Federal University of Pernambuco
Center of Informatics

Post Graduation Program in Computer Science

A Black-box Testing Technique for the Detection of Crashes Based on Automated Test Scenarios

Glaucia Boudoux Peres

MASTER THESIS

Recife
December, 2009

Federal University of Pernambuco
Center of Informatics

Glaucia Boudoux Peres

A Black-box Testing Technique for the Detection of Crashes Based on Automated Test Scenarios

Thesis submitted to the Post Graduation Program in Computer Science of the Center of Informatics of the Federal University of Pernambuco in partial fulfilment of the requirements for the degree of Master in Computer Science.

Advisor: *Alexandre Cabral Mota*
Co-advisor: *Marcelo Bezerra d'Amorim*

Recife
December, 2009

Peres, Glaucia Boudoux
A black-box testing technique for the detection of crashes based on automated test scenarios / Glaucia Boudoux Peres. - Recife: O Autor, 2009.
iv, 60 folhas : il., fig., tab.

Dissertação (mestrado) – Universidade Federal de Pernambuco. CIn. Ciência da Computação, 2009.

Inclui bibliografia.

1. Ciência da computação. 2. Engenharia de software. I. Título.

004 CDD (22. ed.) MEI2010 - 007

Dissertação de Mestrado apresentada por **Glaucia Boudoux Peres** à Pós-Graduação em Ciência da Computação do Centro de Informática da Universidade Federal de Pernambuco, sob o título "**A Black-box Testing Technique for the Detection of Crashes Based on Automated Test Scenarios**", orientada pelo **Prof. Alexandre Cabral Mota** e aprovada pela Banca Examinadora formada pelos professores:

Prof. Alexandre Cabral Mota
Centro de Informática / UFPE

Prof. Márcio Lopes Cornélio
Departamento de Sistemas Computacionais

Prof. Juliano Manabu Iyoda
Centro de Informática / UFPE

Visto e permitida a impressão.
Recife, 20 de fevereiro de 2009.

Prof. FRANCISCO DE ASSIS TENÓRIO DE CARVALHO
Coordenador da Pós-Graduação em Ciência da Computação do
Centro de Informática da Universidade Federal de Pernambuco.

To my parents.

Acknowledgements

First and foremost, I would like to thank God for being always by my side and for guiding me throughout this work.

I am vastly grateful to my parents, Gaspar and Conceição, to whom I dedicate this work. I also thank my sister, “Mana”, and my brothers, Valmir and “Budu”, for the encouragement, comprehension, and friendship they have always given to me. Without the support of my family I would not have come this far.

I am also thankful to a special person, Marcos, who has become my fiancé a few months ago and with whom I want to spend the next fourteen billion years. I want to thank him for the support, encouragement, and reviews he dedicated to this work, and for always being so certain about my achievements.

I also want to thank all my friends for being always there for me, and for understanding my moments of absence.

Many thanks go to the entire Research Project group for all the support, criticisms and suggestions during the development of this work. Special thanks to my research partner, Cristiano, for being always ready to help me, and with whom I shared the results of this work.

I do sincerely wish to express gratitude to my advisor, Alexandre, and my co-advisor, Marcelo, for trusting and pushing me throughout this work.

And I would also like to thank the research cooperation between Motorola and CIn/UFPE, which provided a pleasant environment for the conduction of experiments, as well as financial support.

If all your tests pass, chances are that your tests are not good enough.

—ALBERTO SAVOIA (Beautiful Tests)

Resumo

Apesar dos avanços tecnológicos das linguagens e ferramentas que dão suporte ao desenvolvimento de sistemas, programadores ainda entregam software contendo erros. Inúmeras técnicas foram propostas a fim de aumentar a confiança nos softwares que são postos no mercado. Teste de software é uma delas. Na verdade, testar é a atividade dominante na indústria para garantir software com qualidade. Uma maneira de testar um sistema é fazê-lo executar até que um comportamento incorreto ocorra – *crash* – e então expor um defeito. Entretanto, testes não são baratos. Em uma típica empresa de desenvolvimento, os custos com as atividades de teste, depuração e verificação pode facilmente variar entre 50% e 75% do custo total do desenvolvimento [43]. Automação de testes torna-se então uma importante aliada para reduzir esses custos. O objetivo desta dissertação é propor uma técnica de testes caixa-preta – o Framework de Átomos (AF) – para ajudar na detecção de *crashes*. AF é baseado em cenários automáticos de teste, que são sequências de passos executáveis, escritos manualmente. Também conduzimos um conjunto de experimentos em telefones celulares com a finalidade de verificar a eficácia da técnica que estamos propondo no que diz respeito a sua capacidade de encontrar *crashes*.

Palavras-chave: Teste de software, Geração de testes, Teste de caixa-preta, Teste de detecção de defeito, Checagem de pré e pós-condições

Abstract

Despite the technological advances in languages and tools to support systems development, programmers still deliver software with errors. Several techniques have been proposed to this end – to improve software reliability. Testing is one of them. In fact, software testing is the dominant approach in industry to assure software quality. One way of testing a system is to run it until an incorrect behaviour happens – crash – and so expose a defect. But testing is not cheap. In a typical commercial development organization, the cost of testing, debugging, and verification activities can easily range from 50% to 75% of the total development cost [43]. Automation of software testing then becomes a very important mean to reduce this cost. The objective of this work is to propose a black-box testing technique – the Atoms Framework (AF) – to help the detection of crashes. AF is based on automated test scenarios, i.e. sequences of executable steps that have been manually written. We also conducted a set of experiments on cellular phones to check the effectiveness of AF with respect to its capability to find crashes.

Keywords: Software testing, Test generation, Black-box testing, Defect testing, Pre and post-conditions check

Contents

List of Figures

List of Tables

CHAPTER 1

Introduction

Testing is simple – all a tester needs to do is find a graph and cover it.

—BORIS BEIZER (Software Testing Techniques)

Despite the technological advances in languages and tools to support systems development, programmers still deliver software with errors (bugs). The outcome of a bug can be severe, causing disasters; or in less critical applications, a bug can lead to great market losses. Identifying and fixing potential problems early in the development cycle can then save a considerable amount of time and money.

Several techniques have been proposed to this end – to improve software reliability. It is important to emphasize that each one has limitations. For instance, theorem proving requires a considerable user expertise on logic for both stating program characteristics (invariants) and helping the associated tool support to discard the proof obligations. On the other hand, testing is much simpler and closer to the development than theorem proving, but it can only show the presence of errors, not their absence.

In spite of this, **software testing** is the dominant approach in industry to assure software quality. But testing is not cheap. Santhanam and Hailpern [43] report that, in a typical commercial development organization, the cost of testing, debugging, and verification activities can easily range from 50% to 75% of the total development cost. Reducing the cost of software development and improving software quality then become very important objectives of the software industry.

Defect testing

The specifications of a software under development are in constant change, i.e. they are incomplete, until the final product is delivered to market. Because of these changes, it is hard to keep requirements documents always updated. It demands, for instance, much time, effort, and specialists who know exactly how to write them down. It is indeed a tedious work and currently it still is infeasible to document all requirements in detail. As a result, requirements documents are often also incomplete and do not present all the expected behaviors of the software they describe.

Sommerville [44] defined two orthogonal goals for a testing process: validation (or conformance) and defect testing. The first one aims to design tests to check if the system under test (SUT) is working correctly, i.e. if it is in accordance to what the requirements establish. Defect testing, on the other hand, intends to test everything else that is not in the requirements documents. That is, any situation that makes the system perform incorrectly and so expose a defect. When that happens, we say that the test found a **crash** or it "broke the system".

We recognize a crash situation, for example, if the SUT freezes during the execution of a test, i.e. it stops responding to the testing procedures. The SUT receives the procedures' actions, but it does not correctly react to them. A classical and worldwide known system's crash is the Windows blue screen (see Figure 1.1). A crash may occur in contrived interactions between functionalities of the SUT. For instance, the testing procedures execute an invalid sequence of steps, which the developer has not thought about when writing the code, that results in a system crash.

Figure 1.1 Example of a system crash.

Black-box testing

Traditionally, there are two testing approaches, no matter if the goal is validation or defect testing: white-box and black-box. White-box is the activity of testing that requires knowledge of the program's internal structures, while black-box does not need any information regarding the program organization [12]. Any part of the system providing a public interface is eligible for black-box testing. It consists of exercising the interface of a component (typically the entire system) to find errors.

Advantages of black-box testing compared to white-box testing include independence between the programmer and the tester, which may reduce bias, and no reliance on source code, which may help with work division. Black-box testing is particularly important for organizations that outsource testing activities. Often these organizations need to make private the source code they develop, say for confidentiality reasons. In those situations, black-box testing becomes the primary choice for independent assurance of system's quality.

White-box also offers many advantages when compared to black-box testing. With white-box testing, for instance, we can explore every constituent part of the system and guarantee that we wrote tests that cover, at least once, all possible paths of the code. White and black-box testing are indeed orthogonal approaches to testing, as one may find defects the other may not find. In this work we focus on black-box testing due exactly to lack of access to the source code.

Test automation

Testing is costly. It requires the generation and execution of tests, and also the maintenance of the tests with updated information regarding to changes in the system and its specifications. Although it is a fact that testing increases the quality and reliability of software, some organizations still do not have a testing process or do not attach a lot of importance to testing. They end up executing none or few tests, which results in extra reworks' costs and time expense to correct defects found in later phases of the development cycle.

Generation and execution of tests then become very important activities to automate, especially when requirements are incomplete and the SUT is in constant evolution (changing). In fact, automation of software testing can be one way to **reduce the development costs**, although it is a very time-consuming activity in the beginning. But when well applied, Li and Wu [30] say that automation can reduce the time for testing up to 60%. However, it is necessary to be aware of the trade-off between the size of the projects and the cost to automation. For small projects, this effort may be huge; and the venture costs, high. The time saved during the execution of an automated test must also compensate the time spent to automate it. Otherwise, it will not be worth the effort.

But after the initial investment to prepare the environment to execute automated tests, more cycles of testing can be conducted, even overnight, and more bugs can be found and corrected sooner with less cost. As a result, we have less human error, tests easier to be reproduced and, the most important, **software with higher quality**.

The word automation has several uses in the testing community. It can be used to describe both the automatic generation of sequences of tests or as reference to the automatic execution of tests created manually. In our approach, we use automation towards the first definition.

Context

The focus of this work is on test generation based on existing test scenarios, i.e. sequences of executable steps that have been manually written. This work has been conducted at the Brazil Test Center (BTC) as part of a research effort between Motorola and the Center of Informatics (CIn) of the Federal University of Pernambuco (UFPE) for improving Motorola's software testing process. CIn/BTC is divided in three areas: formal education and hands-on training, operation, and research. This dissertation is one of the research team results. The operation team existing issues and detected improvements motivate the research activities.

One of the operation teams works with several defect testing strategies. One of them, called *Scenario Hunting* (SH) [13], is based on automated test scenarios. Its approach is to frequently develop new test scenarios, based on some indicators, such as: new functionalities (features), escaped defects areas, test cases of other testing phases, and critical features. During execution, these test scenarios reproduce repetitive actions (steps) that would be done by final users in real situations. The creation of SH test scenarios demands the test designer to be both creative and expert in the applications domain, in this case the cellular phone domain. The better the test designer writes elaborated SH test scenarios, the better these test scenarios exercise the applications and lead the execution to detect crashes.

Goals of our work

Despite the effectiveness of SH to find crashes, it requires **much time and manual effort** to always have new test scenarios. In practice, after several execution cycles, the areas where defects were found are fixed. Consequently, these test scenarios become obsolete pretty soon and must be continuously reworked or recycled to be updated with more (or differing) steps, which demands even more time and effort.

Beyond this time and effort expense, each SH test scenario is also a **fixed (data and control) structure**. That is, its steps are always executed in the same order they appear in the scenario, using the same set of inputs. Thus, a SH test scenario always send the same sequence of commands and input data to the SUT. As a result, in every execution it produces the same set of outputs and avoids that other interesting sequence of steps be reproduced.

As it is hard to maintain these test scenarios and also to create new scenarios, we determined as **goals** of our work to:

(i) **propose a new technique**, which will reduce the effort to create new tests, lower the time to find a crash, and also will raise the number of crashes detected;

(ii) **conduct an experimental evaluation** to compare the effectiveness of this new technique with that of SH with respect to their capability to find crashes.

1.1 Our contribution

The main contribution of this work is a black-box testing technique – the Atoms Framework (AF) – to help the detection of crashes. AF is based on SH existing test scenarios, which allows us to take advantage on the knowledge of the specialists who wrote them. We observed that often a step makes sense apart from the sequence. These steps then become reusable units of functionality when extracted from the original scenarios.

We believe that other sequences (with the same set of steps from the test scenario they belong to) can lead the test execution to find crashes more frequently. The technique we propose receives as inputs a set of test scenarios, extracts their steps (which originate what we call *atoms*), adds support for test data, and sorts the generated *atoms* into lists, called execution lists, which guide the execution.

When compared to SH, AF can produce, in each execution, different sequences of steps, i.e. new test scenarios, due to the ordering of the *atoms* that appear in the execution lists. And also, because of the test data support AF offers, each *atom* can exercise different inputs and generate different outputs every time it is executed. For this reason, AF enables *atoms* to share data: one *atom* can consume data another *atom* produces.

Illustrative Example

Note. For confidentiality reasons, the examples we show here and in the following chapters have been modified from the originals. However, this will not compromise their understanding. The examples are also related to the cellular phones domain, but it is worth saying that AF can be used in other domains as its main approach is to generate *atoms*, i.e. tests.

```
log("Capture an audio message.");
navigationTk.launchApp(PhoneApplication.get("VOICE RECORDS"));
multimediaTk.captureVoiceNoteFromVoiceRecord(30);
MultimediaFile audio = multimediaTk.storeMultimediaFileAs(MultimediaItem.get("STORE ONLY"));

log("Listen to an audio message.");
navigationTk.goTo(PhoneApplication.get("VOICE NOTES"));
multimediaGoTo.get("ALL VOICE NOTES"));
multimediaTk.scrollToAndSelectMultimediaFile(audio);

log("Delete an audio message.");
phoneTk.returnToPreviousScreen();
multimediaTk.deleteFile(audio, true);
```

Figure 1.2 A test scenario for audio message.

Figure 1.2 shows a fragment of one test scenario. This scenario consists of three steps (i) capture an audio message, (ii) play that message, and (iii) delete that same message. This test scenario is written in Java and runs on a regular PC. It uses a library to communicate with the cellular phone. AF divides this test based on its steps and generates what we call *atoms*.

In this example, three *atoms* can be generated, according to the instructions associated to each log instruction. Note that one *atom* can have parameters in result of this method extraction. For example, the second *atom* (for listening the audio message) will require a MultimediaFile object. This is key to AF as it enables one *atom* to exercise different inputs.

Overview of the experimental results

Another consequent contribution of our work was to conduct 4 experiments to evaluate AF: one to measure the quality of AF with respect to its ability to find crashes, and other three experiments to give us a better understanding on AF itself. The first experiment compared AF to a scenario-based technique (SH) and quantifies execution time and the capability of finding a crash for each technique on 8 different cellular phone configurations with historical (real) errors. The results show that AF offered a better precision. It found crashes in all 8 configurations. However, when both techniques finds a crash (4 configurations), SH outperforms AF in 3 out of 4 cases.

The second experiment evaluated the impact the randomization of data has in the effectiveness of AF. The experiments reveal that the selection of the random seed for data generation to AF results in a high variance of execution time (i.e., the time the technique takes to either crash the application or timeout in 40h): the mean (across 8 phone configurations) of the standard deviation of execution times (for 10 runs per each phone configuration) was 7.9h. Despite this fact, AF could crash the application consistently: the mean of the precision (fraction of the 10 runs that results in a crash) was 74%.

The third experiment evaluated the impact the random selection of *atoms* to build sequences has in the effectiveness of AF. The experiment indicates that the selection of *atoms* has a great impact on the capability of the technique to find a crash.

