Evaluating Structural Criteria for Testing Mobile Code

Márcio E. Delamaro
Centro Universitário Eurípides de Marília (UNIVEM)
Marília, São Paulo, Brazil
marcio.delamaro@pq.cnpq.br

Auri Marcelo Rizzo Vincenzi
Universidade Federal de Goiás (UFG)
Goiânia, Goiás, Brazil
auri@inf.ufg.br

José Carlos Maldonado
Universidade de São Paulo (USP)
São Carlos, São Paulo, Brazil
jcmaldon@icmc.usp.br

Abstract

Code mobility has become an attractive alternative to the implementation of distributed systems. In the last few years much effort has been made in the study and development of techniques to support the development of software based on mobile code. In spite of this effort, there is still a lack of criteria and environments to test mobile software. In this paper we investigate the use of control and data flow based criteria for testing software with mobility features. A testing tool named JaBUTi/MA supports coverage analysis based on client triggered and server triggered instrumentation. A case study is presented to assess the use of JaBUTi/MA on mobile agents.

Keywords: Software testing, software testing tool, mobile agent

1. INTRODUCTION

Testing is one of the most important elements amongst software validation and verification activities. Complementing other techniques as technical revision and formal methods, its objective is to enhance productivity and provide evidence of the reliability and quality of the product. In addition, testing artifacts can be valuable information to other software engineering tasks like debugging, maintenance and reliability assessment. On the other hand, it can take more than 50% of the software development total cost [3].

Testing is conducted by executing the software with selected input data and comparing the results against a given specification, if available. A critical factor in this process is the quality of the test set used to execute the software. Its selection should be done based on testing requirements that establish which aspects of the software must be exercised. Such requirements can be derived from the specification or from the code under test.

According to Harrold [3], the advantages of testing over other forms of software analysis are: 1) relative easy testing activities; 2) software can be executed in its expected environment; and 3) much of the testing process can be automated. In order to have such advantages it is essential to make use of tools to support the testing techniques. Testing activities can be expensive and error-prone if done manually. Supporting tools aim at reducing the risk of human mistakes in such activities, improving testing productivity. In addition, they are a way to facilitate the understanding of testing concepts at the academia and, according to Pfleeger [8], an important factor to facilitate technology transfer to the industry.

New technologies, on one hand, may bring advances to the software development field by rising productivity, lowering costs, improving quality, promoting scalability etc. On the other hand, they may pose new challenges in the testing field. For instance, object oriented development that has been a big break through in software development, has also been a challenge for testing researchers, as pointed by Vincenzi et al. [13]. To face these challenges, based on the experience and knowledge accumulated through the years, existing techniques for testing conventional software can be adapted to new approaches, like object-oriented development [7]. This is the approach taken in this paper, exploring previous experience and knowledge on control and data-flow based testing in the context of mobile agent-based software.

Code mobility is defined as “the capability to reconfigure dynamically, at run-time the binding between the software components of the application and their physical location within a computer network” [1]. The mobile agent is an alternative approach to the implementation of distributed systems. Using code mobility, it promotes features like reduced bandwidth consumption and enhanced flexibility [10, 6]. Much of the research on code mobility has been concentrated on its technological aspects. More specifically, several languages and environments to support the development of code mobility applications have appeared in the last few years. Karnik and Tripathi [6] and Fuggetta et al. [2] have identified a dozen of them.

As a new technology, code mobility brings new problems – that have not been addressed so far – for the testing area. For example, its distributed and itinerant execution makes it more difficult to collect execution data, essential to the application of white-box testing criteria. This paper proposes the use of structural testing on mobile agents. It discusses the difficulties of doing so and presents a tool, named **JaBUTi/MA**, that supports the application of control-flow and data-flow adequacy criteria to mobile agents.

In the next section, a review on mobile agents and μ CODE – an API to support code mobility in Java – is presented. In Section 3 our approach to use structural testing on mobile agents and **JaBUTi/MA**, a tool to support structural testing of mobile agents are discussed. A case study, showing the application of **JaBUTi/MA** to a mobile agent program is presented in Section 4. In Section 5 the final remarks are presented.

2. MOBILE AGENTS

In the literature, several different definitions for the mobile agent concept can be found. In this paper a mobile agent is considered a piece of code that can choose its execution location in a network without external intervention or, as defined by Karnik and Tripathi [6], a mobile agent is “a program that represents a user in a computer network and can migrate autonomously from node to node to perform some computation on behalf of the user”.

