform, such as J2ME and Brew. On the other hand, few attention has been given to testing activity in this scenario. Some works try to test the functional aspects of a given application, others try to perform load, usability and stress testing. In this article we present a strategy to support coverage testing for mobile device software in such a way that the applications can be tested not only on emulators, but also on their real target mobile devices with the aid of structural coverage assessment. We also present an environment which supports the proposed strategy. Such environment is implemented in a tool, named **JaBUTi/ME**. A simple case illustrating how **JaBUTi/ME** can be used is also presented.

Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: Testing and Debugging—*Testing tools*

General Terms

Verification

Keywords

Mobile Application, Coverage Testing, Testing Environment

*This project is supported by the Brazilian Funding Agency – CNPq – 478001/2004-5

AST'06, May 23, 2006, Shanghai, China.

1. INTRODUCTION

One of the fundamental aspects to the development of high quality software products is to have a well-defined software development process, but even with such a rigorous software process, testing is still necessary. Different testing criteria have been proposed aiming at establishing a set of testing requirements which must be satisfied by a test set, systematizing the test case selection and/or evaluation. Such testing criteria can be classified as functional, structural or fault-based according to the source of information used to derive the set of testing requirements. In this article we are interested on structural testing criteria which use the software product implementation, and the testing requirements are structures of the implementation, such as, statements, decisions, data-flow interactions, and so on. Several structural testing criteria have been developed [2, 6, 8] and the application of such criteria requires the existence of testing tools to support their application [1, 5, 12, 15]. One such tool, named **JaBUTi**, allows the application of control- and data-flow testing criteria on Java programs. The difference of **JaBUTi** from others testing tools is that it performs the static analysis directly on Java bytecode not on Java source code, allowing the application of the supported criteria on non-conventional applications, such as mobile agents [4] and software components [14].

With the growth of the wireless market, a great number of devices with different operational systems, technologies and hardware have been created. The development of mobile applications has growing in the same manner to add value to the mobile device itself and also to attract customers to buy a given device. If on one hand new technologies bring benefits for a general user, on the other hand they introduce new obstacles for software developers and software engineering scientists. In this context, one of the most accepted platform for software development is J2ME [9] which allows the use of Java language in the development of applications for wireless devices and brings to this scenario the advantages of such language, for instance, software portability.

In this article we discuss the problems of testing mobile applications. While most part of the works related with testing mobile applications concentrate efforts on performance, load and usability testing, we focus on the application correctness and we are interested in the development of an architecture which allows to perform coverage testing of mobile applications.

According to Mahmoud [7], the testing of wireless application is different from testing traditional application since, in general, wireless application running conditions vary enormously. Considering the memory limitations, persistent storage, network connection availability and others, the software products for mobile devices are developed cross-platform, as shown in Figure 1. First,

the application is developed on a high-end desktop and then deployed on mobile devices with very different characteristics. The test of such programs are carried out on two distinct steps, first on the desktop using emulators, and second, on the target mobile device itself, since the emulators cannot assure complete compatibility with the target device [11]. Observe that with the limitations of the mobile devices, it is not practical to run a testing tool directly on the mobile device. Therefore, the solution is to provide a strategy that allows the test to be executed cross-platform, i.e., in some way, the information about the execution of a given mobile application should be collected and provided to a testing tool which analyzes and generates reports about the coverage of the executed test cases. In this paper we address the problems of defining such strategy and propose an environment to support it. Such an environment, named **JaBUTi/ME**, extends **JaBUTi** and implements the proposed strategy. It is also described an example showing how **JaBUTi/ME** can be used to perform coverage testing on mobile applications.

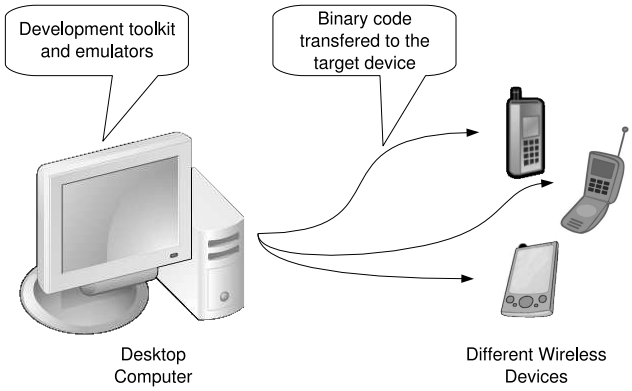


Figure 1: Traditional wireless application development/testing process.