Finally, in the fourth experiment, we compared the random generation of the sequence of *atoms* to a more systematic generation, based on pre and postconditions checks. AF found a crash in all executions. Although sometimes AF takes longer to find a crash when compared to the random generation, the results indicate that the use of pre and postconditions checks seems to be effective as AF could find crashes consistently.

1.2 Dissertation Organization

This work is organized as follows:

- Chapter 2 provides an overview of the theoretical basis necessary for the understanding of this work. It shows some concepts of software testing, including defect testing, black-box testing and test automation.
- Chapter 3 describes the technique we propose to generate automated tests that target defect coverage.
- Chapter 4 reports the results we obtained in the experimental evaluations we conducted to evaluate the technique.
- Chapter 5 shows our final considerations, related and future works.

Chapter 2
Software Testing

Software should be predictable and consistent,
offering no surprises to users.
—GLENFORD J. MYERS (The Art of Software Testing)

Software testing is a process, or a series of processes, designed to make sure a computer code does what it was designed to do and that it does not do anything unintended [34]. Or in a more emphatic definition, software testing is the process of executing a program with the intent of finding errors (bugs) [34]. As computers and softwares are used in critical applications, the outcome of an unintended action – in other words, a bug – could be severe, causing disasters; or in less critical applications, a bug can lead to great market losses. The main idea is that software testing must improve the quality or reliability of the system under test (SUT), and consequently reduce the number of bugs found by the final users. Thus, it is worth emphasizing that the goal of software testing is to reduce the number of bugs and not to prove their absence.

A study done in 2002 by the National Institute of Standards and Technology (NIST) [36] shows that the most common types of bugs are due to failure to conform to specifications or standards, to interoperate with other software and hardware, and to meet minimum levels of performance as measured by specific metrics. So, in order to test a program until no more bugs are found, a test would need to cover all possible paths the program has with every possible input data values. Such combination of paths and input data values is not feasible in general, even for trivial programs. Exhaustive testing, where each possible sequence of execution of the program is tested, is impossible [44].

Although it is not possible to guarantee that a software is completely free of bugs, testing must ensure that the deliverable software is sufficiently good for operational usage and that it provides a high level of confidence to the users. That is, the testing process must check that the SUT works well in distinct environment configurations and that it really does what it is supposed to do.

As we cannot test a program in every possible way, we consider only a part of all possible tests. Requirements are a great source of information to guide the testing process to prioritize the critical parts of the software that must be selected to test, and they help the developers to demonstrate that the software does what was agreed with the clients.

The requirements are in fact the basis of one of the two testing process goals defined by Sommerville [44], the **validation (or conformance) testing**. The intention of validation testing is to find cases where the program does not do what it is supposed to do. The SUT is executed under a given set of tests, which reflects the expected behavior that the requirements establish.

The other testing process goal is **defect testing**. While for validation testing, a successful test is the one with which the SUT works correctly, for defect testing a succeeded test is the one that exposes a defect that causes the program to behave in an anomalous way. In other words, the main idea under defect testing is not to check if the program is in accordance with its requirements. Instead, defect testing aims to find unexpected behaviors, such as, system crash, undesired interactions with other systems, inaccurate calculations and data corruption [44]. When a bug is found while working with a defect testing approach, it is commonly called a **crash**.

During testing, test **oracles** check whether the SUT executed as expected [49], i.e. the actual output is compared to what it is expected to be the correct output. During the execution of a defect testing technique, the expected behavior of the SUT, for instance, may be that it keeps answering the testing procedures and does not get stuck in one screen. Otherwise, the oracle must report a crash. Test oracles may be either automatic or manual (a tester manually interacts with the SUT and visually checks if there are any bugs). Advantages of automatic oracles when compared to manual ones include less human error and more accuracy. Although a manual oracle is very effective when interpreting incomplete and natural language specifications, it is prone to error when dealing with complex behaviors, and also its accuracy drops significantly the more runs of test have to be evaluated [10].

Traditionally, there are two approaches to testing, no matter if the goal is validation or defect testing. One is based on what the software is supposed to do. The other is based on how the software actually works. The first one is called **black-box**; and the second, **white-box**. In the following sections, we discuss each approach separately.

2.1 White-Box Testing

White-box testing, or logic-driven testing, permits one to examine the internal structure of the program, and so the knowledge of the code and the internal structure are pre-requisites for this approach. Thus, white-box testing is concerned with the degree to which tests exercise or cover the logic (source code) of the SUT [34].

Testers that use white-box techniques seek to locate logical errors and verify test coverage. And so, they create tests that focus on the module design of the program and are based on the implementation. Figure 2.1 shows an overview of white-box testing. In the example, the input “x = 1" is given to the SUT. Thus, the testers know exactly what path of the code need to be executed in order to generate the expected output: “y = 43”.

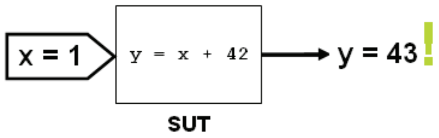

Figure 2.1 White-box testing overview.

In order to know how the program works and whether its modules/functions and structures meet the functional and design specifications, a valid attempt would be to execute all possible paths of the code. However, it is an impossible task to achieve. So, test procedures associated with the white-box approach aim to provide several services, including the following [21]:

1. All independent paths within a module are exercised at least once.
2. All logical decisions, on their true and false sides, are exercised.
3. All loops at their boundaries and within their operational bounds are executed.
4. The internal data structures are exercised to ensure their validity.

There are many techniques for white-box testing with the intention to achieve most of these services, or all of them, for instance: fault insertion, statement coverage, decision coverage, condition coverage, and others. A brief description of them can be found in [21].

An important white-box technique is unit testing. Its purposes are to verify that the logic implemented works properly and that all the necessary logic is actually present [33]. For many reasons, including market pressures to delivery software as soon as possible, sometimes unit tests are put away. But what the experience shows is that, by applying unit testing to a system, the chance to find a bug in later phases decreases considerably; and once a bug is found, it can be corrected right-away by the code writer. Nevertheless, it is an expensive activity, where it takes up to 17% of the overall development cost [48]. Thus, over the years, many works have been done as a try to improve this situation, by systematizing and automating the unit testing process. We describe some of recent researches in this field.

Pacheco *et al* [40] proposed a technique that randomly generates unit tests by incorporating feedback obtained from executing test inputs as they are created. This technique guides the search towards sequences that yield new and legal object states, by not extending inputs that create redundant or illegal states. The technique has a support tool called *RANDOOP*.

RANDOOP incrementally creates sequences of method calls, by randomly choosing a method call to apply and selecting arguments from previously constructed sequences. Its algorithm checks if a generated sequence is equivalent to a previous generated sequence. If the answer is positive, it tries to create a new sequence. Right after a sequence is generated, it is executed against a set of contracts and filters. Each contract checks whether the sequence satisfies or violates the current state of the system (the runtime values created in the sequence so far, and any exception thrown by the last call). So, the sequences that satisfy and the ones that violate the current state are added to different sets. Filters, that determine which values of a sequence are extensible and should be used as inputs to new method calls, are also applied to the ones that violate the state. Finally, the *RANDOOP*'s outputs are two sets (called *nonErrorSeqs* and *errorSeqs*) written as JUnit [7] tests, along with assertions representing the contracts checked. The first set represents the tests where the tested classes pass; they could be used for regression testing later on. On the other hand, the second set contains the tests where the classes fail, and so, they indicate likely errors in the code under test.

Godefroid *et al* [24] proposed a technique for automatic generation of unit test cases that combines static and dynamic approaches. It has also a support tool called *DART*. First, the interface of a program with its external environment is extracted using a static code parsing. Then, a test driver, which performs random testing, is automatically generated for this interface. And finally, dynamic analysis are executed to examine how the program behaves under random testing, while new test inputs are automatically generated to direct the execution along alternative paths in a systematic way with what they call a *directed search*. The execution is started with a random input. After that, an input vector for the next execution is calculated during each execution. The new input vector attempts to force the execution of the program through a new path. By repeating this process, a *directed search* attempts to force the program to sweep through all its feasible execution paths.

There is another approach, Oriat presents in [38], for random generation of unit tests, explicitly for Java classes, supported by a tool called *Jartege*. The classes must be specified in JML (Java Modeling Language) [29], which is a specification language for Java that allows one to write invariants for classes, and pre and postconditions for operations. *Jartege* produces test programs which are composed of test cases. Each test case consists of randomly chosen sequences of method calls for each class under test. While the tests are generated, each operation call is executed on the fly. This permits the elimination of calls that violate the operation precondition, and to tell whether the test case passed or not (the specification in JML is used here as test oracles). The test generation can be parameterized by associating weights to classes and operation, which defines the probability that a class will be chosen, and that an operation of this class will be called; and also by controlling the number of instances created for each class under test. It also provides a way of generating parameter values of primitive types for operations.

2.2 Black-Box Testing

While white-box testing concerns itself with the program's internal behavior, black-box testing compares the behavior of the application against what is stated in the requirements. Black-box testing, or behavioral testing, is the type of testing which is closer to what the user experiences while using the system. The only way to test a program under a black-box technique is to have access to public interfaces such as the user interface or the published application programming interface (API).

Figure 2.2 shows an overview of the black-box approach. The system is treated as a "black box" and its functionalities are tested with regards to their specifications (requirements) and the context with which the system is related to (events). Thus, only the correct inputs/outputs relationship is under investigation. In the example, the testers only know that the output "y = 43" is expected when they give "x = 1" as input to the SUT. The computations involved for generating the right result are out of touch.

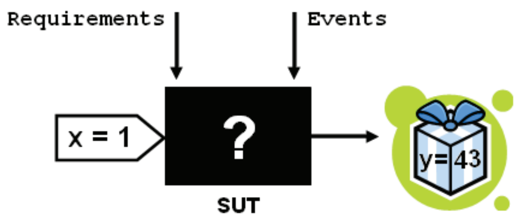

Figure 2.2 Black-box testing overview.

Black-box testing methods, just like the ones of white-box testing, include a considerable number of techniques, such as: all-pairs testing [46], fuzz testing [31], boundary value analysis [34], random testing [25], model-based testing [20], and others.

With the popularization of object orientation and models in software over the years, model-based testing, MBT for short, has shown a considerable growth in its usage. MBT can be understood as an approach that generates test cases using models of the SUT [20]. These models can be developed early in the life cycle as they are essentially a specification of the inputs to the software. A system model can be built either through an automatic or a manual way. In an automatic way, tools are used to automatically extract the system model. In a manual way of generating a system formal model, software engineers design the system behavior following a formalism. Many modeling languages are used to build those models, for instance UML diagrams [11, 14, 19, 32, 47], statecharts [9, 23, 27], CSP notation [37], and LTS [18]. In principle, after having built the model the test cases derivation can be done automatically. There are many different ways to derive tests from a model. Because testing is usually experimental and based on heuristics, there is not a best way to do this.

Hong *et al* [27], for instance, presents a method for automatic selection of test sequences from statecharts [26] based on data-flow analysis. They say that these test sequences allow one to determine whether an implementation establishes the desired associations between definitions and uses, which are expressed in the statechart.

Basanieri and Bertolino [11] presents another MBT approach, which describes a manual technique to generate tests from UML Use Case and Integration diagrams (specially the Sequence diagram). Mingsong et al [32] also describes a technique to generate test cases from UML diagrams, but this time in an automatic way and using Activity diagrams. In this method, the test cases are not directly derived from UML diagrams. Instead, it first randomly generates a set of test case according to a given Activity diagram and then it prunes some redundant test cases and gets a reduced set which meets the test adequacy criteria. In order to do this, it needs to instrument the SUT. Vieira *et al* [47] describes a research on test case generation based on UML diagrams. In their approach, the diagrams are annotated with test data requirements that allow test cases to be generated based not only on Activity diagrams but also on data coverage.

2.3 Test Automation

Automation of testing process is not only desirable, as the whole testing process is itself a very time-consuming activity – it requires up to 50% of software development costs, and even more for critical applications [8]. Test automation is indeed a necessary activity to be accomplished given the demands of the current market for software delivered as early as possible and with high level of quality.

It is a common sense since Boehm, Brown and Lipow [16] showed that the sooner a bug is detected the least expensive it is to fix it. Figure 2.3 illustrates this. With automation, more

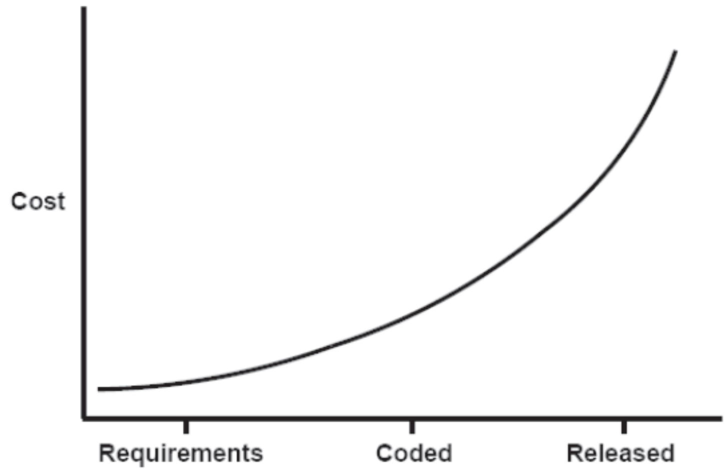

Figure 2.3 How cost with bug grows the later it is detected

tests can be conducted and more bugs can be found faster. Some companies report that their elapsed time for testing has been reduced up to 60% through automation [30]. Besides this time reduction, the number of defects discovered by automated tests increases up to 50% [30]. As a result of testing automation, we have less human error, tests which are easier to be reproduced, lower costs to fix bugs, and software with higher quality.

However, test automation can be an expensive and also a very time-consuming part of the testing process. For small projects, this effort may be too much and may not be worth the venture. Even for larger projects, if the automated tests are only used once or are not reused by other projects, the amount of time spent to create them may not compensate the effort.

There are some ways to alleviate the uncertainty regarding the overall effort and time to be spent in test automation. One of them is to select the types of testing to automate. For instance, **unit** and **integration testing** must be strongly considered to be automated [33]. Unit testing aims to test all the individual modules in a program, i.e. subprograms, subroutines, or procedures [34]. And the purpose of integration testing is to assess whether the interfaces between modules have consistent assumptions and communicate correctly [8]. Automated unit and integration testing are valuable not only because nowadays there are a lot of tools to help developers to achieve these type of testing (such as *CUnit* [2], *DUnit* [3] and *JUnit* [7]), but also because the quality of the builds given to other test phases, such as system testing, increases with unit and integration testing automation [33]. What happens is that more defects regarding to isolated functions or methods are found during development time and corrected earlier, and only defects more related to malfunction issues and requirements divergence are left to be found in later phases.

Other types of testing, for which automation provides great value, are **regression testing** and **stress testing** at the system level. Regression tests seek to verify that the functions provided by a modified software perform as specified and that no unintended change has been made in operational functions [21]. Stress testing subjects the software to heavy loads or stresses over a short amount of time to determine when and how it fails [34]. Ammann and Offutt [8] emphasizes that unautomated regression testing is equivalent to no regression testing at all. In fact, in order to re-test software that has been modified, we need to run and re-run tests, most of time a lot of tests, to determine whether a new error has been introduced. In stress testing, we also need to run and re-run tests to simulate a large amount of accesses to the system and data manipulation. Without automation, both types of testing would be impracticable and require higher effort and time.

Another way to minimize effort and time in test automation is to use testing frameworks, which helps the creation, execution and maintainability of automated tests. There are a lot of commercial and open source testing frameworks ([2, 3, 7, 1, 4]). Each of them with different functionalities for different types of systems. *CUnit* [2], *DUnit* [3] and *JUnit* [7], which we have cited before, are frameworks useful for unit and integration testing for C, Delphi and Java programs, respectively. *Abbot* [1] is an open source framework that performs both GUI (Graphical User Interface) unit and functional testing for Java applications. It also provides an interface to control event playback in order to enhance the integration and functional testing. *GUITAR* [4] is another open source framework for GUI testing that helps the generation, execution and re-execution of functional and regression testing. It offers event-based tools and techniques for various phases of GUI testing.

