FACULDADE DE ENGENHARIA DA UNIVERSIDADE DO PORTO

Development of GUI Test Coverage Analysis and Enforcement Tools

Ricardo Daniel Ferreira Ferreira

FINAL VERSION

Report of Dissertation
Master in Informatics and Computer Engineering

Supervisor: João Carlos Pascoal de Faria (PhD)

July 2009

Development of GUI Test Coverage Analysis and Enforcement Tools

Ricardo Daniel Ferreira Ferreira

Report of Dissertation
Master in Informatics and Computing Engineering

Approved in oral examination by the committee:

Chair: António Augusto de Sousa (Associated Professor)

External Examiner: José Francisco Creissac Freitas Campos (Auxiliary Professor)

Internal Examiner: João Carlos Pascoal de Faria (Auxiliary Professor)

24th July, 2009

Abstract

Software applications are becoming more and more important in today's society. From them depends the functioning of critical systems in several areas. To interact with these systems were developed graphical user interfaces (GUIs). These interfaces are more attractive to the user allowing to use the functionalities of the system with ease.

Tests are needed to assure the reliability of the systems and increase the confidence on its correct functioning. The application can have several elements to be tested. One of them is the GUI. Inside this, faults have to be discovered whether they are in the code of the interface or simply usability faults.

However GUI testing is hard. The GUIs can contain very complex controls, endless sequences of actions and many states. They also have code behind that needs to be tested too. Even when this code is automatically generated, it can contain errors.

There are several approaches for GUI testing. Model-based testing is the one that offers more advantages in automatic test case generation and execution. This technique requires the definition of an abstract model of the system under test (SUT), from which test cases can be generated. We can evaluate the quality of the test cases and then apply them to the SUT.

To evaluate the quality of the tests, coverage criteria are defined. They allow to get a value that is the coverage degree of the tests over the application or the model. This degree of coverage can intervene in test generation process. When this coverage degree has a required value, the test generation can stop.

There are some tools that ease the model-based testing process. Each tool use a different modelling notation to represent the system. From the tools found and presented in this report, none of them have model coverage analysis feature.

The work developed fits into the area of GUI testing. It is presented a survey of coverage criteria applied to models and a model coverage analysis tool that addresses and presents visually the model coverage achieved by a given test suite.

The model supported by the tool use UML notation, namely UML state machines. This notation was chosen because it is a standard and because it is easily interpreted by other people. To assess differences between some notations, it was created a simple system and tested it using the different notations.

The tool developed receives an UML model and several test cases. The tool then executes the model according to the test cases and indicates which elements were exercised by the test cases. In case the model was not fully covered, the tester can add tests to assure maximum coverage of the model.

This tool represents an improvement over existing MBT tools because they usually lack a coverage analysis feature.

Resumo

As aplicações de software são cada vez mais importantes no dia-a-dia. Delas dependem o bom funcionamento de sistemas cada vez mais cruciais em diversas áreas. Para que possa ser possível interagir com estes sistemas foram desenvolvidas interfaces gráficas. Estas interfaces gráficas são mais apelativas para o utilizador permitindo que utilize as funcionalidades do sistema com maior facilidade.

Para que as aplicações sejam fiáveis, aumentando a confiança do utilizador na aplicação, é necessário realizar testes sobre a mesma. A aplicação poderá conter vários elementos a serem testados. Entre os elementos a serem testados encontra-se a interface gráfica. Nesta, terão de ser procuradas e solucionadas falhas que a interface possa conter quer a nível conceptual quer a nível de usabilidade.

Porém, o teste de interfaces gráficas ainda não é facilmente exequível. Estas podem conter controlos bastante complexos, inúmeras sequências de acções e imensos estados possíveis. Ainda, as interfaces gráficas contêm código integrado que também deverá ser testado. Mesmo quando este código é automaticamente gerado pode conter erros.

Para avaliar a qualidade dos testes, são definidos critérios de cobertura permitindo obter um valor que corresponderá ao grau de cobertura dos testes sobre a aplicação. O grau de cobertura poderá intervir na geração de casos de teste no sentido em que poderá parar a geração de casos de teste quando o grau de cobertura for o pretendido.

Entre as técnicas de teste de interfaces disponíveis actualmente, o teste baseado em modelos é o que melhor se enquadra quando o tema é geração e execução automática de casos de teste. Esta técnica permite uma definição do sistema em forma de um modelo abstracto. A partir dos testes sobre o modelo poderemos aferir a qualidade dos casos de teste e aplicá-los então ao sistema a testar.