Considering this objective, in Section 2 main related work is described and compared with our approach. Section 3 describes the proposed strategy and environment for applying structural testing on mobile device applications. In Section 4 an example is presented to illustrate how such an environment can be used to support the coverage testing of mobile applications. Finally, in Section 5 we present the conclusions and future work related to this research.

2. RELATED WORK

As any new technology, the development of software for mobile devices requires different methods for development and testing to be created or adapted from the existent ones. Although several works describing new methods for the development of mobile applications exist, only a few of them take care of testing issues. Moreover, even the ones specifically addressing testing are mainly concerned with performance, usability and compatibility testing instead of with the correctness aspect of the software.

Cundy [3] discusses the differences on testing mobile applications versus traditional ones, considering both hardware and software. He enforces that a strategic approach for testing mobile solutions needs to take into account the particular characteristics of the mobile paradigm, such as: the increased complexity of the mobile devices; the greater sensitivity to security and load problems; and the increased complexity of scale. The author also highlights that it is impossible to provide a bug-free program but it is essential to detect, fix and prevent bugs as earlier as possible to reduce the development cost. Although he discusses several points that need to be taken into account when testing mobile applications, he did not provide any solution, method, technique or testing criterion for this kind of applications.

In the same line, Mahmoud [7] discusses the problem of testing mobile Java applications and presents a checklist to be followed when testing such applications. He states that like all other software, mobile applications must be tested to ensure their functionality and usability under all working conditions (which varies a lot in the wireless world). Although no specific technique nor testing criterion is defined, the author stresses the necessity of performing unit, integration and system testing considering not only functional testing but also structural testing, evaluating the code coverage. The strategy we propose in this paper can be used to detect bugs on mobile application during its development (unit testing), providing part of the solutions identified by Cundy [3] and Mahmoud [7].

Satoh [10, 11] developed a framework and infra-structure to test mobile applications considering the problem that it is difficult to validate the applications on their real environment. He implemented emulators, called Flying Emulator, as mobile agents such that the agents travel to the wireless network encapsulating both the mobile application itself and an emulator for its target platform. In this way it is possible to evaluate the application's behavior on such wireless network without the need of the tester to move across different networks to validate a given mobile application behavior. Although such an architecture can be used to perform coverage testing it was not designed to do so. As stated by Satoh, its main purpose is to enable application-level software to be executed and tested with the services and resources provided through its current network as if the software was being moved and executed on that target device when attached to the network. Moreover, for each possible wireless device (supposed to be) available on the network it is necessary to implement the mobile-agent-based emulator which is not a trivial task. In our strategy no emulator implementation is required since our main purpose is to test an application on its real target mobile device.

With respect to the automation of the testing activity for mobile applications, one of the pre-requirements is that the tool supports the test of binary code since not always the source code is available. Considering such requisite, different testing tools can be identified, such as J2MEUnit [12], GlassJar Toolkit [5] and **JaBUTi** [15], which support functional, control-flow and data-flow testing, respectively. However, none of them supports the testing of mobile application directly on the target device, i.e., all of them allows the testing of mobile application only on the computer desktop through emulators. The strategy and environment proposed in this paper allows the testing of mobile applications not only on emulators, but also on the desired target mobile device, as illustrated in Figure 2. Observe that this is an important testing requirement for mobile applications since the emulators are not totally compatible with the original target device [11].

3. STRUCTURAL TESTING ON MOBILE DEVICES

Structural testing is a technique based on the internal structure of a given implementation, from which the test requirements are derived. In general, structural testing criteria use a representation known as Control Flow Graph (CFG) to abstract the structure of the program or of part of the program, as a procedure or method.