Despite the great number of testing frameworks available nowadays, organizations still develop their own framework to fit their needs for specific functionalities. Sometimes they have to test components developed by different manufacturers, which use different programming languages and run in different platforms. Motorola Inc., for instance, relies on such an environment and tests the cellular phones they develop, which are built on different hardware and software configurations. Because of this, Motorola developed a proprietary testing framework called TAF (Testing Automation Framework) [28, 35], which is used to automate and execute high level tests for cellular phones.

What differs TAF from other testing frameworks is its automated tests, which are composed of method calls with high abstraction level. Thus, it is possible to keep tests that can be used in different products without changes in the tests' procedures, while methods (written in what we call high level of abstraction, i.e. specific details related to implementation, hardware, screen's layout, etc. are not considered) are reimplemented only when they need to be adjusted to represent specific behaviors of new phones. In this way, Motorola can minimize costs with the maintainability of tests and increase methods' reuse in different tests. Section 2.3.1 gives more details on TAF, which is used to create the tests we describe later in this work.

2.3.1 TAF: Test Automation Framework

TAF [28, 35] is a proprietary object-oriented testing framework that supports the automation of integration and system testing for all software embedded in the cellular phones Motorola develops. TAF enables reuse by raising the abstraction level so as to make automated tests largely independent of model-specific phone properties.

Figure 2.4 Screens of different phone models.

Although there are cellular phones that implement the same functionalities (features), the way the user interacts with them varies depending on the phone model. Figure 2.4 shows screens of two different phone models. Note that the *Address Book* item appears on the left-hand side of screen (1) and on the right-hand side of screen (2). The same happens to the *Messaging* item. But, when writing tests with TAF for these features, the position of the items on the screen are totally irrelevant to the test developer. So, for instance, when a test calls the Address Book application in phone model (1), it clicks on the bottom-left button of the phone. And when the same test is run against phone model (2), it clicks on the bottom-right button to access the Address Book. Thus, the same set of TAF tests can be reused in several phone models.

TAF uses another proprietary artifact to communicate with the cellular phone, a library called Phone Test Framework – PTF [22]. PTF communicates with the phone via a USB (Universal Serial Bus) interface to provide functions that allow, for instance, the recognition of all enabled contents of each phone screen and the simulation of the phone key pressing. Since most PTF methods are encoded at low abstraction levels, it leads to test scripts that are hard to read, difficult to maintain and inefficient to reuse to other phones. However, PTF represents a highly appropriate basis for test automation implementation [22].

TAF encapsulates direct calls to PTF methods in *Utility Functions* (UFs). Each UF represents a sequence of keys to be pressed and/or the verification of each screen's contents. Thus, UFs are primitive entities that hierarchically isolate functionality from implementation, leading to high-level automated tests [28, 35]. In other words, each UF is an implementation of a high-level step that can be executed automatically by a test. Table 2.3.1 shows fragments of test scripts written in TAF and PTF. Although written in different levels of abstraction, their objectives are the same: to launch the camera application. Note that, to do so, TAF uses only one command, while PTF uses four. On the one hand, the UF *launchApp* hides specific details regarding on how the phone reaches the camera application. On the other hand, a PTF script needs to explicitly have all of these navigation details (go to the initial phone screen, press the phone's right key, look for the desired item and, at last, press the phone's center key) to open the camera correctly.

TAF Test Script	**PTF Test Script**

// Launch Camera application
navigationTk.launchApp(PhoneApplication.CAMERA);
	// Go to Idle (initial screen)
	ptf.nav.idle();
	// Enter in Main Menu
	ptf.phone.pressKey(RIGHT);
	// Scroll to Camera
	ptf.nav.scrollTo(CAMERA);
	// Select to enter in Camera
	ptf.phone.pressKey(CENTER);

Table 2.3.1 TAF versus PTF test scripts.

Figure 2.5 shows a fragment of a TAF test script, which performs 2 steps when executed against a phone. It first takes a picture and stores it in the phone file system (lines 2 to 4). Then, it deletes the picture taken (lines 7 to 8). Note that it uses 5 TAF UFs: *launchApp* (in this example, this UF opens the camera application), *capturePictureFromCamera* (captures a picture with the camera application), *storeCapturedPictureAs* (stores the picture taken in the phone file system), *returnToPreviousScreen* (drives the phone to the previous screen it visited) and *deleteFile* (in this example, it removes the picture taken from the phone file system).

TAF has potentially more than one implementation of the same UF in order to allow the same UF to run in several phone models and reproduce the same high-level objective. TAF relies on the notion of Feature Toolkits (from now on called toolkits), which are collections of UFs that implement functions of the same application or objective and allow proper instan-

```
1 // Take a picture
2 navigationTk.launchApp(PhoneApplication.get("CAMERA"));
3 multimediaTk.capturePictureFromCamera();
4 PictureFile pic = multimediaTk.storeCapturedPictureAs(MultimediaItem.get("STORE_ONLY"));
5
6 // Delete a picture
7 phoneTk.returnToPreviousScreen();
8 multimediaTk.deleteFile(pic, true);
```

Figure 2.5 Fragment of a TAF test script.

tiation of UF implementations for a given phone model. Thus, a TAF test consists in several calls to methods (UFs) encapsulated in these toolkits. For instance, UFs under the *navigationTk* toolkit (*launchApp* in line 2) have the objective to help the navigation in the phone system. UFs under the *multimediaTk* toolkit (*capturePictureFromCamera* in line 3, *storeCapturedPictureAs* in line 4, and *deleteFile* in line 8) deal with Multimedia applications and files. Kawakami *et al* [28] show a more detailed description of the TAF toolkits.

Figure 2.6 summarizes the hierarchy of TAF layers from the highest to the lowest level. It shows the relation between the layer where the test scripts are, the one where the toolkits are implemented, the one where the UFs are implemented and the PTF layer.

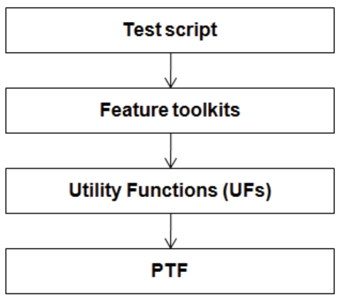

Figure 2.6 Overview of the TAF architecture.

CHAPTER 3

The Atoms Framework

Testing can only show the presence of errors, not their absence.

—DIJKSTRA ET AL

Atoms Framework – AF – is a black-box testing technique for the detection of crashes. It builds on existing test scenarios, i.e. sequences of executable steps that have been manually written. Each of these steps characterizes *one* important interaction with the system. AF receives test scenarios as input, extracts their steps and shuffles the steps in sequences that guide the execution. The main idea is to use the knowledge of an expert (as the input test scenarios are manually written) in order to create new tests that verify situations not described in the original scenarios.

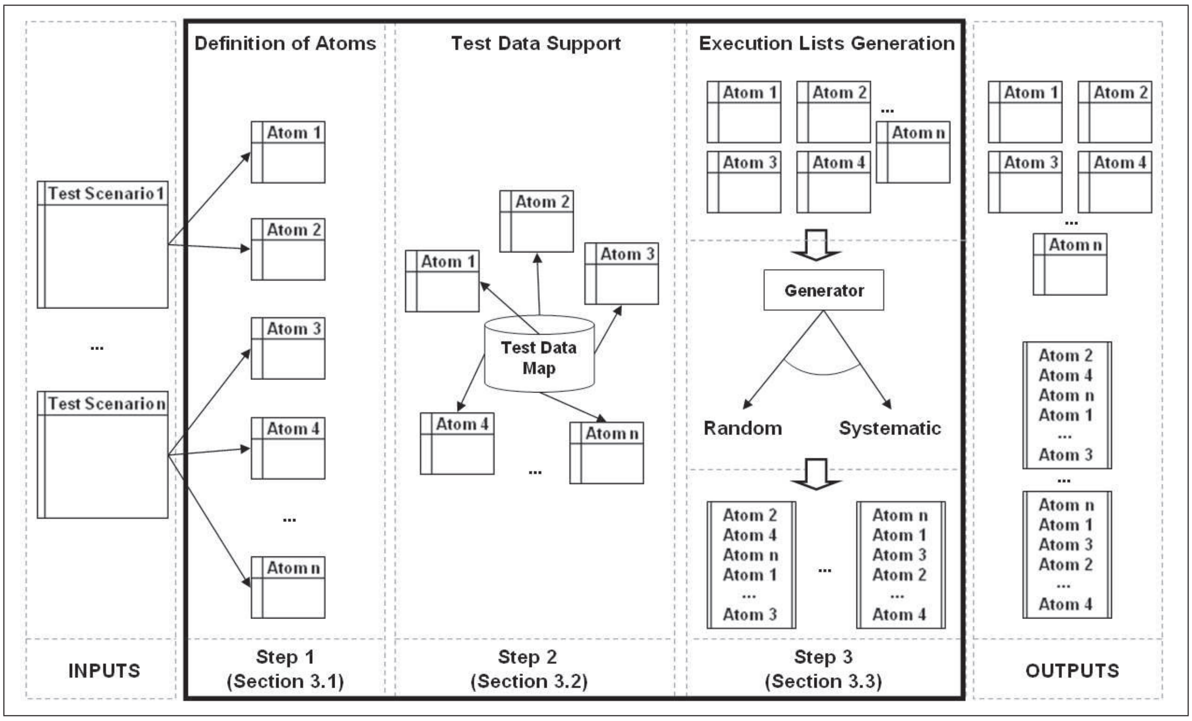

Figure 3.1 Overview of AF's activities to produce *atoms* and execution lists.

Figure 3.1 illustrates an overview of the activities AF defines to produce *atoms* and execution lists. The inputs for AF are tests scenarios. We observed that each test scenario is a sequence of relevant steps, and that often a step makes sense apart from the sequence. These steps become *reusable* units of functionality when extracted from the original scenarios. We believe that other sequences (with the same set of steps from the test scenario they belong to) can lead the test execution to also find a *crash* and faster. We decided to call the extracted steps, *atoms*. Thus, AF identifies the steps of each test scenario and creates *atoms* based on them (**Step 1**).

The test scenarios we use in our approach fix in their steps all input data they need in order to execute. As AF creates *atoms* based on those steps, the resulting *atoms* may depend on data other *atom* generates. Therefore, *atoms* can assume role as producer and consumer of data. We decided to add a data map to support this data dependence (**Step 2**). Thus, an *atom* uses the map to retrieve input data and to store all data it generates.

In order to execute *atoms* we need to add them to execution lists. These lists guide the execution, i.e. *atoms* are carried out in accordance to the order they appear in the list. AF defines two strategies to the creation of execution lists (**Step 3**): one that randomly order *atoms* in a list; and another one more systematic that takes into account the context required by each *atom* to execute. The *atoms* created and a set of execution lists are the outputs for the technique. The size of the set is defined by the user.

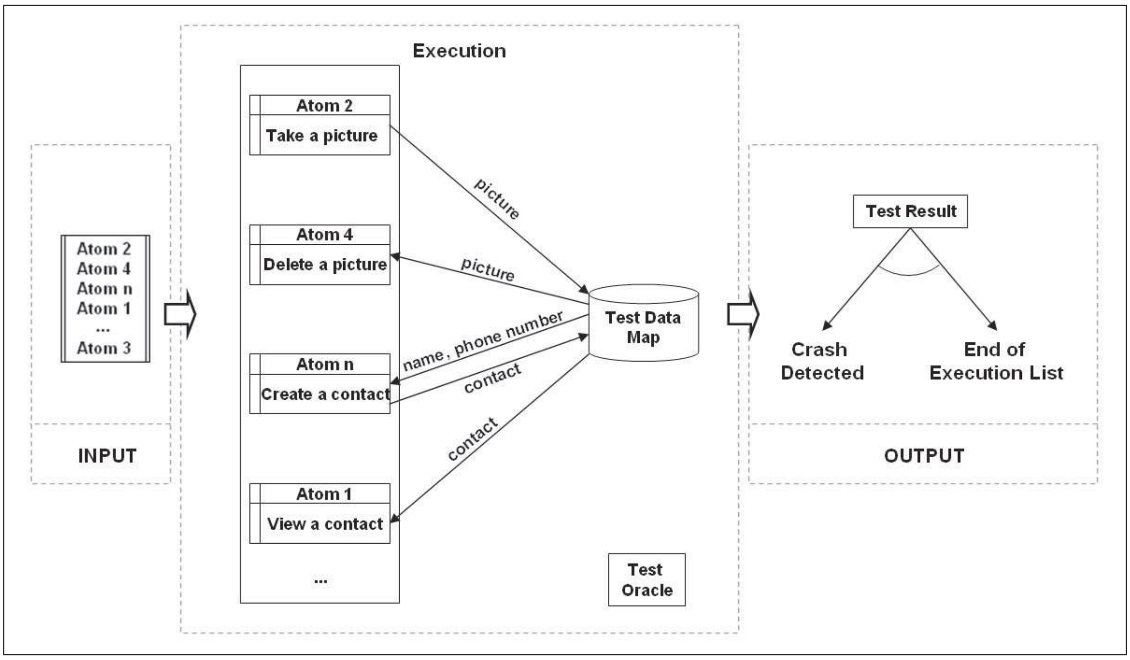

Figure 3.2 Overview of execution.

Our approach does not focus on the execution of *atoms* since we reuse a Motorola proprietary framework to automate this step. However, Figure 3.2 shows an overview on how the execution proceeds with *atoms*. On its left-hand side, there is an execution list, which is the input for the execution process. Before the execution process starts, the data map can be loaded with inputs and also an initial configuration test can be run to fill the SUT with data, which are also stored into the data map. The first *atom* in the example takes a picture, and then puts it into the data map. When the second *atom* is loaded, it first accesses the data map to retrieve a picture to be deleted. Then it removes from the system and also from the data map the selected picture, which can be the one taken by the first *atom* or another picture present in the data map. The third *atom* creates a contact. In order to do so, it fills the name and phone number fields with data retrieved from the data map and stores the created contact in the data map.

A test oracle checks whether the effect of the execution of an *atom* is as expected. Test oracles are out of the scope this work. To simplify the discussion, the test oracle we use is another program that checks whether the application can still make progress with the execution. When a crash is detected by the oracle or there are no more *atoms* in the execution list, the execution ends. The output of the execution process is a test result that indicates whether the execution detected a crash or the end of the execution list was reached without a crash.

3.1 Definition of Atoms

The first step to create *atoms* based on existing test scenarios is to define (recognize) them. In our approach, we consider a test scenario as a sequence of automated steps. Figure 3.3 shows an example of a test scenario written in Java over the TAF framework [28, 35]. It is a class that implements 3 methods: *setup* (set of steps that creates a context for the appropriate execution of the testing procedures, line 5), *buildProcedures* (procedures of the test, line 7) and *tearDown* (set of steps that restores the initial context, line 28). In our example, *setup* and *tearDown* are empty, i.e. the test does not need any previous/post actions to update the system state before/after the execution of the procedures of the test.

In order to develop a test scenario, a set of TAF UFs is used to implement each desired step that composes the test. Figure 3.3 shows a test, which performs 4 steps when executed against a phone. First, it creates a playlist[1] with 3 songs (line 10). Then, it plays the playlist created in the first step (lines 13 to 15). After that, it takes a picture and stores it as a file (lines 18 to 20). And finally, it deletes the picture taken (lines 23 to 24).