Like any other program, how to implement a mobile agent might depend on several factors like language, operating system, and support platform. This paper focuses on Java mobile agents and uses μ CODE [9] as the platform to support code mobility. However, the concepts behind the current approach can be extended to other platforms that support the same kind of abstraction, i.e., the mobile agent abstraction.

μ CODE was chosen to be used in this work because it is a simple and lightweight Java API that supports mobility of Java classes and objects. It supports not just the development of programs based on the mobile agent approach. In fact, the API emphasizes flexibility by providing a series of basic classes and primitive mobility operations on top of which programmers can construct their own abstractions.

μ CODE provides a few abstractions on its own, including primitives to remotely clone and spawn threads, ship and fetch classes to and from a remote μ Server, and a full-fledged implementation of the mobile agent concept. This is the model adopted in this work. It is a very simple model and easy to implement.

A mobile agent in this context is a thread that has the following functionality regarding mobility: 1) it can interrupt and ship itself to another host; and 2) at destination it is restarted from the beginning. An abstract class `MuAgent` is provided as a subclass of `java.class.Thread`. Besides the abstract method `run` the class implements a method `go(String destination)` that stops the thread, creates an object to pack all the necessary classes and ships it to the destination host.

It is very easy to implement a mobile agent using μ CODE. All the programmer has to do is to subclass `MuAgent`, providing a concrete implementation of method `run`. This method is the “body” of the agent and is used to start the agent at its original location and to restart it wherever it migrates to. The method `go` in the `MuAgent` class can be used when the agent needs to be relocated.

3. TESTING MOBILE AGENTS

JaBUTi (Java Bytecode Understanding and Testing Tool) [14, 12] intends to be a complete toolsuite for understanding and testing Java programs and Java-based components. **JaBUTi** can be used to perform coverage analysis using different testing criteria, and to localize faults using slicing heuristics. It also collects complexity metrics information based on the bytecode. The focus of this section is to describe how to use **JaBUTi** as a coverage analysis tool.

JaBUTi implements four intra-method control-flow based testing criteria and four intra-method data-flow based testing criteria. The main activities executed by **JaBUTi** to perform the coverage analysis are: to instrument class files, to collect the dynamic execution trace during the classes execution, and to determine how well the classes have been tested according to a given testing criterion. Different testing reports can also be generated to analyze the coverage with respect to each criterion, each class, each method, and each test case.

The use of structural testing requires a series of tasks like program analysis for testing requirement computation and coverage analysis that are not trivial. The tasks **JaBUTi** performs are:

- Static analysis – the program is parsed and the control- and data-flow information is abstracted in the form of def-use graphs. Other information, as for instance, call graphs, inter-method control-flow graphs and data-flow data can also be gathered.
- Requirement computation – based on the information collected in the task above, the tool computes the set of testing requirements. Such requirements can be a set of nodes, edges or def-use associations.
- Instrumentation – in order to measure coverage, i.e., to know which requirements have been exercised by the test cases, it is necessary to know which pieces of the code have been executed. The most common and probably the simplest way to do so is by instrumenting the original code. Instrumentation consists of inserting extra code in the original program, in such a way that the instrumented program produces the same results of the original program and, in addition, produces a list of “traces”, reporting the program execution.
- Execution of the instrumented code – a test is performed using the instrumented code, instead of the original program. If it behaves inappropriately, a fault is detected. Otherwise, a trace report is generated and the quality of the test set can be assessed based on such a report.
- Coverage analysis – confronting the testing requirements and the paths executed by the program, the tool can compute how many of the requirements have been covered. It can also give an indication of how to improve the test set by showing which requirements have not been satisfied.

What differentiates **JaBUTi** from other tools is that its static analysis and instrumentation processes are not done in the Java source code, but in the object code (Java bytecode). In fact, the tester does not even need the source code to test a program, although, because the Java bytecode retains information about source code, if the source is available, the tool maps the data about the testing requirements and coverage back to the source program.

JaBUTi has two main parts. The first, referred as “analysis tool”, does all the analysis, requirement computation and test case evaluation. The second is a test driver that instruments the classes, and loads and executes the instrumented program.

By instrumenting the code, the test driver “attaches” new code to the existing classes and new classes to the program, i.e., a “prober” class, which collects trace data and writes it to a file, becomes part of the application and is necessary to make the application run. The analysis tool constantly polls the trace file. When a new test case is inserted in the trace file, the tool updates the test information, recomputing the coverage.