Figures 3(a) and 3(b) show an example of a method in its source form and the corresponding CFG that abstracts the control structures of the program. The CFG is a digraph on which each vertex (node) represents an indivisible block of statements and each edge

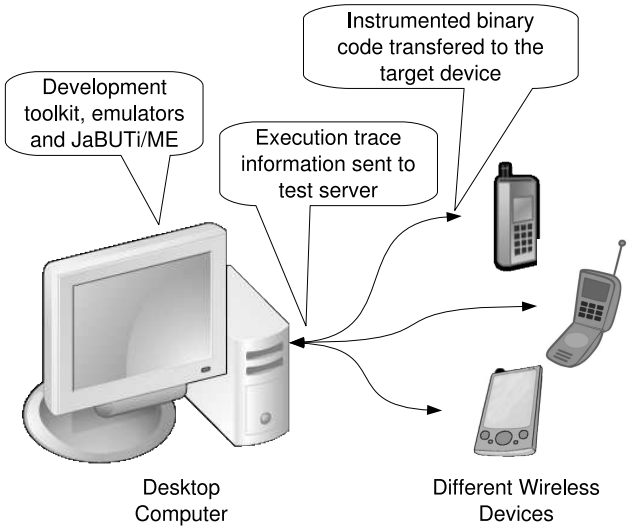
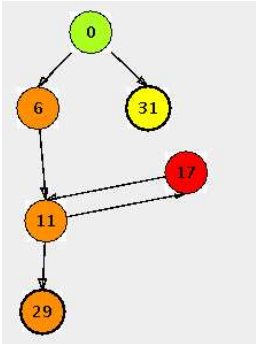


Figure 2: Server-based testing: JaBUTi/ME.

represents a possible sequence of execution from one block to another. From the CFG one can establish requirements to be fulfilled by a test set as, for instance, the execution of each vertex or each edge at least once. Because such criteria use only characteristics related to the control structure of the program to determine the set of testing requirements, they are called "control-flow based criteria".

```
long compute(int x)
{
   if (x <= 10)
   {
      long r = 1;
      for (int k = 2; k <= x; k++)
      {
         r *= k;
      }
      return r;
   }
   else
   {
      return (long) x * compute(x - 1);
   }
}
```

(a) Original Program



(b) Control flow graph (CFG)

Figure 3: Example of a CFG.

By extending the CFG with information about variable usage, other class of criteria can be defined, based on the program's data-flow. The idea in this case is to require the execution of paths in CFG that go from points where a variable is assigned a value (a definition of the variable) to points where that variable is used (a use of the variable), without redefining its value. In this way, each value computed in the program should be exercised on a subsequent use by a "definition free" path. Such a graph is called "def-use graph" [6].

In order to apply the structural testing technique, a tool is necessary to perform, at least, the following tasks: 1) static analysis of the program code and creation of the CFG; 2) generation of the structural testing requirements to be fulfilled; 3) code instrumentation to make it possible to collect, for each execution of a test case, which pieces of the code have been executed; and 4) analysis of which testing requirements are covered by the execution of the test cases. Figure 4 summarizes the features of such a tool. The top part of the figure corresponds to those static analysis tasks, represented by the mentioned items 1 to 3. The bottom part corresponds to the dynamic aspects, that correspond to to the execution and analysis of the test (item 4).

The **JaBUTi** tool implements the support to structural test on Java programs and components, according to the diagram in Figure 4. The tool allows the extraction of testing requirements and their analysis by some different types of visualization tools as source code view, object code (Java bytecode) view, def-use graph view or by several kinds of coverage reports.

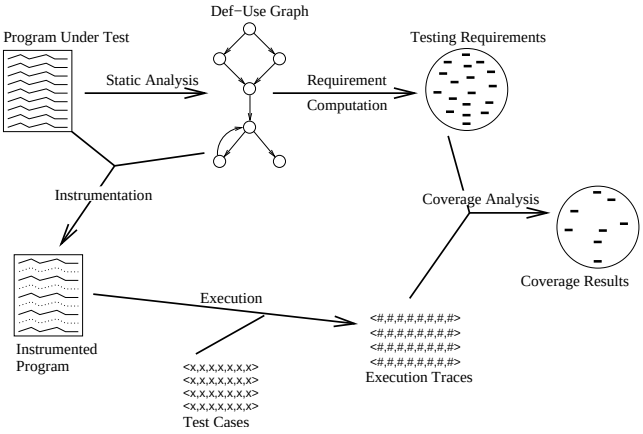


Figure 4: Summary of a structural testing tool.

On the left side of Figure 5 there is an example of visualization using both, source-code view and def-use graph view. Each color in these views (gray levels in this text) represents the weight of a given requirement. More details can be found somewhere else [13]. The tool provides also the functionalities of the bottom part of Figure 4, i.e., the instrumentation of the program under test and execution of test cases using the instrumented version. Such functionality is represented in the right side of Figure 5. The execution of the instrumented program produces, besides the ordinary behavior of the original program, a list of all the points in the program reached by the test case execution.