[1] A customized sequence of songs that can be played by the phone.

```
public class TC_MULTIMEDIA {

  public TC_MULTIMEDIA() {}

  public void setup() {}

  public void buildProcedures() {

    log("Create a playlist.");
    stressTk.createRandomFilesPlaylist(Playlist.get("PLAYLIST_02"), 3, 10);

    log("Play the created playlist.");
    navigationTk.goTo(PhoneApplication.get("MUSIC"), MultimediaGoTo.get("PLAYLIST"));
    multimediaTk.scrollToAndSelectMultimediaFile(Playlist.get("PLAYLIST_02"));
    navigationTk.goToAndSelectMenuItem(MultimediaItem.get("PLAY"));

    log("Take a picture.");
    navigationTk.launchApp(PhoneApplication.get("CAMERA"));
    multimediaTk.capturePictureFromCamera();
    PictureFile pic = multimediaTk.storeCapturedPictureAs(MultimediaItem.get("STORE_ONLY"));

    log("Delete a picture.");
    phoneTk.returnToPreviousScreen();
    multimediaTk.deleteFile(pic, true);

  }

  public void tearDown() {}

}
```

Figure 3.3 A test scenario for multimedia.

Note that not always only one TAF UF is employed to reproduce a step. Sometimes a sequence of calls to UFs is used to do so. In order to take a picture, for example, it is first employed a UF that launches the camera application (*launchApp*), then another to capture a picture (*capturePictureFromCamera*), and finally another one to store the captured picture in a file (*storeCapturedPictureAs*). In the remaining steps, 6 other different UFs are employed: *createRandomFilesPlaylist* (it creates a playlist with songs chosen randomly), *goTo* (it takes the phone to a sub-application), *scrollToAndSelectMultimediaFile* (it looks for and selects a specific multimedia item in a list), *goToAndSelectMenuItem* (it selects a specific menu item), *returnToPreviousScreen* (it drives the phone to the previous screen it visited) and *deleteFile* (it removes a specific file from the phone file system).

Each step can be delimited by a call to the UF *log*. This UF simply prints a text (description of the step) in a log file. The log file is a good practice to keep the trace of all high-level steps executed during the testing procedures. In our approach, these calls to the UF *log* help the recognition of *atoms*, as it will be explained later in this chapter.

Our aim is to fragment a test scenario into smaller units that correspond exactly to the high-level steps the test performs. The idea is that new scenarios could be exploited by split-

```
// Step #1
log("Create a playlist.");
stressTk.createRandomFilesPlaylist(Playlist.get("PLAYLIST_02"), 3, 10);
```

```
// Step #2
log("Play the created playlist.");
navigationTk.goTo(PhoneApplication.get("MUSIC"), MultimediaGoTo.get("PLAYLIST"));
multimediaTk.scrollToAndSelectMultimediaFile(Playlist.get("PLAYLIST_02"));
navigationTk.goToAndSelectMenuItem(MultimediaItem.get("PLAY"));
```

```
// Step #3
log("Take a picture.");
navigationTk.launchApp(PhoneApplication.get("CAMERA"));
multimediaTk.capturePictureFromCamera();
PictureFile picture = multimediaTk.storeCapturedPictureAs(MultimediaItem.get("STORE_ONLY"));
```

```
// Step #4
log("Delete a picture.");
phoneTk.returnToPreviousScreen();
multimediaTk.deleteFile(picture, true);
```

Figure 3.4 Steps highlighted from a multimedia test scenario.

ting the original test scenario into their corresponding steps. These steps will originate *atoms*. Figure 3.4 shows 4 steps highlighted from the multimedia test scenario in Figure 3.3: *Step #1* creates a playlist, *Step #2* plays the playlist, *Step #3* takes a picture and *Step #4* deletes a picture.

In our example, each step were recognized by the call to the UF *log*, i.e. all code after a call to the UF *log* until another call to *log* or the end of the file represents a step. But what if the test scenario does not have any calls to the UF *log* due to bad codification? This means that we cannot keep track of the high-level steps executed by the phone during the test.

In order to illustrate this, Figure 3.5 shows the procedures of another test scenario. This time, without calls to *log*. When executed, it performs 5 steps: it first creates a playlist with 3

```
public void buildProcedures() {

  stressTk.createRandomFilesPlaylist(Playlist.get("PLAYLIST_02"), 3, 10);
  multimediaTk.addAllSongsToPlaylist(Playlist.get("PLAYLIST_02"));
  multimediaTk.playSoundInBackground(Playlist.get("PLAYLIST_02"));
  phoneTk.sleep(5);
  navigationTk.goTo(PhoneApplication.get("MUSIC"), MultimediaGoTo.get("PLAYLIST"));
  multimediaTk.scrollToAndSelectMultimediaFile(Playlist.get("PLAYLIST_02"));
  navigationTk.goToAndSelectMenuItem(MultimediaItem.get("PLAY"));
  phoneTk.sleep(15);
  navigationTk.goTo(PhoneApplication.get("MUSIC"), MultimediaGoTo.get("PLAYLIST"));
  multimediaTk.deleteFile(Playlist.get("PLAYLIST_02"), true);

}
```

Figure 3.5 Test procedures without log.

randomly chosen songs (line 3); then, it adds all available songs to the playlist (line 4); then, it plays the playlist in background[2] during 5 seconds (line 5 to 6); after that, it plays again the playlist during 15 seconds, but not in background (lines 7 to 10); and finally, it deletes the playlist (lines 11 to 12). In the following, we outline some tips to recognize the steps of a test scenario when there are no calls to the function *log* in its procedures. The tips were collected by empirical observations and may not work in all situations. They are also not exhaustive. They are just a guide to help the recognition.

Tips for recognizing steps:

[Tip A] Sequential calls to UFs from different toolkits (not including the *navigationTk* or the *phoneTk* toolkits) usually mean that each call corresponds to a different step. Lines 3 and 4 of Figure 3.5 correspond to two steps. The first one itself creates a playlist, and it uses the UF *createRandomFilesPlaylist* from the *stressTk* toolkit to do so. The second one adds all available songs in the phone to a playlist, and to do so it is necessary to use only the UF *addAllSongsToPlaylist* from the *multimediaTk* toolkit.

[Tip B] We consider blocks of code, structures that have in the first line a call to the UF *goTo* or to the UF *launchApp*, both from the *navigationTk* toolkit, followed by other UFs. They always correspond to steps. A block usually has calls to functions from at most 3 different toolkits, two of them as being the *navigationTk* and the *phoneTk* toolkits. We can delimitate the end of a block by one of the following possibilities: (i) by the beginning of another block, i.e. another call to the function *goTo* or *launchApp*; or (ii) by a call to a function from another toolkit that is not already part of the block; or (iii) by a call to the UF *sleep*; or (iv) by the end of the procedures. The block of code from line 7 to 10 corresponds to a step that plays back a playlist during 15 seconds (i or iii). And the block of code from line 11 to 12 corresponds to another step, which deletes a playlist (iv).

[Tip C] If we notice sequential calls to UFs from the same toolkit (again, not including the *navigationTk* or the *phoneTk* toolkits) and they are not inside a block, they usually correspond to different steps. For instance, lines 4 and 5.

After identifying the steps present in a test scenario, they become *atoms*, i.e. Java classes that implement the methods *setup*, *buildProcedures* and *tearDown*, as regular test scenarios. It is not the case of our example, but if the methods *setup* and *tearDown* of the original test scenario were not empty, we simply copy their content to all *atoms* we create from that test scenario. In the following, we show 4 *atoms* created based on the 4 steps (Figure 3.4) extracted

[2]Playing a playlist in background means that the phone can perform any other tasks while it plays songs.

```java
public class ATOM_CREATE_PLAYLIST {

  public ATOM_CREATE_PLAYLIST() {}

  public void setup() {}

  public void buildProcedures() {
      log("Create a playlist.");
  --> stressTk.createRandomFilesPlaylist(Playlist.get("PLAYLIST_02"), 3, 10);
  }

  public void tearDown() {}

}
```

Figure 3.6 Atom #1: Creates a playlist.

from our first example (the multimedia test scenario in Figure 3.3).

Figures 3.6 and 3.7 show *Atom #1* and *Atom #2* related to *Step #1* and *Step#2*, respectively. Although different, these *atoms* are related to each other. *Atom #1* creates a playlist named *PLAYLIST_02* while *Atom #2* selects the same playlist to be played. Note that the playlist's name is fixed. This means that *Atom #2* will only execute correctly, if a playlist named *PLAYLIST_02* has been created before by another *atom* or it already exists on the phone. The same happens to *Atom #3* (Figure 3.8) and *Atom #4* (Figure 3.9) related to *Step #3* and *Step #4*, respectively. *Atom #3* creates a picture file (line 11 of Figure 3.8), while *Atom #4* deletes a picture file (line 10 of Figure 3.9). *Atom #4* will not compile correctly, because the object it tries to delete is not defined in the class.

We can fix this compilation problem, by keeping in a map all objects present in the phone during the *atoms* execution. Thus, all objects created during the execution are added to a map. When an *atom* needs an object, for instance a picture, it then can retrieve from the map. We

```java
public class ATOM_PLAY_PLAYLIST {

  public ATOM_PLAY_PLAYLIST() { }

  public void setup() {}

  public void buildProcedures() {
      log("Play the created playlist.");
      navigationTk.goTo(PhoneApplication.get("MUSIC"), MultimediaGoTo.get("PLAYLIST"));
  --> multimediaTk.scrollToAndSelectMultimediaFile(Playlist.get("PLAYLIST_02"));
      navigationTk.goToAndSelectMenuItem(MultimediaItem.get("PLAY"));
  }

  public void tearDown() {}

}
```

Figure 3.7 Atom #2: Plays a playlist.

```
public class ATOM_TAKE_PICTURE {

  public ATOM_TAKE_PICTURE() {}

  public void setup() {}

  public void buildProcedures() {
      log("Take a picture.");
      navigationTk.launchApp(PhoneApplication.get("CAMERA"));
      multimediaTk.capturePictureFromCamera();
  --> PictureFile pic = multimediaTk.storeCapturedPictureAs(MultimediaItem.get("STORE_ONLY"));
  }

  public void tearDown() {}

}
```

Figure 3.8 Atom #3: Takes a picture.

will discuss more about this in Section 3.2.

As can be noticed, we changed line 24 of *Step #4* in Figure 3.4 when we created *Atom #4*. It is necessary when a step has a call to the function *returnToPreviousScreen*. As the steps of a test scenario are executed in sequence, this function knows exactly in which screen the phone is and was before it was called. When we are talking about *atoms*, we do not know whether the previous screen is the one we are interested in. It depends on the last *atom* executed. For instance, the last *atom* executed could be in a screen of the messaging application, while the current *atom* in execution needs the phone to be in a specific screen of the calendar. This is the reason why in *Atom #4* we needed to replace the call to function *returnToPreviousScreen* by a call to function *goTo* (Figure 3.9, line 9), indicating in which application the phone must be (in this case, the phone must be in the sub-application *ALL_PICTURES* of the application *PICTURES*). This is the only way we can guarantee that the next functions will execute correctly.

```
1  public class ATOM_DELETE_PICTURE {
2
3    public ATOM_DELETE_PICTURE() {}
4
5    public void setup() {}
6
7    public void buildProcedures() {
8        log("Delete a picture.");
9    --> navigationTk.goTo(PhoneApplication.get("PICTURES"), MultimediaGoTo.get("ALL_PICTURES"));
10   --> multimediaTk.deleteFile(pic, true);
11   }
12
13   public void tearDown() {}
14
15 }
```

Figure 3.9 Atom #4: Deletes a picture.

The resulting *atoms* we showed in this section are incomplete, due to the lack of test data maintenance. The following section discusses about this in detail.

3.2 Test Data Support

The test scenarios we extract *atoms* from (as the one in Figure 3.3) do not keep the data generated during their execution. Once the execution of a test scenario finishes, we miss reference to all generated data. And also, all input data a test requires for its execution (for instance, a name to give to a playlist – *PLAYLIST_02* back to line 10 of Figure 3.3) are fixed on its procedures. Every time a test scenario is executed, it uses the same input data. Keeping a map to store data generated during execution and to maintain input data is key to our approach as it enables one *atom* to exercise different inputs and to consume data another *atom* produces.

Our solution for the data map consists in a Java HashMap [6] that is kept in memory during the *atoms* execution. This map associates a list of objects to each input category (the map key). Thus, we can group objects of the same category under a key that represents this category. For instance, the key *playlists* groups objects that can be used in a test as names for playlists. In order to fill our map with input data, we use TAF database files, called TDF files [28]. In addition to configuration settings, TDF files keep data to be used as input during testing executions. They are XML files and may have different structures to expose their contents.

```
<CONSTANT name="PLAYLIST_01" description="Name of a playlist.">
    <APPLICATION_CONTENT>
        <VALID>
            <FOR id="*"/>
        </VALID>
        <CONTENT source="plain" value="playlist01"/>
    </APPLICATION_CONTENT>
</CONSTANT>
<CONSTANT name="PLAYLIST_02" description="Name of a playlist.">
    <APPLICATION_CONTENT>
        <VALID>
            <FOR id="*"/>
        </VALID>
        <CONTENT source="plain" value="playlist02"/>
    </APPLICATION_CONTENT>
</CONSTANT>
```

Figure 3.10 Example of TDF items.

Figure 3.10 gives an example of TDF file's items. It shows contents that can be used to give names to playlists. As we are only interested in input data, we must not consider TDF items that correspond to configuration settings. We differentiate data items from configuration settings by the attribute *source*, which must have *plain* as its value. Any other items that have a value for their *source* attribute different from *plain* do not correspond to a data item. So, no matter how the TDF files are structured, if we can find items that have a *source* attribute with value *plain*, we can extract test input data from them. From the TDF items Figure 3.10 shows, we can extract two playlist names: *playlist01* and *playlist02*, which are related to the attribute *value* of the tags *CONTENT* present in lines 6 and 14, respectively.

Figure 3.11 illustrates an overview of the process of filling the data map with input data from TDF files. An engine (*Parser*) receives as input a set of TDF files (*Configuration Files*). Then the engine looks for data items to extract input data from them. After that, all data are stored in a HashMap (our *Test Data Map*) under keys that correspond to their categories. For instance, all data that correspond to playlist names must be kept in the map under the key *playlists*. Thus, each key groups a list of objects of a same category.

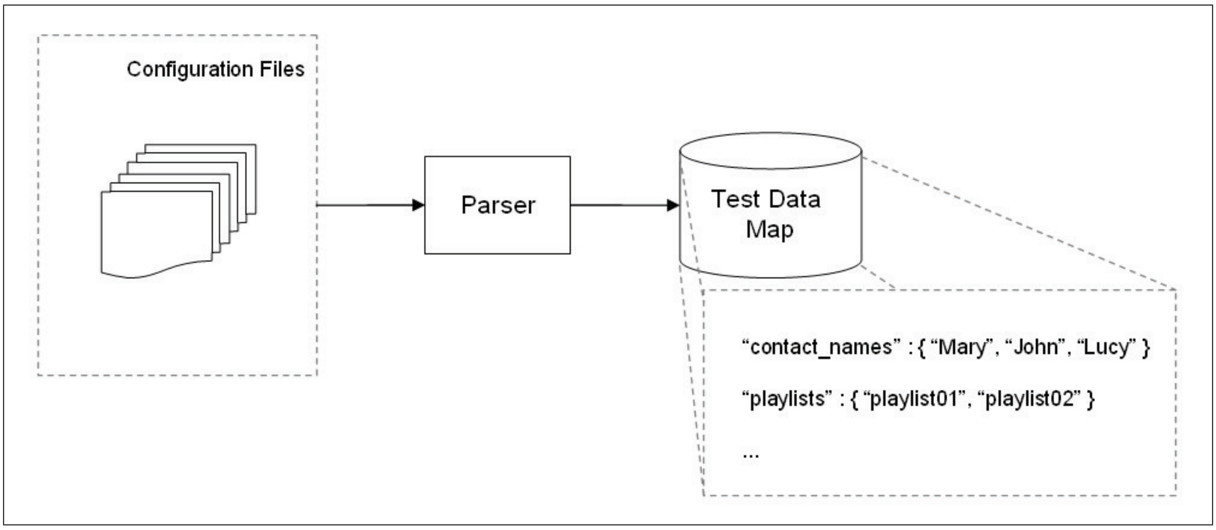

Figure 3.11 Filling the test data map with input data.