Under the perspective of structural testing, it is harder to test a mobile agent application than a conventional program. Difficulty relates, mostly, to the fact that collecting execution information in a conventional program is as simple as instrumenting the program under test to write trace data to a local file. This is the approach used by **JaBUTi** as shown in Figure 1(a). Testing mobile agents, on the other hand, requires a way to collect trace data from several different points, i.e., from every place in the network where the agent executes, and to deliver it to the analysis tool, as shown in Figure 1(b).

FIGURE 1: Collecting trace data for structural testing.

Note that this is not just a matter of changing from a local trace file to a remote one using conventional communication schemes like sockets or database connections. When dealing with mobile agents, it is very admissible or even probable to consider an heterogeneous environment where hosts differ in terms of both, software and hardware. Computers in a network may have different operating systems, different memory and CPU configurations or present more radical differences like desktop computers, laptops, PDAs or cell phones, all co-existing. In such an environment one cannot presuppose the availability of those communication mechanisms. Even if a more “universal” mechanism is chosen, like sockets, for instance, it is not always warranted that a mobile agent has access to it on the hosts where it travels. Security concerns may prevent the agent from accessing that resource.

Therefore, the goal of this work was to develop an approach to apply structural testing on mobile agents, based on **JaBUTi**, with the following characteristics:

- the tester should be able to collect information from a mobile agent execution in its “real environment”, not by simulating its execution locally;
- wherever the agent could execute, it should be able to collect trace data, independently of the platform;

- the existing testing structure implemented in **JaBUTi** should be reused, without major changes, to perform the coverage analysis.

The extension of **JaBUTi** to deal with mobile agents, named **JaBUTi/MA**, has those features. The idea is to use the mobility structure furnished by μ CODE to support the delivery of trace data from the mobile agent under test (*MAUT*) to the analysis tool. Since it is guaranteed that μ CODE is present in every single host the agent visits, the resources necessary to make the *MAUT* communicate with **JaBUTi** are present too. Moreover, the security issues are also minimized because the only operation the *MAUT* should be able to perform is to create new agents to transmit trace data.

Together with the analysis tool, there is a test μ Server ($T_{\mu}S$) that receives trace data from a mobile agent and writes it to a trace file. The analysis tool is exactly the same as in the original **JaBUTi**. All it does is to poll the file, waiting for new data to arrive. The changes in the tool were restricted to the instrumentation portion. The instrumentation “piggybacks” a prober class to the *MAUT* that keeps track of all the blocks of code executed in the agent, exactly as it is done for conventional programs. In addition, the prober class stores information about where to deliver trace information, i.e., the $T_{\mu}S$ address.

When it is the correct time, i.e., at migration or termination, the prober class creates a new test agent (*TA*) to deliver the trace information. The prober class creates a *TA* and sends it to the $T_{\mu}S$. Note that the prober class must migrate together with the *MAUT*, along with the information about the $T_{\mu}S$. In this way, when the agent is restarted, so is the process of collecting trace data.

To promote flexibility, **JaBUTi/MA** provides two different ways to test an agent. The first is instrumenting the agent and then starting it already instrumented. So, the execution of the *MAUT* and the collection of trace data are triggered by the agent user (the client). The second is instrumenting the agent when it arrives at a given host, i.e., the instrumentation and trace data collection are started by a μ Server. In this case, the tester can also set the μ Server to forward the original code or the instrumented code. With these possibilities the tester can choose in which hosts she/he wants to collect trace data.

4. A CASE STUDY

In this section we describe the use of **JaBUTi/MA** to test a mobile agent-based application.

4.1. Goal

The goal of this study is to assess the use of the tool by measuring two factors:

G1: the improvement it may provided in the quality of a test set; and

G2: the extra cost it represents.

4.2. Subjects(Players)

The subjects were programmers, being by two undergraduate students and two master students. They had similar skills and some experience on Java programming, i.e., approximately two years of scholar projects. None of them had previous experience on developing or testing “real” software projects. These students were arranged in three teams:

Team 1: two undergraduate students;

Team 2: one master student;

Team 3: one master student.

4.3. Materials

4.3.1. Object of study.

We asked each of the three teams to develop a mobile agent that plays the Yahtzee game¹. This is a dice game where the player rolls five dice to try to fill one of the 13 spots in a score table. Each spot grants a certain number of points. The amount of points earned for the spots in some cases is fixed and in some cases depends on the dice configuration. For example, the *full-hand* spot always grants 15 points, as far as the dice configuration presents a triplet and a pair. The “sixes” spot grants the number of dice with a six in the final configuration, times six. The player has three chances in each round. In the first, all five dice are thrown. In the next two the player can select which dice to keep and which to re-throw. Thus, the strategy of the game is to choose correctly which dice to keep. Once a spot is filled it cannot be used again. So, the player’s strategy should also incorporate the current state of the score table. At the end of 13 rounds, all the spots are filled and the points earned in each one are summed to give the final score.