The instrumentation of the program under test is essential to the application of structural testing. In the case of **JaBUTi**, it inserts at the beginning of each basic block, corresponding to the nodes of the CFG, a call to a method that takes care of registering which piece of the code is about to be executed and stores that information. Such a method is place in a special class that gathers some other func-

tionalities such as initialization and finalization of the collection of trace data and its delivery to the tool. In this way, besides the code inserted in the middle of the original program, the instrumentation process aggregates new classes to the program. Those classes manage the collection, storing and delivery of the execution data (trace data) and become part of the program being tested.

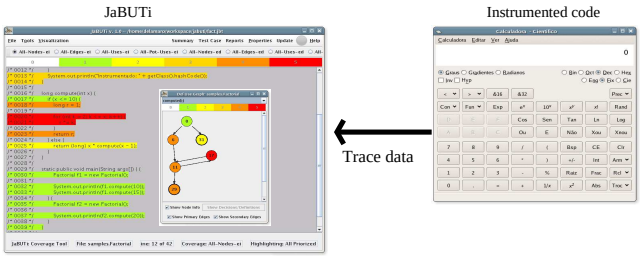


Figure 5: Example of JaBUTi execution.

Structural testing is important to assure some properties to the testing activity as the execution of critical parts of the code or the execution of all the branches in the program. In addition, it favors to find pitfalls in the program structure and logic [2]. Unfortunately, given the limited resources of the mobile devices cited before, it is not practical the use of a test environment as the one described in Figure 5 directly in the device. The restrictions of performance and space (memory and storage) make a tool like **JaBUTi** too complex to exist in the mobile device. On the other hand, it is important at a given moment in the testing process, to test the program on its real operation environment.

To address this problem, the solution described in the Figure 6 has been developed. It can be seen in the figure that the software is been executed in the mobile device and sends the trace data to an extended version of **JaBUTi** using a network connection. A test server is run on the desktop side and is responsible for receiving execution trace data and delivering it to the tool. The current version uses a simple socket connection for this purpose. This extended version of the tool is called **JaBUTi/ME**. In this way, it is possible to execute the test case on the real environment and still apply structural testing criteria with the support of the tool.

In order to adopt such a solution, it is necessary to sophisticate the instrumentation of the program under test. The instrumented program, besides collecting trace data as in ordinary Java applications, should be able to send such data to the testing tool. Fortunately, the Java runtime (J2ME) on mobile devices provides the ways to create a network connection and to delivery trace data from the device to the **JaBUTi/ME** tool. On the other hand, the variety of devices and configurations requires a greater flexibility when determining how the instrumentation should be done in order to optimize the processes of collecting and transmitting the trace data.

For instance, in a PDA (Personal Data Assistant) one may have a permanent link with the desktop without any extra cost and with a good speed through a wire connection or through wireless interfaces as Bluetooth or Wi-Fi. In this case, a connection to transmit trace data can be create when the application is initialized and all trace data can be sent as soon as they are collected. On the opposite extreme, one may have an application being tested in a cell phone communicating to the extern world through a paid connection on which the time connected is relevant. In this case, a better approach would be to collect as much data as possible and send them all at once. In this way, the connection would be active only when the data were ready to be sent. In addition, collecting a large amount

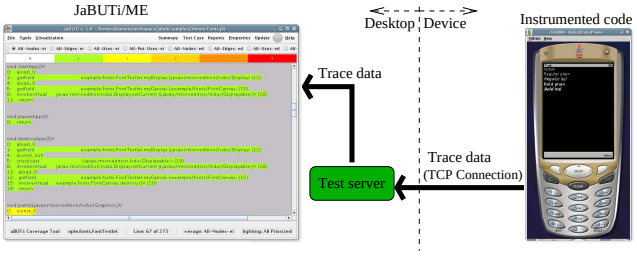


Figure 6: JaBUTi/ME coupled to a mobile device.

of data before sending it would save the program from creating several connections what could interfere in the normal program execution, since the creation of such a connection corresponds to make a phone call.