In order to ease the recognition of objects of a same category by the *Parser*, we assume that the value for the *name* attribute of a TDF data item follows this pattern: *CATEGORY_NAME*, where *CATEGORY* represents the category of the item, and *NAME* is a **unique** string to distinguish the item among others of the same category. Thus, all items that have the same prefix (*CATEGORY*) in the value of their *name* attribute belong to a same category.

Once the Test Data Map is filled with data extracted from TDF files, it can provide input data for *atoms*. Thus, an *atom* may read data from or update the map with data it generates during execution. In order to do so, it first needs to get an instance of the Test Data Map (at this time, the data map is already filled with input data) and then access this instance to read one object from (*getOneObjectRandomly*), or update the map to remove (*removeObjectFromKeyRandomly*) or to add one object (*addObjectToMap*). The first two methods use a seed, defined by the user, to randomly access the map. In the following, we enumerate some tips to help the addition of the data map support in the *atoms* creation.

Tips for including data map support in *atoms*:

[Tip D] We first need to get an instance of the data map, i.e. add this line in the beginning of the procedures of all *atoms*: *TestDataMap tdm = TestDataMap.getTestDataMap();*. During execution, if it is the first time the data map is accessed, a new Test Data Map instance is created and filled with data extracted from TDF files. Otherwise, an instance of the already filled data map is used to guarantee that the map is updated with the changes previous *atoms* have made.

[Tip E] Then, verify if the *atom* needs input data. These input data can be either (i) a parameter that helps the creation of an object (an example of this can be found in the line 9 of Figure 3.6 where to create a playlist it uses the name *PLAYLIST_02*), or (ii) an object that is manipulated by the *atom* but was not created by the *atom* (as in line 10 of Figure 3.7, which plays a playlist not created by the *atom*; and line 10 of Figure 3.9, which deletes a picture that the *atom* did not create either).

```
public class ATOM_CREATE_PLAYLIST {

  public ATOM_CREATE_PLAYLIST() {}

  public void setup() {}

  public void buildProcedures() {
      log("Create a playlist.");
      TestDataMap tdm = TestDataMap.getTestDataMap();
  --> String name = (String) tdm.getOneObjectRandomly(TestDataMapConstants.PLAYLISTS);
  --> stressTk.createRandomFilesPlaylist(Playlist.get(name), 3, 10);
      tdm.addObjectToMap(TestDataMapConstants.PLAYLISTS_CREATED, name);
  }

  public void tearDown() {}

}
```

Figure 3.12 Atom #1 with data map support.

[Tip F] If the input data the *atom* needs is related to [Tip E] (i), then retrieve from the map an object that correspond to the desired parameter's category. Use method *getOneObjectRandomly* to do so. After that, the next UFs in the *atom* can use the retrieved object. In Figure 3.12 we show the use of Tip F to retrieve from the data map the input *Atom #1* needs to create a playlist. We created a class with constants that are related to all keys used in the map (*TestDataMapConstants*). Thus, instead of calling directly the desired map's key, we call its corresponding constant (in this case, *PLAYLISTS*). This helps the maintenance of the code and avoids spelling errors.

[Tip G] If the input the *atom* needs is related to [Tip E] (ii) and it is simply used by the procedures of the *atom* but it is not removed from the phone file system, also use method *getOneObjectRandomly*, indicating the key that correspond to the category of the object to be used. As an example for the use of this tip, we can see Figure 3.13, which shows *Atom #2*. The *atom* needs to retrieve a playlist from the data map in order to play it. The arrows on the lefts indicate the exactly point where we followed Tip G. It first gets from the data map an object (the name of a playlist) under the key *PLAYLISTS_CREATED* (line 10) and uses it to select the playlist to be played (line 12).

```
1  public class ATOM_PLAY_PLAYLIST {
2
3    public ATOM_PLAY_PLAYLIST() {}
4
5    public void setup() {}
6
7    public void buildProcedures() {
8        log("Play the created playlist.");
9        TestDataMap tdm = TestDataMap.getTestDataMap();
10   --> String name = (String) tdm.getOneObjectRandomly(TestDataMapConstants.PLAYLISTS_CREATED);
11       navigationTk.goTo(PhoneApplication.get("MUSIC"), MultimediaGoTo.get("PLAYLIST"));
12   --> multimediaTk.scrollToAndSelectMultimediaFile(Playlist.get(name));
13       navigationTk.goToAndSelectMenuItem(MultimediaItem.get("PLAY"));
14   }
15
16   public void tearDown() {}
17
18 }
```

Figure 3.13 Atom #2 with data map support.

[Tip H] If the input the *atom* needs is related to (ii) and it is consumed by any TAF UF that deletes it from the phone file system, then it also needs to be removed from the data map. Use method *removeObjectFromKeyRandomly*, indicating the key that correspond to the category of the object to be deleted. Figure 3.14 shows *Atom #4* after the use of this tip. We noticed that the *atom* needs as input a picture object to be deleted (Tip E). So we followed Tip H to randomly remove one picture among all pictures stored in the data map (line 10). After that, the picture removed from the data map is also removed from the phone file system (line 12).

[Tip I] Finally, verify if the *atom* generates data, i.e. creates any object. The creation of the object may be explicit in the code as in line 11 of Figure 3.8, which creates a *PictureFile* object; or implicit as in line 11 of Figure 3.12, which creates a playlist in the phone file system but does not create an object in the code that represents the playlist. If the *atom* explicitly creates an object, the object created must be added to the data map. But if the object created by the *atom* is not explicitly shown in the code, what must be added to the data map is one of its attribute. Use the one that helped the creation of the object (for instance, the name of the

```
public class ATOM_DELETE_PICTURE {

  public ATOM_DELETE_PICTURE() { super(1); }

  public void setup() {}

  public void buildProcedures() {
      log("Delete a picture.");
      TestDataMap tdm = TestDataMap.getTestDataMap();
  --> PictureFile pic = (PictureFile) tdm.removeObjectFromKeyRandomly(TestDataMapConstants.
      PICTURES)
      navigationTk.goTo(PhoneApplication.get("PICTURES"), MultimediaGoTo.get("ALL_PICTURES"));
  --> multimediaTk.deleteFile(pic, true);
  }

  public void tearDown() {}

}
```

Figure 3.14 Atom #4 with data map support.

playlist created by *Atom #1* in Figure 3.12). The object created or the attribute must then be added to the data map under the key that corresponds to the category of the object created. Use the *addObjectToMap* method.

Atom #3 (Figure 3.15) takes a picture from the camera and stores it in the phone file system, i.e. its procedures creates an object (a picture). This situation is reported in Tip I. Since the object created is explicitly shown in the code, i.e. a *PictureFile* is created (line 12), we add it to the data map we instantiated in the beginning of the procedures (Tip D). The picture is added to the data map under the key that corresponds to the pictures category – *PICTURES* (line 13).

With the help of the tips we presented here, we completed the creation of the 4 *atoms* (showed in Section 3.1) and added the data map support. Figures 3.12 to 3.15 illustrated them.

```
1  public class ATOM_TAKE_PICTURE {
2
3    public ATOM_TAKE_PICTURE() {}
4
5    public void setup() {}
6
7    public void buildProcedures() {
8        log("Take a picture.");
9        TestDataMap tdm = TestDataMap.getTestDataMap();
10       navigationTk.launchApp(PhoneApplication.get("CAMERA"));
11       multimediaTk.capturePictureFromCamera();
12   --> PictureFile pic = multimediaTk.storeCapturedPictureAs(MultimediaItem.get("STORE_ONLY"));
13   --> tdm.addObjectToMap(TestDataMapConstants.PICTURES, pic);
14   }
15
16   public void tearDown() {}
17
18 }
```

Figure 3.15 Atom #3 with data map support.

3.3 Execution Lists Generation

After the creation of *atoms*, we organize them in sequence to build what we call execution lists. These execution lists have the objective to guide the *atoms* execution. Thus, during execution time, each *atom* is executed in the order it appears in the execution list.

AF defines two strategies to group *atoms* in execution lists: one that randomly organizes the available *atoms* (Section 3.3.1); and another that also organizes *atoms* but in a more systematic way (Section 3.3.2). In both strategies, the size of the generated execution lists can be defined by the user or determined by the number of the available *atoms* to be added to the resulting lists.

3.3.1 Random Generation Algorithm

One AF strategy to generate execution lists is by randomly ordering *atoms* in sequences. Algorithm 1 shows the pseudo-code for the random generation. Function *genList* receives as input a set of *atoms* (*allAtoms*), a number that indicates the size of the execution list to be generated (*size*), and a seed for the random choice of *atoms* (*seed*). Its output is a list of *atoms* that represents the execution list generated (*execList*).

Algorithm 1: genList for Random Generation

```
1 genList(Set⟨Atom⟩ allAtoms, int size, long seed): List⟨Atom⟩
2 begin genList
3     List⟨Atom⟩ execList = ∅ ;
4     for i = 1..size do
5         Atom atomChosen = pickOne(allAtoms, seed);
6         execList.add(atomChosen);
7     endfor
8     return execList;
9 end
```

The code fragment within the range 4 – 7 randomly selects a number (*size*) of *atoms* from *allAtoms* and adds them to the resulting list (*execList*). The selection of *atoms* is done by function *pickOne* that uses the random *seed* to randomly select one *atom* from the set *allAtoms*. It only selects one *atom*, but does not remove it from the set. So, each call to function *pickOne* will have the same input set of *atoms* as in the beginning of the generation. Figure 3.16 illustrates this random generation.

As in the example showed in Figure 3.16, the random algorithm does not guarantee that all available *atoms* will appear at least once in the execution list it generates. In the example,

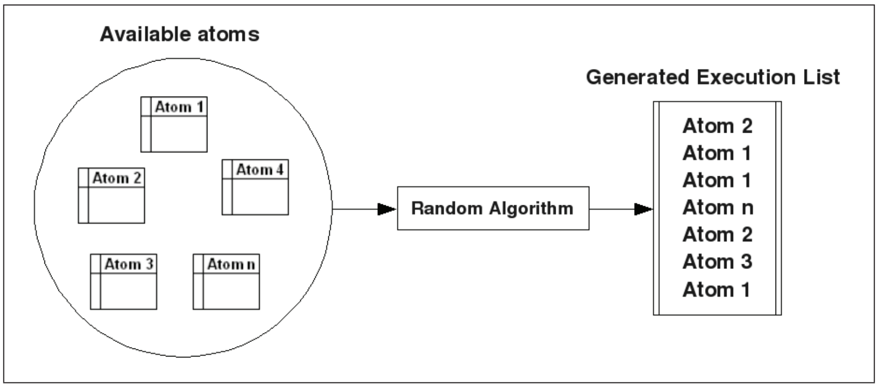

Figure 3.16 Generation of an execution list using the Random Algorithm.

Atom 4 is not in the resulting list. What we have noticed though is that the longer the resulting execution list is, the more uniform the frequency of *atoms* becomes.

The execution lists randomly generated also does not guarantee that all *atoms* will execute correctly. This happens when the execution of an *atom* requests inputs from the data map in order to run its procedures. If there are no compatible values in the map, we say that the *atom* fails, i.e. it does not execute (its procedures are not sent to the system under test), and the following *atom* in the execution list takes place.

3.3.2 Pre and Postconditions Check

Because we noticed that, with the random generation, *atoms* can be discarded from the execution list due to lack of values in the data map, in this work we propose a more systematic strategy to generate execution lists. Its main objective is to guarantee that every *atom* in the list will always have values available in the data map corresponding to the categories of the inputs its procedures request.

In this strategy, each *atom* has pre and postconditions. The preconditions indicate which data the *atom* requires as input for its procedures, and the postconditions represent any data the *atom* creates/updates/deletes. In order to check the pre and postconditions, we use an abstract view of all data that should be present in the system during the *atoms* execution. While an execution list is generated, this abstract view is gradually updated with information about the quantity of each object that is created/updated/deleted by the *atoms* postconditions.

The idea is to build execution lists by checking the pre and postconditions without executing the *atoms*. In order to make it possible and closer to what would happen during the real execution of the *atoms*, their pre and postconditions must be complete. That is, the preconditions must represent all need for data their corresponding *atom* has; and the postconditions need to be aware of all objects created/updated/deleted by the *atom*'s procedures. To make it clearer, let's take a look in some examples.

Figure 3.17 shows the procedures of an *atom* that creates a picture album with all available pictures in the phone file system. It first gets a name from the data map for the picture album it will create (line 10). Then it creates a picture album with the name previously chosen (line 11). And finally, it adds the name of the created picture album to the data map (line 12).

```
1  @preconditions({
2    @precondition("PICTURE_ALBUMS_NAMES > 0"),
3    @precondition("PICTURES > 0")
4  })
5  @postcondition("PICTURE_ALBUMS++")
6  public void buildProcedures() {
7
8    log("Create a picture album.");
9    TestDataMap tdm = TestDataMap.getTestDataMap();
10   String name = (String) tdm.getOneObjectRandomly(TestDataMapConstants.PICTURE_ALBUMS_NAMES);
11   multimediaTk.createPictureAlbumWithAllFiles(PictureFile.get(name));
12   tdm.addObjectToMap(TestDataMapConstants.PICTURE_ALBUMS, name);
13
14 }
```

Figure 3.17 Atom that creates a picture album.

Note that to create a picture album, the *atom* needs to retrieve a name from the data map and also there must be at least one picture in the phone file system. If there are no names and/or no pictures in the data map, the *atom* does not create the picture album, i.e. it fails. From this we can say that the *atom*'s preconditions state that there must be at least one name for picture albums (line 2) and at least one picture stored in the data map (line 3). After the creation of the picture album, the *atom* adds the album's name to the data map to keep track of the object created. So, the *atom*'s postcondition in this case states that the data map will be updated with one more picture album (line 5).

Sometimes, the pre and postconditions are simply "true". For instance, Figure 3.18 shows an *atom* that takes a picture without saving it in the phone file system. Therefore, it does not need any input data to execute its procedures, as there is no need for data to take a picture; and it does not creates any object in the phone, as it does not save the picture taken. This means we can add the *atom* to an execution list anywhere, i.e. there is no restriction of data that disables its execution and it will be always ready to execute whatever the state of the data map is.

As can be noticed by the pre and postconditions we exemplified here, the abstract view of the system we previously described must have the information regarding the access to the data map by each *atom*. Thus, before adding an *atom* to an execution list under construction, its preconditions are checked against a representation of the data map. In the beginning of the

```java
@precondition("true")
@postcondition("true")
public void buildProcedures() {

  log("Launch camera and take a picture.");
  navigationTk.launchApp(PhoneApplication.get("CAMERA"));
  multimediaTk.capturePictureFromCamera();

}
```

Figure 3.18 Atom that takes a picture without saving it.

execution list's creation, the *atom*'s preconditions are checked against an initial state of the data map, which can be determined by the user. After that, the data map is updated with the *atom*'s postcondition. Finally, this process is repeated until the execution list reaches the desired size or no more *atoms*'s preconditions are satisfied by the current state of the data map.

Algorithm 2 shows a pseudo-code for the execution list generation with pre and postconditions checks. The signature of function *genList* is the same as the one for the random generation. It receives a set of *atoms* (*allAtoms*), the desired size for the execution list to be generated (*size*), and a seed for the random choice of *atoms* (*seed*). The output is a list of *atoms* that represents the execution list generated (*execList*).

Algorithm 2: genList with pre and postconditions check

```
genList(Set⟨Atom⟩ allAtoms, int size, long seed): List⟨Atom⟩
begin genList
    List⟨Atom⟩ execList = ∅ ;
    Map⟨String, Integer⟩ currentState = buildMap();
    for i = 1..size do
        List⟨Atom⟩ satAtoms = select(allAtoms, currentState);
        if satAtoms.isEmpty() then break;
        Atom atomChosen = pickOne(satAtoms, seed);
        execList.add(atomChosen);
        currentState = executePost(atomChosen);
    endfor
    return execList;
end
```

The implementation of function *genList* is different though. It first calls function *buildMap* that creates a representation of the data map (*currentState*). This version of the data map is different from the one used by our *atoms*. In this case, *currentState* is a map, whose keys are the categories of the *atoms*' inputs. Each key maps to the number of objects that would be present in the test data map during the execution of *atoms*. This representation of the data map can be initially loaded with a set of input data defined by the user. This helps the creation of execution lists for different initial configurations of the system to be tested.