4.3.2. Testing techniques and tools

Two testing techniques were used in the study. Initially functional testing was applied manually, without tool support. Then, data- and control-flow (structural) criteria were applied with the support of **JaBUTi/MA**.

4.4. Variables

In order to assess **G1** and **G2**, the following dependent variables were taken in consideration:

1. number of test cases produced by functional testing;
2. number of structural testing requirements;
3. structural coverage obtained by functional test set;
4. number of additional effective test cases required by structural criteria;
5. total number of additional test cases required by structural criteria.

4.5. Procedure

Next we describe the steps taken on conducting the case study.

4.5.1. Design

The study was divided in five steps:

1. elaboration of a strategy for playing Yahtzee;
2. construction of the Yahtzee-mobile agent, based on the strategy;
3. testing the agent using functional testing;
4. evaluating functional test set against structural requirements; and
5. complementing the test set in order to obtain adequate structural coverage.

4.5.2. Training

The students involved in the study received formal training on functional and structural testing techniques during their regular courses, which included approximately four hours on studying and using **JaBUTi**. In addition, approximately another four hours were spent on practicing with **JaBUTi/MA**. In these practice sessions, sample agents were provided to the students who could try using **JaBUTi/MA** features with the support of a tutorial and of one of the tool developer.

¹In Brazil, also known as **Bozó** or **General** game.

4.5.3. Applying the steps

Step 1: The strategy to play Yahtzee may be based on several different factors, e.g., the current state of the score table, the configuration of the dice and the probability of obtaining a given configuration. We asked the three teams to describe their strategies in terms of the rules they used to decide how to proceed in the game. Rules like:

- if there are five dice with the same number, choose Yahtzee; or
- if there are two pairs and full-hand is available, try full-hand otherwise keep the higher pair and roll the other dice.

Step 2: Each team used its strategy to implement an agent. The agent should be launched, go to a given μ Server and ask for an object that represents a new game. This object is responsible for managing the game, providing the dice, controlling the number of tries in each round, the number of rounds to play and the assignment of scores in each spot of the score table. The authors of the current paper developed the server and published its interface, so the Yahtzee-agents could use it. When a game is over, the agent should move back to its original location and display the score as well as all the rounds and choices it made.

Step 3: The teams were asked to validate their strategies using functional testing, i.e., to execute the agent and check, by the sequence of choices done by the program, which rules had been verified. Verified meant that the tester had been able to see that either the rule had been correctly implemented or had not been correctly implemented. A test case that verified at least one rule was considered “effective”.

Step 4: The teams were instructed to use **JaBUTi/MA** to evaluate this functional test set against structural requirements. They took the effective test cases and evaluated them using two control-flow and one data-flow based criteria. With that, one can tell if and how **JaBUTi/MA** can be usefull, i.e., can compare a testing strategy with the use of the tool against one without using it.

Step 5: The testers tried, by executing additional test cases, to cover the missing structural requirements. The teams inserted new test cases on the previuos test sets until they reached a requirement coverage level they judged adequate. In this case, “adequate” meant that improving the coverage would require too much effort because a combination of dice too hard to obtain would be necessary.

4.5.4. Data collection

The tables below show the data collected in order to assess the goals previously established.

Table 1 shows a summary of the effort required to cover the rules for each program. The second column shows the number of rules extracted from each implementation and the third column shows the number of test cases necessary to cover all the rules. The fourth shows the number of executions necessary to cover the rules. In all the cases the testers found out that there were rules too hard to cover. They represent situations so improbable that it might be necessary an excessive number of executions to cover them, thus, they were left uncovered.

TABLE 1: Summary of functional testing

Version	# of rules	# of effective test cases	total # of test cases	# of rules verified	% of rules verified
Team 1	25	5	7	20	80%
Team 2	22	4	9	16	73%
Team 3	42	12	15	37	88%
Average	29.7	7	10.3	24.3	80.3%

Table 2 shows the number of requirements covered by this first test set created using only the functional rules, the total number o requirements and the percentage relative to these two

numbers for each of the three criteria: All-Nodes; All-Edges; and All-Uses. In addition, the size of the classes, measured in terms of the number of bytecode instructions, is reported in the first column. This table gives the idea of the cost of using the tool (number of requirements) and the adequacy of the initial test set (percentage of covered requirements).