Taking in consideration the factors: type of communication link, available memory in the device and existence or not of temporary storage (file system) in the device, some parameters were defined and the tester can choose among them when instrumenting the program. The parameters are:

- **the address of the test server**. Determines the IP address and the port to which the connection to send trace data should be established.
- **identification name of the program being tested**. Because the test server can deal with several connections and several test sessions at the same time, such an identification allows the instrumented program to identify itself to the test server when sending trace data. In this way the server can decide what to do with the data it receives, i.e., can decide to which test session the data should be delivered.
- **name of the file used for temporary storage of trace data**. This parameter is optional. If provided, it instructs the code inserted in the instrumented program to store all the trace data until the end of the execution and then, send it at once to the test server. The data is stored in a temporary file in the mobile device's file system. The device must, of course, have support for this kind of resource. This parameter is useful when the device has support to a file system and a kind of communication link that demands a large delay to establish the connection or that has a high price of use and can not be kept open. If that is the case, the use of this parameter allows a connection to be created and kept open only during the transmission duration.
- **minimum amount of available memory**. With this parameter the tester can determine the amount of memory that could be used by the instrumented code to store trace data before deciding to send it to the test server. When the instrumented code collects new trace data and the amount of memory available at that moment is bellow the values established by this parameter, the trace data stored in the main memory is delivered, directly to the test server or to the temporary file (depending on the other parameters used in the instrumentation). Besides controlling the memory usage, this parameter may be useful to set other behaviors of the instrumented code. For instance, if the tester uses a very high value for this parameter, above the real amount of memory available in the device, this would make the data to be delivered immediately, and never stored in the main memory.

- **keep or not the connection**. The ordinary behavior of the instrumented code is to create a single connection with the test server at the beginning of the execution of the test case and keep it open until the program ends. In this way, each time trace data is available to delivery, the connection can be used without having to create a new one. For the cases when the cost of keeping the connection open is restrictive, this parameter can be used to instruct the instrumented code to create a connection each time it is needed, an to close it after its use. If the option to use a temporary file is provided, then this parameter is always false (the connection is not kept open), despite the tester choice because the trace data is transmitted only once, at the end of the execution.

The approach described in this section allows the tester to deal with the problems of structural testing on mobile devices, by given her support to a wide variety of devices and configurations. In the next section an example that illustrates the complete process of creating and running a **JaBUTi/ME** test session in a mobile device is presented.

4. A COMPLETE EXAMPLE

This example uses a set of programs (midlets) called "Demos", furnished with the Wireless Toolkit (WTK), the J2ME development environment provided by Sun Microsystems. These midlets exemplify several characteristics of the Java runtime for the mobile devices as, for instance, the use of fonts, threads, colors and network. All the midlets are packaged in a single deployment unit, i.e., a single *jar* file and its corresponding *jad* file.

Figure 7 summarizes the steps to follow to create a test session to the package "Demos". The tool allows the tester to select specific programs and classes to test, inside the given package. In this example, we use one of the midlets, named "FontTestlet", that is an example of the use of different kinds of fonts. In the sequence, each of the steps in the process described in Figure 7 is discussed in detail.

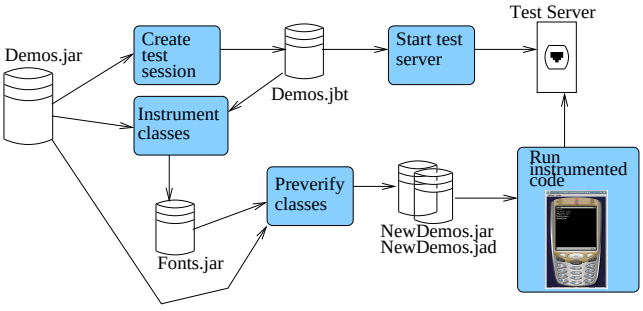


Figure 7: Steps to run a test session.

Create test session

The first step to test the midlet is to create a test session using **JaBUTi/ME**. The tool performs all the static analysis and computation of testing requirements on the Java bytecode of the original program. If the source code is available as well, the tool maps the requirements computed from the bytecode, back to the source code.

From the many classes found in the "Demos" package the tester selects the base class she wants to test, i.e., the class that implements the midlet to be tested. From that class the tool finds the set of all classes required to build the program and that will take part in the test session. A name is assigned to the test session, that corresponds to the XML file where the session features are described. In the example, the name is "Demos.jbt".