The code fragment within the range 5 – 11 builds the resulting execution list. The algorithm uses function *select* to pick a subset (*satAtoms*) of *allAtoms* whose preconditions are satisfied by the data present in *currentState* (line 6). For instance, suppose that there are no data stored in *currentState* – the data map is empty. This means that only *atoms*, whose have preconditions are "true", i.e. no need for input data, will be added to *satAtoms*.

If after the selection of *atoms*, the subset *satAtoms* is empty, the generation of the execution list ends (line 7). This means that no more *atoms* can be added to *satAtoms* because their preconditions are not compatible to the data stored in *currentState*, i.e. the data map does not contain the input data *allAtoms* require. Otherwise, when *satAtoms* is not empty, the generation continues. Function *pickOne* randomly selects one *atom* from *satAtoms* (line 8). The building execution list then receives the selected *atom* (line 9). And finally, function *executePost* updates *currentState* with the *atom*'s postconditions (line 10). For instance, if the postconditions state that the selected *atom* creates an object, *executePost* gets the key that corresponds to the category of the object created and increments the number it maps.

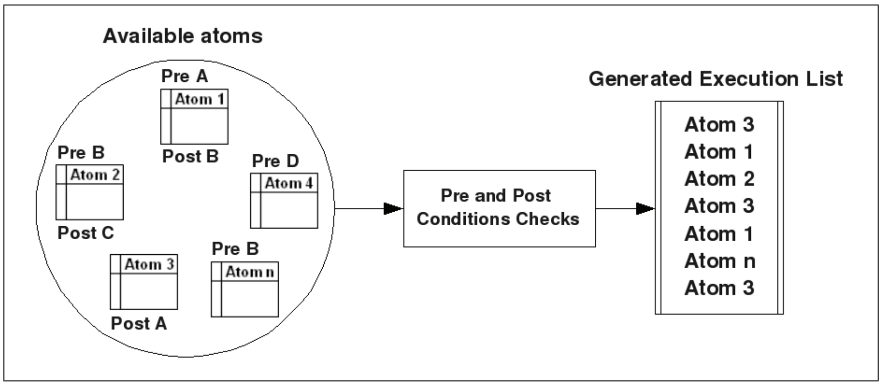

Figure 3.19 Generation of an execution list using pre and postconditions check.

Figure 3.19 illustrates the generation with pre and postconditions check. Differently from the *atoms* showed in Figure 3.16, all *atoms* here have pre and postconditions. To ease comprehension, we assumed that the initial state of the data map is empty, i.e. with no data stored. When the Pre and Postcondition Check Algorithm starts, it adds *Atom* 3 to the building execution list, as it is the only one with "true" preconditions. Then, it updates the data map with *Atom* 3's postcondition *Post A*. With the data map updated with *Post A*, now the algorithm can randomly choose either *Atom* 3 again or *Atom* 1, whose precondition is *Pre A* (suppose that *Pre A* requires the data *Post A* indicates). In our example, the algorithm adds *Atom* 1 to the execution list and updates the data map with its postcondition *Post B*. The algorithm then adds *atoms* to the resulting execution list until it reaches the desired size.

The Pre and Postconditions Check Algorithm also does not guarantee that all available *atoms* will appear at least once in the generated execution list. In the example, *Atom 4* is not present. This may happen either due to lack of data in the data map that satisfy some *atoms*' preconditions or due to the random selection when there are more than one *atom* whose preconditions are satisfied by the available data in the map. On the other hand, all *atoms* added to the execution list will execute correctly, as the data map contains the inputs they require.

CHAPTER 4
Empirical Evaluations

Testing is a skill. While this may come as a surprise to some people it is a simple fact.
—M. FEWSTER AND D. GRAHAM (Software Test Automation)

This chapter provides details on the empirical evaluation of the Atoms Framework using Motorola cellular phones. We conducted 4 sets of empirical experiments. Firstly, we compared the use of *atoms* with the use of regular test scenarios with respect to their capability to crash phones with known bugs (Section 4.3). In the other two experiments we evaluated the impact of randomization for data (Section 4.4) and sequence generation (Section 4.5) on AF. And in the final experiment (Section 4.6) we examined the impact of pre and postconditions check in the execution lists generation. We reported the majority of the experiments in [13].

4.1 Characterization of Subjects

We characterize each subject with (i) the phone model (i.e., a list of external and internal phone features to identify a set of similar phones functions), (ii) the hardware version, (iii) the software version (i.e., the build for the operating system and its applications), and (iv) the flex bit (FB) version. The flex bit configuration allows the user to dynamically configure the phone prior to the tests. Example of such configurations includes enabling the phone to send and receive bluetooth signals, and setting the phone to debug mode.

Table 4.1.1 shows the subjects we used in our experiments. Column "Config." introduces a unique identifier to distinguish each combination of model, hardware, software and flex bit. The other columns show each of these attributes. The identifiers we use in this table are artificial. Note that some configurations share the same model or hardware, but the software and flex bit vary. The selection of these configurations was driven by the availability of equipment where past errors have been detected.

Config.	Model	Hard.	Soft.	FB
A	M1	H3	S1	F1
B	M1	H4	S2	F2
C	M1	H4	S3	F3
D	M2	H2	S4	F4
E	M2	H1	S5	F5
F	M3	H5	S6	F6
G	M3	H6	S7	F7
H	M2	H2	S8	F8

Table 4.1.1 Characterization of experimental subjects.

4.2 Failures

The oracle does not operate on the phone. It is a general Motorola proprietary program that monitors the phone memory for bad states.

The oracle detects 12 distinct kinds of crashes across all experiments. In the following, we distinguish them using crash identifiers (CIDs) from 1 to 12. Each identifier denotes a different undesirable scenario of the application that the oracle is able to capture. For example, CID=1 is a general report to denote that the system makes no progress but the oracle is unable to ascertain the reason, CID=2 means that an issue with the hardware interface (e.g., it is not possible to allocate memory) prevents the application from making progress, CID=6 denotes a programming error like division by zero, etc. Important to note is that the oracle reports only the crash event; it does not inform the reason for the crash (as debuggers do). Consequently, it may happen that distinct techniques report different manifestations (CIDs) of the same defect.

4.3 Experiment I: Atoms *versus* Test Scenarios

This section describes the experiment we conducted to compare AF with a technique that uses test scenarios to find crashes on phones. This technique is called Scenario Hunting (SH) [13] and we use it as baseline for this experiment. To this date, the Motorola Test Center in Recife (Brazil) uses SH among other techniques for crashing phones.

Setup. The goal of this experiment is to compare the effectiveness of AF with that of SH for crashing phones with historical defects. We used 8 different phone configurations for which SH found 4 crashes. For each configuration we run once each technique until execution runs out of time (timeout=40h) or finds a crash. The execution of SH confirmed the crashes documented in the bug report database. For AF, we used the Random Generation Algorithm (see Section 3.3.1) to generate all execution lists used and we fixed the random seed for data generation across different configuration runs. The *atoms* that AF uses derive from the test scenarios SH used – we did not include any *atoms* original from a different set of tests. This helps us to fairly compare SH and AF.

Results. Table 4.3.1 shows a summary of the results obtained in the experiments. Column "Config." shows the identifier for one subject configuration, column "CID" shows the identifier of the crash, and column "time" shows the execution time (in hours) for each experiment. Recall that one experiment either timeouts or finds a crash. Line "avg." reports the averages of each column. For column CID, it shows the fraction of experiments that revealed a crash. For column time, it shows the arithmetic mean of the elapsed time.

Config.	SH		AF	
-	CID	time	CID	time
A	-	40.0	6	6.5
B	-	40.0	4	33.6
C	-	40.0	4	36.5
D	2	4.3	7	5.0
E	3	3.9	1	3.6
F	-	40.0	1	4.0
G	4	3.1	6	11.2
H	5	2.3	5	2.8
avg.	50%	21.7	100%	12.9

Table 4.3.1 Comparison between SH and AF.

We list next our key observations:

- *Precision.* Only AF could find a crash for all eight experiments with a timeout of 40h. SH found a crash in only 50% of the cases. But when it found, it outperformed AF, except in Config. E where the difference was of only 0.3h (that is, 18min).
- *Time.* The variation of time reported was not very high when both techniques found a crash. For example, the difference in time for crash for experiment D was +0.7h (i.e., AF took 0.7h more to find the crash), -0.3h for experiment E, 0.5h for H. This difference was high only for experiment G, where AF took 8.1h more to find the crash.
- *Kind of crash.* The variation of the CID reported was high. Note that only in experiment H both techniques reported the same CID.

4.4 Experiment II: Impact of Randomization of Data

This section describes the experiment we conducted to evaluate the impact that the use of different random data has in the effectiveness of AF.

Setup. In this experiment, we run AF for 10 times, on each configuration. We used the same sequence of *atoms* in all executions of a configuration (i.e., we fixed the values of the random seed for the generation of execution lists), however it varied across different configurations. We used the Random Generation Algorithm (see Section 3.3.1). Figure 4.1 shows the distributions of execution time (in hours) for this experiment.

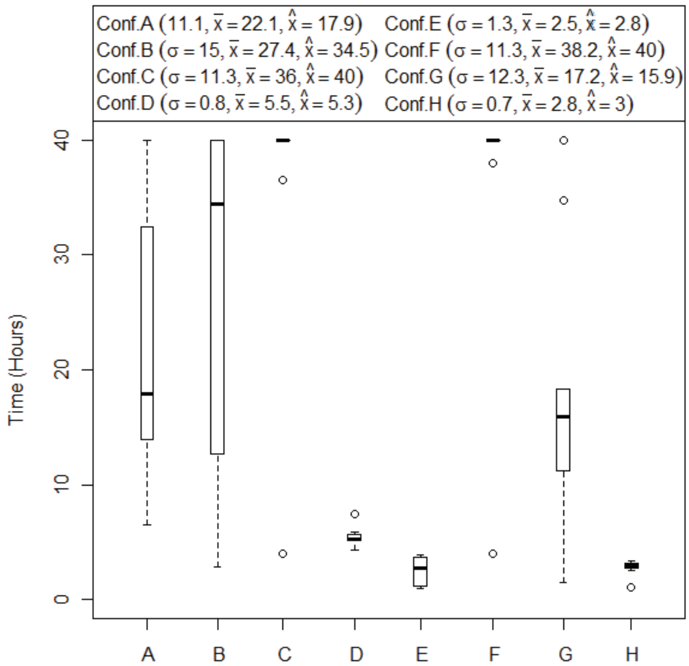

Figure 4.1 AF time distributions for random data.

Note on distribution representation. We use box-plot notation to illustrate a data distribution. The lower and upper hinges of one box indicate respectively the upper bounds of the first and third quartiles of the distribution; the line across the box defines the second quartile (i.e., median value). The lines below and above the box limit the first and fourth quartiles. Small circles outside the hinges correspond to outliers. The symbol $\bar{x}$ denotes the mean value, the symbol σ denotes the standard deviation – an average for the dispersion of data points from the mean value, and the symbol $\hat{x}$ denotes the median value.

Results. Table 4.4.1 shows detailed data for the 10 runs of AF over each configuration from A to H. The value “-” for column “CID” indicates a missed crash (resp., value “40.0” for column “time” indicates a timeout). For example, for configuration A, AF misses the crash on experiment 4.

	Config. A		Config. B		Config. C		Config. D		Config. E		Config. F		Config. G		Config. H	
#	CID	time	CID	time	CID	time	CID	time	CID	time	CID	time	CID	time	CID	time
1	6	6.5	4	33.6	4	36.5	7	5.1	1	3.6	1	4.0	6	11.2	5	2.7
2	6	14.6	-	40.0	-	40.0	7	5.5	5	3.6	-	40.0	6	15.1	5	3.4
3	6	13.9	6	37.8	-	40.0	7	5.3	6	1.9	-	40.0	4	18.3	5	3.3
4	-	40.0	-	40.0	-	40.0	7	5.7	5	3.9	-	40.0	4	16.6	5	3.1
5	4	33.9	6	4.5	12	4.0	7	5.9	6	1.2	-	40.0	-	40.0	5	2.9
6	9	19.0	4	27.2	-	40.0	7	5.1	5	3.7	12	37.8	4	34.7	9	2.9
7	4	30.5	-	40.0	-	40.0	7	7.4	5	1.0	-	40.0	4	17.6	5	2.5
8	6	16.7	6	12.7	-	40.0	7	5.3	9	1.5	-	40.0	4	1.5	5	3.0
9	4	32.4	6	35.3	-	40.0	7	4.9	9	1.1	-	40.0	9	15.2	5	3.2
10	6	13.4	6	2.9	-	40.0	7	4.3	5	3.9	-	40.0	6	1.6	5	1.1
avg.	90%	22.1	70%	27.4	20%	36.1	**100%**	5.5	**100%**	2.5	20%	36.2	90%	17.2	**100%**	2.8

Table 4.4.1 Impact of using random seeds in AF.

We list next our key observations:

- *Precision.* Although some executions did not find a crash, the mean precision – the value of the avg. CID for each configuration – was high: 74%.
- *Time.* The dispersion of the data points in AF was high for almost all configurations. The median standard deviation of execution times for all configurations was 7.9h. That means that AF execution time was very sensitive to the selection of the seed.
- *Kind of crash.* Only Config D found the same kind of crash in all executions. All others configurations found more than 2 types of crash (for instance, Config. E found 4 different types).

4.5 Experiment III: Impact of Randomization of Sequences

This section describes the experiment we conducted to evaluate the impact of randomizing the list of *atoms* used by AF.

Setup. We run AF for 10 times on phone configurations E and H. These configurations have the lowest average execution times (see Table 4.4.1). We used the same random seed for data and execution list generation in all runs.

In this experiment, we did not use the Random Generation Algorithm or the Pre and Post-conditions Check Algorithm for the execution lists generation we described in Chapter 3. Instead, we used the one Algorithm 3 shows. This version associates each *atom* to one domain category. The categories we defined are as follow: applaunch (*atoms* that only go to an application), browser (access the Internet), mms (deal with multimedia messages), multimedia (deal with multimedia files and camera), phonebook (deal with calendar, events and contacts), and sms (deal with text messages). Function *partition* in line 3 takes as input a user-defined set of *atoms* (*allAtoms*) and returns a map that associates a set including all *atoms* of a category with the category it belongs.

Algorithm 3: genList with Allpairs

```
1  genList(Set⟨Atom⟩ allAtoms, int nAtoms, long seed): List⟨Atom⟩
2  begin genList
3      Map⟨Category, Set⟨Atom⟩⟩ partition = partition(allAtoms);
4      Map⟨Category, Set⟨Atom⟩⟩ selected = ∅;
5      foreach entry in partition do
6          Set⟨Atom⟩ atoms = entry.value();
7          Set⟨Atom⟩ tmp = ∅;
8          for i = 1..nAtoms do tmp = tmp ∪ atoms.pickOne(seed);
9          selected.put(entry.key(), tmp);
10     endfch
       /*concatenates all sequences of atoms.  each sequence includes 1 atom on each
       category*/
11     return allpairs(selected);
12 end
```

The code fragment in the line range 5 – 10 selects *nAtoms* on each category and assigns the resulting map to variable *selected*. We apply pairwise coverage [34] to generate sequences of *atoms* with the property that each *atom* of each category is paired to another *atom* of another category in at least one case. For this, we use the Allpairs [5] tool. Finally, it gives as output sequences of *atoms* (each sequence includes one atom of each category), and then concatenates these sequences to build one longer sequence AF executes, i.e. an execution list.