TABLE 2: Structural requirements covered by functional test sets

Version	Size	All-nodes	All-Edges	All-uses
Team 1	1213	125 / 134 (93%)	181 / 202 (89%)	433 / 675 (64%)
Team 2	1341	167 / 181 (92%)	214 / 251 (85%)	457 / 583 (81%)
Team 3	2372	267 / 291(91%)	366 / 416 (87%)	925 / 1145 (80%)
Average	1642	186 / 202 (92%)	253 / 289 87%	605 / 801 (75%)

Table 3 shows the result of running additional test cases with the aid of **JaBUTi/MA** to assess the structural coverage. An effective test case is one that covers at least one structural requirement. In average, 22.7 new test cases were run (third column) and eight of them were effective on covering structural requirements. The code coverage (All-Nodes) had a small improvement, going from 92% to 94.3% in average. The highest improvement occurred for the team 2, with 10 new nodes covered. For the stronger criteria, the coverage improvement was a little higher. For instance, the Team 1 managed to cover 55 new data-flow associations, going from 64% to 72% of def-use coverage.

TABLE 3: Improvement in structural coverage

Version	# of effective test cases	total # of test cases	All-nodes	All-Edges	All-uses
Team 1	16	33	127 (94%)	186 (92%)	488 (72%)
Team 2	3	24	177 (97%)	235 (93%)	504 (89%)
Team 3	5	11	269 (92%)	372 (87%)	953 (83%)
Average	8	22.7	94.3	90.7	81.3

4.5.5. Results

It is well known that functional and structural testing are complementary techniques. They address different kinds of faults and so should be used in conjunction. In addition, functional testing may lack precision when a formal specification is not available to help derive testing requirements. If that is the case, the test requirement computation may be highly influenced by factors like the tester experience. Structural testing, on the other hand, provides precise information about testing completeness, based on very precise concepts as node, edges or def-use associations.

With respect to factor **G1**, with the data obtained in this case study, we could see that the same can be considered true when we deal of mobile code. The need for a tool like **JaBUTi/MA**, the only one we know developed specifically for this kind of programs, was assessed by showing that test sets developed using functional testing can be sensibly improved by the use of structural criteria, in the three mobile agents used as study objects.

The data also provide a cost measurement (factor **G2**). The number of structural requirements, the final number of effective test cases and the number of executions required to obtain the test set gives an indication of the cost. The numbers show that the cost of using **JaBUTi/MA** and structural testing can be considered manageable and not prohibitive.

In addition to the improvement obtained in the test set by using structural testing, there is also the subjective but still important fact that testers preferred structural over functional testing. The testers who participated in this experiment were unanimous in agreeing that using structural criteria was much more comfortable than using functional testing. They are all students with little experience in software testing, and felt more secure on having a tool computing the requirements than having themselves computing and trying to cover the (functional) requirements, making the testing less dependent on their tester skills.

5. CONCLUSION

Testing is an essential activity in the software lifecycle. A lot of effort has been spent on the development of testing techniques and tools. Several issues are still open in this area and the adoption of new technologies introduces new ones. Distributed systems, in particular in the form of mobile agents, bring new challenges to the testing research area.

Only the use of systematic and theoretically sound approaches may provide quality test sets. That is the reason why testing criteria have been widely studied in the last decades. Amongst them, control- and data-flow are some of the best known criteria. Initially proposed to procedural units, the technique has been extended in several different directions, e.g., on integration testing by [5], on object-oriented testing by [4] and on concurrent program testing by [11].

Mobile code is an alternative to conventional distributed systems which, according to the literature, may present advantages for some classes of applications. Much of the effort in this area has been spent in technological aspects such as the development of environment and languages to support code mobility

This paper presented an approach and a tool named **JaBUTi/MA** to support the application of structural testing criteria to applications based on the mobile agent paradigm. Code mobility introduces new challenges to the test activity. In particular, the use of structural criteria faces the serious problem of execution data collection, as the agent moves through the nodes of a network.

In a case study, the **JaBUTi/MA** was used as an alternative to test an agent that plays the Yahtzee game. The goal was to show that such a tool is useful also in the domain of mobile code applications. We believe this is a good example of a mobile agent that shows why it is important to test the agent in its real environment. The goal of testing this agent is to exercise the program and check whether or not the game strategy has been correctly implemented. However, test cases to do so are not provided by the tester. It is provided by the server. In this dice game, the tester could spend a little extra implementation effort and create its own Yahtzee-server to test his/her agent, since the behavior of an honest server is well known. But in cases where the behavior of the test case provider (the server) is not completely known, there is no other option besides the use of the actual server(s).