Start test server

Using the **JaBUTi/ME** graphical interface or a command line script, the tester starts the execution of the test server, which will listen to the specified port and receive the data about test case executions. To start the server, some configuration parameters must be provided. The first, is the number of the port to which the server will be attached. In addition, a list of the names of the midlets from which the server is supposed to received data and the names of the respective test sessions are also provided at the initialization. In this way, when a given executing midlet creates a connection with the test server, it identifies itself using its name and the server knows which test session the data belongs to.

In the current example, the server is informed that the name "Fonts" will be used to identify a midlet and the trace data produced by such program should be made available to the test session named "Demos.jbt". The name of the midlet is given at the program instrumentation, as described next. The test server uses a simple socket connection to receive data. Thus, it is due to the midlet to start and to control the connection. As described next, this is appropriate because it allows the application to decide when to send data to the test server as, for instance, when available memory to store trace data is low.

Instrument classes

In this step the tester chooses the parameters and an instrumented version of the original code is created. As shown in Figure 7, the instrumented classes are stored in a *jar* file and in the sequence are preverified. The instrumentation is performed by a stand alone program, apart from the **JaBUTi/ME** graphical interface. By executing such a program the tester can define all the arguments, as described in Section 3. For instance, in the current example we suppose the tester wants to test the midlet "FontTestlet" in a PDA with a filesystem, connected directly to the desktop via an wireless network interface, thus she can issue the statement shown in the top of Figure 8.

```
> java br.jabuti.device.ProberInstrum \
        -p Demos.jbt -name Fonts -o Fonts.jar \
        -h 192.180.30.250:1988 \
        -temp /internal/temp/trace.txt \
        example.fonts.FontTestlet
------------------------------
Files to insert:
example/fonts/FontCanvas.class
example/fonts/FontTestlet.class
br/jabuti/device/j2me/DeviceProber$1.class
br/jabuti/device/j2me/ProbedNode.class
br/jabuti/device/j2me/DeviceProber.class
------------------------------
```

Figure 8: Statement used to instrument the program.

In the statement, the name of the test session is given (*-p Demos.jbt*), as well as the name to be assigned to the midlet (*-name Fonts*) and the name of the output file, where to store the instrumented classes (*-o Fonts.jar*). Next, the address of the destination test server, the name of the temporary file to store trace data in the device and the name of the midlet class are provided. The result of this statement is the creation of file "Fonts.jar" that contains the classes listed in the bottom part of Figure 8. It is important to note

that, in addition to those classes that are part of the original midlet, three other classes are included in the package. These classes are responsible for the collection, storage and delivery of the trace data and now are also part of the midlet.

Preverify classes

At last, before executing the instrumented code and collecting trace data, the classes just instrumented must be preverified. This is an ordinary task on J2ME software development. The Java code that runs in the J2ME environment has to respect a few restriction. This step takes care of verifying whether these restrictions are respected and modifying the code, if necessary, to make it compliant to the J2ME runtime. In addition, in this step the final package that is ready to be transferred to the device is created. As seen in Figure 7, such a package is composed by a *jar* file that is a copy of the original "Demos.jar" with some of the original classes replaced by the instrumented ones and with the addition of the new classes, inserted by the instrumentation process and a *jad* file that is a plain text file that simply describes the content of the former.

Run instrumented code

At this point the program can be transferred to the device and executed. It is necessary to have the test server running and ready to receive the trace data. The execution of the instrumented program is expected to be exactly the same of the original one, except that in some devices the runtime explicitly asks the user for authorization to use the network functionality and filesystem access that may have been added to the program during its instrumentation.

At the end of the execution of the instrumented midlet, the trace data are transferred to the test server along with the identification of the name of the midlet. The server makes the data available to the corresponding test session, what characterizes a new test case. The tool analyzes the test case data and updates the coverage information and the tester can evaluate the test session progress.

5. CONCLUSION

This paper presented a strategy that aims to contribute to the improvement of the testing activity of Java programs developed for mobile devices. Initially, the difficulties imposed by the limitations of such devices were discussed. In particular, the restrictions of performance and storage greatly difficult the use of structural testing criteria in this context.

To overcome these problems and to permit testing the software on its real operational environment with the aid of structural criteria, we proposed a test environment composed by a supporting tool that allows the tester to perform all the activities concerning structural testing in a desktop computer and collect the trace data of a test case execution from the mobile device. An instrumentation scheme that, besides collecting trace data during test execution, also makes possible to deliver such data to the testing tool in the desktop, has been created.