Results. Table 4.5.1 shows detailed data for the 10 runs of AF over configurations E and H. Our key observations for this experiment are as follows:

- *Precision.* For both configurations, AF found a crash in only 4 out of 10 runs, while AF found a crash for 100% of the cases when using *atoms* derived from existing tests (see Table 4.4.1).

- *Time.* The average time that Config. E took to find a crash (5.5h) was higher than the one for Experiment II (2.5h, see Table 4.4.1). The average time that Config. H took to find a crash (3.8h) was also higher than the one reported in Experiment II (2.8h). However, if we only consider the runs that found a crash in Config. H, AF performed faster in this experiment. Its slowest time was 2.2h, while Experiment II reported 3.4h as the slowest time to find a crash for Config. H.

- *Kind of crash.* The types of crash reported for both configurations were the same in all runs that found a crash.

This experiment indicates that the interaction between categories (functionalities of the system) may not be as important as the selection of critical *atoms*.

	Config. E		Config. H	
#	CID	time	CID	time
1	-	5.5	5	1.6
2	5	2.4	-	5.0
3	5	5.3	-	5.4
4	-	10.4	5	1.3
5	-	6.2	-	6.7
6	-	4.9	5	1.1
7	5	3.1	-	4.3
8	5	6.7	-	5.5
9	-	5.4	-	5.4
10	-	5.5	5	2.2
avg.	40%	5.5	40%	3.8

Table 4.5.1 Runs of AF with different execution lists for configurations E and H.

4.6 Experiment IV: Impact of Pre and Postconditions Check

This section describes the experiment we conducted to evaluate the impact the pre and post-conditions check has on the effectiveness of AF to find crashes.

Setup. We run AF for 10 times on phone configurations E and H. These configurations have the lowest average execution times (see Table 4.4.1). We used the same random seed for data generation in all runs. We varied the execution lists in all executions of the same configuration, i.e. we varied the value of the random seed for the generation of execution lists. However, we fixed it across different configurations. We used the Pre and Postconditions Check Algorithm (see Section 3.3.2) to generate the execution lists uses.

Results. Table 4.6.1 shows detailed data for the 10 runs of AF over each configuration.

	Config. E		Config. H	
#	CID	time	CID	time
1	5	15.6	5	1.9
2	5	15.4	5	1.5
3	5	20.9	5	2.9
4	5	22.8	5	1.1
5	5	19.3	5	1.8
6	5	16.9	5	1.9
7	5	17.0	9	1.3
8	5	37.2	5	1.1
9	5	25.4	5	1.7
10	5	17.8	5	1.8
avg.	100%	20.8	100%	1.7

Table 4.6.1 Runs of AF with pre and postconditions checks for configurations E and H

Our key observations for this experiment are as follows:

- *Precision.* All runs of both configurations found a crash. The precision achieved a 100%.
- *Time.* When compared to Experiment III (see Table 4.5.1), in this experiment AF took longer to find a crash in Config. E. The average time was 20.8h against 5.5h of Experiment III. However, it was faster for Config. H, i.e. the average time here was 1.7h against 3.8h for Experiment III.
- *Kind of crash.* Just like Experiment III, AF reported the same kind of crash in all runs for both configurations (CID=5), except in run 7 of Config. H (CID=9).

4.7 Threats to Validity

This section describes threats to internal and external validity of our experiments. **Internal validity** determines whether the techniques have a cause-and-effect relationship in the experimental observations. **External validity** determines whether or not one can generalize the experimental observations to other scenarios. One threat to internal validity is internal randomness. In principle, it is possible that the system does not answer promptly to the commands that the automated test issues. This depends on the operating system's scheduling decisions. This effect could impact our observations. One threat to external validity is portability of the technique. We implemented AF with the goal of testing cellular phones, although in principle, there is no reason to believe that it is not applicable to other kinds of application.

Chapter 5
Conclusions

My conclusion is that testing is an intellectual endeavor and not part of arts and crafts.

—JAMES A. WHITTAKER (How to Break Software)

Software has tremendously increased in complexity over the past decades. We now have software applications for almost everything. As a result, the size of software products is no longer measured in terms of thousands of lines of code, but millions of lines of code. If we do not give much attention to a proper development process, which focuses on the production of software with quality and reliability, this huge number of lines of code becomes a perfect environment for the insertion of bugs. Software failure is expensive. The longer a bug stays in the program, the more costly it becomes to fix it. And software with bugs is software with no quality, no competition in the market, and not attractive to customers.

Software testing then becomes very important to decrease the number of bugs and increase the quality of the product we deliver to our clients. And test automation even more. An increased confidence in the quality of the final product emerges with automated tests. Proper automation can also increase the effectiveness and efficiency of the overall testing process [41], reducing costs for the entire project.

This dissertation proposed a black-box testing technique - the Atoms Framework (AF) - to help the detection of crashes. A crash is a system malfunction caused by faulty software or hardware, that makes the system either partially or totally inoperable. A crash is detected, for instance, when a test exposes a bug and the system stops responding to the testing procedures. Finding crashes is important as it leads the system to an abnormal termination, whose causes are manifold, such as data corruption, deadlock, and access violations. But to find a crash is often non-trivial: the search space is typically intractable (specially for system testing).

We use the term test scenario to denote a sequence of executable steps that have been manually written. The idea is to fragment test scenarios into small units, which we call *atoms*, and compose them in different ways. Along with this, AF adds support for test data and the generation of execution lists (whose goal is to guide the *atoms* execution). Thus, AF allows the exercise of different inputs (which is a key ingredient to AF as it enables one *atom* to exercise different inputs every time it is executed) and sequences of steps (due to the ordering of the *atoms* in the execution lists), which results in the exploration of new test scenarios in each execution.

This work was part of a research effort between Motorola and the Informatics Center/UFPE for improving Motorola's software testing process. Thus, we could evaluate AF on Motorola's cellular phones. We conducted a set of experiments to quantify the execution time and the capability of the technique to find a crash. We used 8 different phone configurations with historical (real) errors. We firstly compared AF to a technique that uses the same test scenarios AF receives as input. This technique is currently in use in Motorola and it is called Scenario Hunting (SH). In this first experiment, AF found crashes in all 8 configurations, while SH found in 4. However, AF was slower when both techniques found a crash.

We then conducted three other experiments to give us a better understanding of the effectiveness of AF. We evaluated the impact of the use of (i) random data, (ii) random generated execution lists and (iii) a more systematic generation of execution lists, based on pre and post-conditions checks. In experiment (i), we run 10 times each configuration, varying the random seed for data selection. The results revealed a high variance of execution time: almost 8 hours. We believe that this high variance is due to the size of the inputs the random selection of data gave to the *atoms*. For instance, in the context of cellular phones, loading large pictures or videos in the phone file system takes longer than loading small ones. Despite this, AF found crashes consistently: in 74% of the runs.

In experiment (ii), we chose two configurations which had the lowest averages of execution time reported in (i) and ran each of them 10 times. We expected to evaluate the impact of the selection of which *atoms* to include in an execution list. The results report a precision (the fraction of runs that ends in a crash out of the total number of runs) of 40%. This might be an indication that the selection of *atoms* is a critical factor in order for the technique to find a crash. In experiment (iii), we run again 10 times the same configurations we used in (ii) but with a more systematic approach to build execution lists. The mean of the execution time of one configuration was higher than the mean time reported by the same configuration in experiment (ii). And the other configuration was faster. Despite this fact, in experiment (iii) AF achieved a precision of a 100%, that is, all runs found a crash. We believe that more systematic algorithms to select *atoms* to include in execution lists can lead the executions to find crashes more efficiently.

Our current experimental results show that AF and SH are complementary. The experimental results indicate that the use of *atoms* offered a better precision, i.e. it could find crash more often than the use of test scenarios. However, as we considered two metrics of quality – crash detection and time, we cannot say from the experimental results that one technique subsumes (that is, it is superior with regard to both metrics) the other in all situations. AF found more crashes. On the other hand, when both found a crash, SH was faster. Also, the variance of the type of crash reported by both techniques was high, i.e. it seems that they can crash the applications in different ways when running in parallel. In addition, it is worth noting that having different crash reports is very important as sometimes these crashes correspond to different faults. This suggests that a testing team should run both techniques when possible.

5.1 Contributions

The main contribution of this work is the idea of taking advantage on the knowledge of a specialist, who spent time and manual effort to write automated test scenarios, to generate different tests (variants/combinations of the original tests). When we shuffle these tests and run them in sequence, we can automatically originate other test scenarios that would exercise the applications under test differently in every execution.

Besides, the *atoms* AF generates allow the use of different inputs every time they are executed. AF also keeps a data map during runtime, which provides input data and stores all data each *atom* generates. From this, we can check how the application deals with several input data, such as big ones, small ones, valid, invalid, etc. AF either allows every execution to start from a different initial configuration of the data map. For instance, in the beginning, the data map may be full of input data, or entirely empty.

The fact that we developed this work inside a company (Motorola) brought a practical appeal to its accomplishment. We could conduct experiments under the actual environment Motorola executes tests, using their cellular phones, and could also compare the technique we are proposing to another one that Motorola has been using in real projects for a while.

5.2 Related Work

The generation of test cases has been extensively studied throughout the years. Our approach aligns more with the works related to test generation based on models extracted from software artifacts, as in model-based testing (MBT) [20]. Such models encode the intended behavior of the system under test (SUT). In MBT, it is firstly necessary to build a formal (or semi-formal) model which describes the system's behavior we are interested in testing. Then, an algorithm selects traces of the model. And finally, these traces constitute test cases for the SUT. The generation often enough amounts to a search problem [42], i.e. it is often necessary to define a graph coverage (search) criterion, which selects test cases from the set of all possible traces of the model.

The behavioral model and the test generation algorithm are what mainly differs one MBT approach from another. For instance, Vieira et al [47] use UML Use Cases and Activity diagrams to respectively describe which functionalities should be tested and how to test them. The generation's algorithm they propose combine Activity diagrams and data annotation to reach a specific graph and data coverage, which directly affect the number of test cases produced. Nogueira [37] proposes a test generation based on CSP models extracted from requirements documents. The generation is guided by test purposes that indicate which functionalities should be considered using a requirements coverage's criterion.

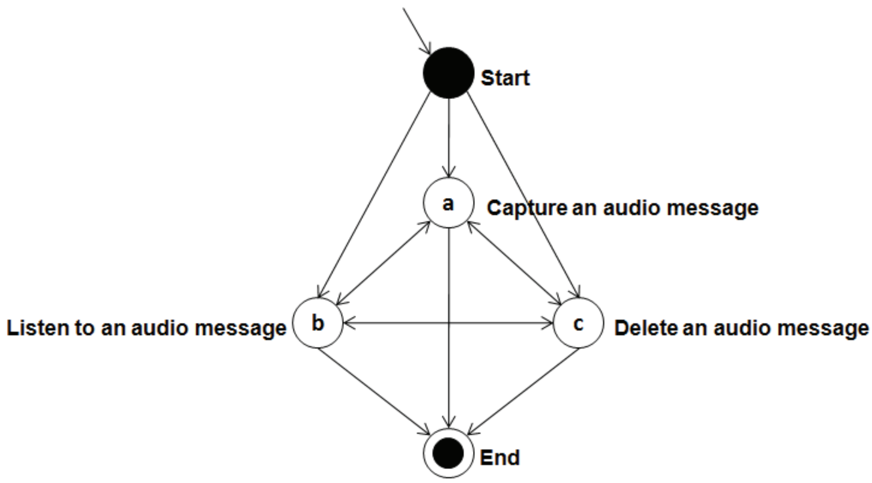

Figure 5.1 Graphic model of a test scenario for audio message.

If what we are doing was exactly MBT, we would first need to (i) identify the *atoms* of each test scenario AF receives as inputs, (ii) construct a model (in any notation) and then (iii) select traces of the model to generate what we call execution lists (sequences of *atoms* to be executed). Figure 5.1 shows an example of how a graphic model of the test scenario we illustrated in Figure 1.2 (page 5) would look like. Although the test scenario imposes a specific execution order for its steps: a –> b –> c (see Figure 1.2), in our approach we are interested in every combination of these steps. Thus, note that the model has a *clique* [15] to establish that each node correspondent to an *atom* (a, b, c) is connected to every other *atom* in the graph; and also that each node is directly connected to the start and end points. It is indeed possible to model test scenarios with a state machine and then explore its paths as in MBT. However, AF does not need to construct a model, as it is already implicit in our strategy.

We may cite other works which are complementary to AF. For instance, a popular approach to automate test documentation and execution is **capture-and-replay**. The idea is to record in a test script the actions that the user makes while interacting with the graphical user interface (GUI) and then to (re-)execute this script when necessary. Stefen at al [45] propose the jRapture, a capture-and-replay tool that operates on Java bytecodes. The authors argument that jRapture can capture much more graphical elements (graphical widgets) than commercial tools. Orso and Kennedy [39] present a technique for selective capture-and-replay of program executions and a tool, called SCARPE, that implements the technique. SCARPE captures (selects) only the minimal subset of the application's state and environment required to replay the execution, while jRapture captures complete input/output information for each execution.

Other works rely on the user's experience to collect information from users, either at runtime or during software development, to be used in software testing. For instance, Brooks and Memon [17] propose a technique to generate test cases based on usage information, in the form of **usage profiles**. These profiles describe event sequences captured from the user's experience, i.e., event sequences captured while the user interacts with the GUI. The technique employs the usage profiles to create and update an abstract model of the GUI and use the model to generate test cases automatically. Although the idea of using profiles is appealing, it is not yet practical in the domain we proposed our technique (cellular communication). In spite of this limitation, it is important to say that this work is complementary to ours. In particular, SH, and consequently AF, use test cases drawn from requirements and history of faults, for example. These do not necessarily correspond to the actual scenarios of use.

Bertolini et al. [13] describe two techniques, Driven Hopper (DH) and Behavior eXplorer Tool (BxT) whose goal is to explore the SUT (in this case, a cellular phone) from its GUI, by performing random key pressings (some of which can change the current screen) for some (pre-configured) time until they find a crash. DH requires tests to drive the phone to an initial screen. Then it starts pressing random keys until the configured timeout is reached and a subsequent test drives the phone to another screen. BxT attempts to make a more systematic selection of keys to be pressed than DH: it recognizes which controls are available at a screen and selects keys according to these controls. For instance, in a screen that contains only two buttons, say "OK" and "Cancel", DH may press several keys before it hits the ones for "OK" and "Cancel". BxT, differently, makes a random selection between one of these two options.

Tools that randomly assign test cases (actions) to the SUT without a user's bias are often called **test monkeys** [30]. Test monkeys and AF are complementary approaches to testing. On the one hand, test monkeys are completely automated, but they perform tests with no knowledge of how humans use the SUT. On the other hand, AF is not entirely automated (yet), but it takes advantage on the knowledge of specialists to generate tests.

5.3 Future Work

In the following, we outline some improvements to complement the work presented here.

Automate the creation of *atoms*

In our approach, we defined a set of tips to help the recognition of *atoms* and another one for including data map support to the *atoms* created. However, this process is not yet automated. As a future work, we intend to develop a tool to automatically extract *atoms* from a set of test scenarios, following the tips we outlined in this dissertation.

Automatically generate *atoms* from new UFs TAF

The idea is to automatically create *atoms* every time a new UF is released. This would remove the need for test scenarios to create *atoms*. We believe that the main challenge here will be the recognition of the UF's dependencies so that we know exactly which other UFs to call before and after the new UF in order to the *atom* execute correctly. We think that it would be very interesting to have *atoms* constantly being generated from new UFs TAF, but we do not know if it is actually viable. We first need to verify how new UFs are released and whether there are enough information to guarantee a sound generation of *atoms*.

Automate the extraction of pre and postconditions

In order to create execution lists based on the Pre and Postconditions Check Algorithm we described in Section 3.3.2, we need to annotate *atoms* with pre and postconditions. We are currently doing this annotation manually. As a future effort, we intend to recognize the pre and postconditions automatically.