In the case of the Yahtzee-agent or other systems where the server is the test provider, the tester cannot decide which test cases to use. This is not comfortable, because he/she is tied to the chances of having a good test set produced by the server. On the other hand, in this scenario it is very useful to have a measure of the quality of the test set, supported, for instance, by a structural criterion. Such a criterion can be used to access the level of coverage obtained and as an indicator when to stop testing.

Three versions of such agent were developed by students and tested, initially without the aid of the tool and then with the support of the tool. It could be shown that the use of the tool can improve test sets initially developed using functional testing only. In addition, the cost of doing so is perfectly manageable and is not prohibitive. In addition, we could verify that the use of **JaBUTi/MA** to guide the testing process provided more confidence to the students, with low experience in the software testing activity.

As far as the authors are aware, this is a innovative work and no other environment to aid test case evaluation for mobile agents have been developed so far. Comparing functional and structural testing turned it possible to show that the use of the tool may lead to improved test sets and can provide the necessary assessment to the tester to decide whether code has been sufficiently exercised. In addition, the testers involved in the study could attest the importance of a tool like **JaBUTi/MA** to generate and verify test requirements. The use of structural criteria, in contrast to functional criteria, can ease the testing activity by automatizing the process and making it less dependent on the tester experience.

REFERENCES

- [1] A. Carzaniga, G.P. Picco, and G. Vigna. Designing Distributed Applications with Mobile Code Paradigms. In *Proceedings of the 19th International Conference on Software Engineering (ICSE'97)*, pages 22–32, Boston - MA (USA), May 1997. ACM Press.
- [2] A. Fuggetta, G.P. Picco, and G. Vigna. Understanding Code Mobility. *IEEE Transactions on Software Engineering*, 24(5):342–361, May 1998.
- [3] M. J. Harrold. Testing: A Roadmap. In *International Conference on Software Engineering - Future of Software Engineering Track*, pages 61–72, Limerick - Ireland, 2000.
- [4] M. J. Harrold and G. Rothermel. Performing Data Flow Testing on Classes. In *Second ACM SIGSOFT Symposium on Foundations of Software Engineering*, pages 154–163, New York, NY, December 1994. ACM Press.
- [5] M. J. Harrold and M. L. Soffa. Selecting and Using Data for Integration Test. *IEEE Software*, 8(2):58–65, March 1991.
- [6] N. M. Karnik and A. R. Tripathi. Design Issues in Mobile-Agent Programming Systems. *IEEE Concurrency*, 6(3):52–61, July-September 1998.
- [7] A. J. Offutt and A. Irvine. Testing object-oriented software using the category-partition method. In *17th International Conference on Technology of Object-Oriented Languages and Systems*, pages 293–304, Santa Barbara, CA, August 1995. Prentice-Hall.
- [8] S. L. Pfleeger. Understanding and improving technology transfer in software engineering. *Journal of Systems and Software*, 47(2-3):111–124, July 1999.
- [9] G.P. Picco. μ CODE: A Lightweight and Flexible Mobile Code Toolkit. In *Proc. of Mobile Agents: 2nd Int. Workshop MA'98*, volume 1477 of *LNCIS*, pages 160–171. Springer, September 1998.
- [10] G.P. Picco. Mobile Agents: An Introduction. *Journal of Microprocessors and Microsystems*, 25(2):65–74, April 2001.
- [11] R.N. Taylor, D.L. Levine, and C.D. Kelly. Structural testing of concurrent programs. *IEEE Transactions on Software Engineering*, 18(3):206–215, March 1992.
- [12] A. M. R. Vincenzi, M. E. Delamaro, J. C. Maldonado, and W. E. Wong. Establishing structural testing criteria for Java bytecode. *Software Practice and Experience*, 36(14):1513–1541, November 2006.
- [13] A. M. R. Vincenzi, J. C. Maldonado, M. E. Delamaro, E. S. Spoto, and W. E. Wong. *Component-Based Software Quality: Methods and Techniques*, volume 2693 of *Lecture Notes in Computer Science*, chapter Component-Based Software: An Overview of Testing, pages 99–127. Springer-Verlag, New York, NY, June 2003.
- [14] 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 (Tool Section)*, pages 79–84, Manaus, AM, Brazil, October 2003.