In addition, due to the inherent restrictions and many different possible configurations of the target environment, in terms of hardware and software, it is extremely necessary to use a flexible instrumentation framework that allows the tester to adequate and configure it to her testing environment. Features of the target device like memory size, availability of a file system and type of network communication link should be taken in consideration to improve trace data collection and delivery.

In this same line of research, the authors are exploring some other factors that could improve even more the presented strategy. One of them is the study of the consequences of applying structural testing criteria on obfuscated code. The goal is to evaluate the results of applying such criteria on obfuscated code, comparing them with the same data collected on the original (non-obfuscated) code. This might lead to an strategy that indicates how to test the original and/or the obfuscated code. Another topic of interest is the development of approaches to reduce the amount of information to be collected, necessary to the assessment of structural requirement coverage. The current version of **JaBUTi/ME** uses the same approach to collect and store execution trace information, what may represent a large amount of data and be infeasible on some kinds of devices. Such a reduction could enhance the performance during the test case execution since the trace data, in the described testing environment, has to be made available to the **JaBUTi/ME** through a network link, what may become inefficient in the presence of a slow connection.

6. REFERENCES

[1] H. Agrawal, J. Alberi, J. R. Horgan, J. Li, S. London, W. E. Wong, S. Ghosh, and N. Wilde. Mining system tests to aid software maintenance. *IEEE Computer*, 31(7):64–73, July 1998.

[2] B. Beizer. *Software Testing Techniques*. Van Nostrand Reinhold Company, New York, 2nd edition, 1990.

[3] M. Cundy. Testing mobile application is different from testing traditional applications. VeriTest testers' network – on-line article, July/Aug. 2001. Available at: , accessed on:.

[4] M. E. Delamaro and A. M. R. Vincenzi. Structural Testing of Mobile Agents. In E. A. Nicolas Guelfi and G. Reggio, editors, *III International Workshop on Scientific Engineering of Java Distributed Applications (FIDJI'2003)*, Lecture Notes on Computer Science, pages 73–85. Springer, Nov. 2003.

[5] T. Edge. Glass JAR toolkit. WEB Page, 2002. Available at: `http://www.testersedge.com/glass.htm`. Accessed on: 11/15/2005.

[6] P. G. Frankl and E. J. Weyuker. An applicable family of data flow testing criteria. *IEEE Transactions on Software Engineering*, 14(10):1483–1498, Oct. 1988.

[7] Q. H. Mahmoud. Testing wireless java applications. Sun Microsystems – on-line article, Nov. 2002. Available at: `http://developers.sun.com/techtopics/mobility/midp/articles/test/`, accessed on: 10/12/2005.

[8] J. C. Maldonado. *Potential-Uses Criteria: A Contribution to the Structural Testing of Software*. Doctoral dissertation, DCA/FEE/UNICAMP, Campinas, SP, Brazil, July 1991. (in Portuguese).

[9] J. W. Muchow. *Core J2ME Technology and MIDP*. Prentice Hall, 2001.

[10] I. Satoh. A testing framework for mobile computing software. *IEEE Transactions on Software Engineering*, 29(12):1112–1121, Dec. 2003.

[11] I. Satoh. Software testing for wireless mobile application. *IEEE Wireless Communications*, pages 58–64, Oct. 2004.

[12] SourceForge. J2ME unit testing framework. WEB Page, 2002. Available at: `http://j2meunit.sourceforge.net/`. Accessed on: 11/15/2005.

[13] A. M. R. Vincenzi, M. E. Delamaro, W. E. Wong, and J. C. Maldonado. Establishing structural testing criteria for Java bytecode. *Software Practice and Experience*, 2006. (to appear).

[14] A. M. R. Vincenzi, J. C. Maldonado, W. E. Wong, and M. E. Delamaro. Coverage testing of Java programs and components. *Journal of Science of Computer Programming*, 56(1-2):211–230, Apr. 2005.

[15] A. M. R. Vincenzi, W. E. Wong, M. E. Delamaro, and J. C. Maldonado. JaBUTi: A coverage analysis tool for Java programs. In *XVII SBES – Brazilian Symposium on Software Engineering*, pages 79–84, Manaus, AM, Brazil, Oct. 2003.