The pre and postconditions are directly related to the access to the data map. If there is no access to the data map, both conditions are true. Otherwise, if an *atom* retrieves (resp., stores) data from (resp., to) the data map, the *atom* has preconditions (resp., postconditions). As the data map manipulation is explicitly shown in the procedures of an *atom*, the extraction of its pre and postconditions can be done automatically.

Improve the generation of execution lists

In this dissertation, we proposed two approaches to the generation of execution lists. The first is based on a random selection algorithm; and the second, on a more systematic approach, which involves pre and postconditions checks. But in reality, both of them randomly select *atoms* from a given set to build execution lists. So, as a future work, we intend to implement a more complex algorithm based on exhaustive search and pruning techniques to reduce the search space. As a result, we expect to build more interesting execution lists with a better distribution of *atoms* in order to reduce the number of repetitive paths in a same execution list.

Generalize our approach to other applications

Although in this dissertation we focus our work in cellular phone applications, it is possible to generalize AF to other applications. What we need is a set of executable test scenarios some other technique generates, instead of SH, and to recognize the *atoms* we can extract from them. We also need a set of XML files, structured as the TDF files we used here and filled with data to give as input to the *atoms*. As a result, we could conduct some experiments to observe if AF is also effective when used in applications of other domains.

Conduct more experiments to compare AF and SH

Due to lack of time and resources, we could only conduct one experiment to compare AF and SH. In this experiment we run both techniques only 8 times, which was the number of different phone configurations available for us. As another future work, we plan to conduct more experiments to compare AF and SH to better check which one finds crashes more efficiently. A valid effort would be to get more different phone configurations and execute both techniques again. Or even execute the same 8 phone configurations we used in this work and perform a greater number of runs with execution lists different from the ones we used here.

Create filters to select input data

Currently, if an *atom* requires data from a specific category during its execution, we randomly select an object from the data map among all available objects under such category. We do not distinguish one object from another of the same category. We say they are equally representative for the execution.

As a future work, we intend to create filters to select specific classes of data to use during execution. Initially, we thought about four different classes of data: VALID (for data considered valid, i.e. with the size or type expected for their category), INVALID (the opposite of VALID), BIG (for data with a big size, i.e. objects which size almost reaches or indeed reaches the maximum size expected for their category – it can be, for instance, a long String or a large picture file), and SMALL (the opposite of BIG). More classes could be defined if needed.

Figure 5.2 shows a suggestion on how to place the classes in the *name* attribute of a TDF data item. It follows this pattern: *CATEGORY_CLASS_NAME*, where *CLASS* could be present or not. The first item, *CONTACT_NUMBER_BIG_01*, represents a data item under category *CONTACT_NUMBER*, which class is *BIG* and name is *01*.

```
<CONSTANT name="CONTACT_NUMBER_BIG_01" description="A big contact number.">
    <APPLICATION_CONTENT>
        <VALID>
            <FOR id="*"/>
        </VALID>
        <CONTENT source="plain" value="12345678901234567890123456789012345678901234567890"/>
    </APPLICATION_CONTENT>
</CONSTANT>
<CONSTANT name="CONTACT_NUMBER_INVALID_02" description="An invalid contact number.">
    <APPLICATION_CONTENT>
        <VALID>
            <FOR id="*"/>
        </VALID>
        <CONTENT source="plain" value="123#$%Rs"/>
    </APPLICATION_CONTENT>
</CONSTANT>
<CONSTANT name="CONTACT_NUMBER_03" description="A contact number without a class">
    <APPLICATION_CONTENT>
        <VALID>
            <FOR id="*"/>
        </VALID>
        <CONTENT source="plain" value="87654321"/>
    </APPLICATION_CONTENT>
</CONSTANT>
```

Figure 5.2 Example of data items with different classes.

As a possible solution, we can add a parameter to the execution (a different one for each class of data we defined), or may be more than one, in order to say which classes of data we want the *atoms* to use, and inside the method that retrieves data from the data map we add a parser to the parameter. Depending on the given parameter, the parser will drive the selection of data under the same category to select only data of a specific class. For instance, we would inform the parameter INVALID if we want to select only invalid data items. In accordance to our example, only the second data item (*CONTACT_NUMBER_INVALID_02*) would be selected. If we inform no parameters, it means that we are interested in all data of the data map (no matter their classes) and *atoms* will select data the same way they do today.

With these filters, we could use the same execution list in different executions and check how the SUT deals with different set of data under different classes. For instance, if we run an execution list with only valid data but it does not crash the SUT, as an alternative, we could run the same execution list with only big and/or invalid data items and try again to find a crash.

Snapshot of the state of the data map after execution

As another future work we expect to develop a solution to take a snapshot of the data map after each execution. By snapshot we mean a printed visualization of all data present in the data map at a specific moment, grouped by their categories. This snapshot can be derived from a data log (other future work we describe later).

We believe that a snapshot of the state of the data map after execution would we useful in two ways: (i) to compare the data items stored in the data map to the ones present in the SUT after the execution, and (ii) to compare the state of the data map after two different executions of a same execution list. (i) and (ii) allow us to check if the SUT is working as expected. That is, if it is proper manipulating data as expected by the procedures the *atoms* executed.

Develop a data log system

The idea is to develop a program to generate log files with information regarding to all data manipulation during the execution of *atoms*. The resulting log file would show a representation of all data each *atom* consumed, generated, stored, deleted, and updated. This representation could be any attribute of the object manipulated, since it well distinguishes the object from other different object of the same type. Other important information to be present in the log file would be the time and the name of the corresponding *atom* in each entry of the log.

```
1 Jan 10 10:15:04 2009 ATOM_1 CONSUMED NAME_CONTACT_MARY
2 Jan 10 10:15:04 2009 ATOM_1 CONSUMED NUMBER_CONTACT_7
3 Jan 10 10:15:06 2009 ATOM_1 STORED CONTACT_MARY
4 Jan 10 10:15:10 2009 ATOM_5 CONSUMED CONTACT_MARY
5 Jan 10 10:15:11 2009 ATOM_5 CONSUMED NUMBER_CONTACT_3
6 Jan 10 10:15:13 2009 ATOM_5 UPDATED CONTACT_MARY
7 Jan 10 10:15:15 2009 ATOM_3 CONSUMED CONTACT_MARY
8 Jan 10 10:15:17 2009 ATOM_3 DELETED CONTACT_MARY
```

Figure 5.3 Example of a log file.

Figure 5.3 shows an example of a possible structure of a log file. It first shows the time, then the *atom*'s name, the action it did and the data it manipulated. In this example, from the log entries we can say that the execution first stored a contact with name and number (lines 1 to 3), then updated the contact created previously with another phone number (lines 4 to 6), and deleted the contact (lines 7 and 8).

Load the data map with data from the SUT

And another future work would be to load AF's data map, right before execution, with data already present in the SUT. This would be useful to consume data other executions produced.

Consequently, it would help to decrease the time spent in configuration tests, which load the SUT with data some tests will require during execution, and it would also allow to start the execution from the point another execution left.

Bibliography

[1] Abbot web page. http://abbot.sourceforge.net/, January 2009.

[2] CUnit web page. http://cunit.sourceforge.net/doc/index.html, January 2009.

[3] DUnit web page. http://dunit.sourceforge.net, January 2009.

[4] GUITAR web page. http://www.cs.umd.edu/~atif/GUITARWeb/, January 2009.

[5] James Bach - Satisfice, Inc web page. http://www.satisfice.com/tools/pairs.zip, January 2009.

[6] Java HashMap web page. http://java.sun.com/javase/6/docs/api/java/util/HashMap.html, January 2009.

[7] JUnit web page. http://www.junit.org, January 2009.

[8] Paul Ammann and Jeff Offutt. *Introduction to Software Testing*. Cambridge University Press, 2008.

[9] Anneliese A. Andrews, Jeff Offutt, and Roger T. Alexander. Testing web applications by modeling with FSMs. *Software and Systems Modeling*, 4:326–345, 2005.

[10] Luciano Baresi and Michal Young. Test oracles. Technical Report CIS-TR-01-02, University of Oregon, Dept. of Computer and Information Science, Eugene, Oregon, USA, August 2001. http://www.cs.uoregon.edu/ michal/pubs/oracles.html.

[11] F. Basanieri and A. Bertolino. A practical approach to UML-based derivation of integration tests. In *QWE'00: Proceedings of the 4th International Software Quality Week Europe*, Brussels, Belgium, 2000.

[12] Boris Beizer. *Software testing techniques - Second Edition*. Van Nostrand Reinhold Co., New York, NY, USA, 1990.

[13] Cristiano Bertolini, Glaucia Peres, Marcelo d'Amorim, and Alexandre Mota. An empirical evaluation of automated black-box testing techniques for crashing GUIs. In *ISCT'09: Proceedings of the 2nd International Conference on Software Testing, Verification and Validation*, pages 21–30. IEEE, 2009.

[14] A. Bertolino, E. Marchetti, and A. Polini. Integration of "components" to test software components. *Electronic Notes in Theoretical Computer Science*, 82:49–59, 2003.

[15] Norman Biggs. *Algebraic Graph Theory*. Cambridge University Press, 1993.

[16] B. W. Boehm, J. R. Brown, and M. Lipow. Quantitative evaluation of software quality. In *ICSE'76: Proceedings of the 2nd International Conference on Software Engineering*, pages 592–605, Los Alamitos, CA, USA, 1976. IEEE.

[17] Penelope A. Brooks and Atif M. Memon. Automated GUI testing guided by usage profiles. In *ASE'07: Proceedings of the 22th IEEE/ACM International Conference on Automated Software Engineering*, pages 333–342, New York, NY, USA, 2007. ACM.

[18] Emanuela Cartaxo. Test case generation by means of UML sequence diagrams and label transition system for mobile phone applications (in Portuguese). Master's thesis, Federal University of Campina Grande, UFCG, Brazil, 2006.

[19] Yanping Chen, Robert L. Probert, and D. Paul Sims. Specification-based regression test selection with risk analysis. In *CASCON'02: Proceedings of the 2002 Conference of the Centre for Advanced Studies on Collaborative Research*, page 1. IBM Press, 2002.

[20] S. R. Dalal, A. Jain, N. Karunanithi, J. M. Leaton, C. M. Lott, G. C. Patton, and B. M. Horowitz. Model-based testing in practice. In *ICSE'99: Proceedings of the 21st International Conference on Software Engineering*, pages 285–294, New York, NY, USA, 1999. ACM.

[21] Elfriede Dustin, Jeff Rashka, and John Paul. *Automated Software Testing: Introduction, Management, and Performance*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.

[22] I. Esipchuk and D. Vavilov. PTF-based test automation for Java applications on mobile phones. In *ISCE'06: Proceedings of the 10th International Symposium on Consumer Electronics*, pages 1–3. IEEE, 2006.

[23] G. Friedman, A. Hartman, K. Nagin, and T. Shiran. Projected state machine coverage for software testing. In *ISSTA'02: Proceedings of the 2002 ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 134–143, New York, NY, USA, 2002. ACM.

[24] Patrice Godefroid, Nils Klarlund, and Koushik Sen. DART: directed automated random testing. In *PLDI'05: Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 213–223, New York, NY, USA, 2005. ACM.

[25] Dick Hamlet. When only random testing will do. In *RT'06: Proceedings of the 1st International Workshop on Random Testing*, pages 1–9, New York, NY, USA, 2006. ACM.

[26] David Harel. Statecharts: A visual formalism for complex systems. *Sci. Comput. Program.*, 8(3):231–274, 1987.

[27] Hyoung Seok Hong, Young Gon Kim, Sung Deok Cha, Doo-Hwan Bae, and Hasan Ural. A test sequence selection method for statecharts. *Software Testing, Verification and Reliability*, 10(4):203–227, 2000.

[28] Luiz Kawakami, André Knabben, Douglas Rechia, Denise Bastos, Otavio Pereira, Ricardo Pereira e Silva, and Luiz C. V. dos Santos. An object-oriented framework for improving software reuse on automated testing of mobile phones. *Testing of Software and Communicating Systems, Lecture Notes in Computer Science*, 4581:199–211, 2007.

[29] Gary T. Leavens, Albert L. Baker, and Clyde Ruby. Preliminary design of JML: a behavioral interface specification language for Java. *SIGSOFT Softw. Eng. Notes*, 31(3):1–38, 2006.

[30] Kanglin Li and Mengqi Wu. *Effective GUI Test Automation: Developing an Automated GUI Testing Tool*. SYBEX Inc., 2005.

[31] Barton P. Miller, Louis Fredriksen, and Bryan So. An empirical study of the reliability of UNIX utilities. *Commun. ACM*, 33(12):32–44, 1990.

[32] Chen Mingsong, Qiu Xiaokang, and Li Xuandong. Automatic test case generation for UML activity diagrams. In *AST'06: Proceedings of the 1st International Workshop on Automation of Software Test*, pages 2–8, New York, NY, USA, 2006. ACM.

[33] Daniel J. Mosley and Bruce A. Posey. *Just Enough Software Test Automation*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2002.

[34] Glenford J. Myers. *Art of Software Testing - Second Edition*. John Wiley & Sons, Inc., New York, NY, USA, 2004.

[35] Euclides Napoleão Arcoverde Neto. Abordagem para geração automática de código para framework de automação de testes (in Portuguese). Master's thesis, Informatics Center of Federal University of Pernambuco, CIn/UFPE, Brazil, 2006.

[36] National Institute of Standards and Technology. The economic impacts of inadequate infrastructure for software testing. Planning Report 02-3, May 2002. http://www.nist.gov/director/prog-ofc/report02-3.pdf.

[37] Sidney Nogueira. Guided csp test case generation guided by purposes (in Portuguese). Master's thesis, Informatics Center of Federal University of Pernambuco, CIn/UFPE, Brazil, 2006.

[38] Catherine Oriat. Jartege: A tool for random generation of unit tests for Java classes. In *SOQUA'05: Proceedings of the 2nd International Workshop on Software Quality*, pages 242–256, Erfurt, Germany, 2005.

[39] Alessandro Orso and Bryan Kennedy. Selective capture and replay of program executions. In *WODA'05: Proceedings of the 3rd International Workshop on Dynamic Analysis*, pages 1–7, New York, NY, USA, 2005. ACM.

[40] Carlos Pacheco, Shuvendu K. Lahiri, Michael D. Ernst, and Thomas Ball. Feedback-directed random test generation. In *ICSE'07: Proceedings of the 29th International Conference on Software Engineering*, pages 75–84, Washington, DC, USA, 2007. IEEE Computer Society.

[41] William E. Perry. *Effective methods for software testing*. Wiley Publishing Inc., third edition, 2006.

[42] Alexander Pretschner. Model-based testing. In *ICSE'05: Proceedings of the 27th International Conference on Software Engineering*, pages 722–723, New York, NY, USA, 2005. ACM.

[43] Padmanabhan Santhanam and Brent Hailpern. Software debugging, testing, and verification. *IBM Systems Journal*, 41:4–12, 2002.

[44] Ian Sommerville. *Software Engineering*. Addison-Wesley, 2007.

[45] John Steven, Pravir Chandra, Bob Fleck, and Andy Podgurski. jRapture: A capture/replay tool for observation-based testing. *SIGSOFT Softw. Eng. Notes*, 25(5):158–167, 2000.

[46] Keith Stobie. Too darned big to test. *Queue*, 3(1):30–37, 2005.

[47] Marlon Vieira, Johanne Leduc, Bill Hasling, Rajesh Subramanyan, and Juergen Kazmeier. Automation of GUI testing using a model-driven approach. In *AST'06: Proceedings of the 1st International Workshop on Automation of Software Test*, pages 9–14, New York, NY, USA, 2006. ACM.

[48] Joachim Wegener and Ines Fey. Systematic unit-testing of ADA programs. In *Ada-Europe'97: Proceedings of the 1997 Ada-Europe International Conference on Reliable Software Technologies*, pages 64–75, London, UK, 1997. Springer-Verlag.

[49] Qing Xie and Atif M. Memon. Designing and comparing automated test oracles for GUI-based software applications. *TOSEM: ACM Transactions on Software Engineering Methodology*, 16(1):4, 2007.