Existem então algumas ferramentas que auxiliam o processo de utilização de teste baseado em modelos. As ferramentas utilizam diferentes tipos de modelação do sistema. De entre as ferramentas encontradas e referenciadas nesta dissertação, nenhuma delas apresenta análise da cobertura do modelo por parte dos casos de teste gerados.

O trabalho foi desenvolvido no âmbito do teste de interfaces. Para isto, propõe-se a apresentar critérios de cobertura aplicáveis a modelos e também a criar uma ferramenta que permita fazer uma análise da cobertura dos modelos de entrada.

Quanto aos modelos, foram escolhidos modelos em notação UML nomeadamente os modelos de máquinas de estado. Esta notação foi escolhida por ser um standard da indústria e pela sua fácil interpretação por terceiros. Para aferir diferenças entre algumas notações de modelação foi criado um sistema que foi testado utilizando diferentes tipos de modelos.

A ferramenta desenvolvida deverá receber o modelo UML e vários casos de teste e indicar, pintando, no modelo original quais foram os elementos visitados pelos casos de teste. No caso de não ser completamente coberto, poderão ser adicionados mais testes e verificada a sua cobertura sobre o modelo.

Esta ferramenta traz vantagens sobre as actualmente existentes visto que pretende colmatar uma falha existente nas ferramentas actuais, permitindo obter uma análise da cobertura do modelo através da sinalização dos seus elementos.

Acknowledgements

I would like to thank my supervisor, Professor João Carlos Pascoal de Faria, from Engineering Faculty of Porto University, for his guidance, determined search of resources, unforgettable mentoring and encouragement that made this dissertation possible.

A special thank to my co-supervisor Professor Ana Cristina Ramada Paiva Pimenta, also from Engineering Faculty of Porto University, for her inputs, enthusiasm and his invaluable perceptiveness in the discussions we had.

I owe special thanks to André Grilo and Nelson Rodrigues, for the suggestions, feedback, the time we spent working together, and the talk he gave here in Engineering Faculty of Porto University.

I wish to express my gratitude to my parents and sister, Sebastião Ferreira, Laura Maia and Laura Ferreira, for all their support, care, comprehension, and love. Thank you.

Ricardo Daniel Ferreira Ferreira

Contents

List of Figures

List of Tables

Abbreviations

ASM	Abstract State Machine
FIT	Framework for Integrated Testing
FSM	Finite State Machine
GUI	Graphical User Interface
GUITAR	GUI Testing frAmewoRk
LINQ	Language INtegrated Query
MBGT	Model-Based GUI Testing
MBT	Model-Based Testing
MTC	Model Test Coverage
OCL	Object Constraint Language
ParTeG	Partition Test Generator
PIN	Personal Identification Number
RPN	Reverse Polish Notation
SQL	Structured Query Language
SUT	System Under Test
UML	Unified Modeling Language
UTP	UML Testing Profile
XMI	XML Metadata Interchange
XML	eXtensible Markup Language

1 Introduction

Graphical User Interfaces (GUIs) are the most used interface for software, providing user-friendly access to the functionalities of the system by responding to user events such as mouse movement, menu selections, etc.. With the widespread use of this kind of interface, the construction of increasingly complex GUIs is becoming frequent. As they make software easy to use, they also make software development process more difficult [BrM07, MSP01].

As it is an important part of the system, GUIs must also be tested for a better confidence in the application developed. Due to the very close relation between the final user and the GUI of the application, GUI testing is an important issue because its defects can drastically influence user's impression about the overall quality of the system developed.

Testing GUIs is not an easy task because of its flexibility and variety. They also have an enormous amount of different permutations of events and interactions possible that can be executed by the user. Even when tools automatically generate the GUI, they are not bug free and can cause the system to fail. To effectively test a GUI, it is important to understand how tests can be abstracted to ensure resilience and cost-effective test automation [BDG07]. As the GUIs are mostly event-driven, usual testing techniques cannot be used [MeS03].

To assess the adequacy of some test suite, coverage criteria have to be defined. Coverage criteria are a set of rules that help determining the quality of a test suite. Coverage criteria can also be used to guide the test generation process. But, for GUI testing, not all coverage criteria can be used. For example, code-based coverage criteria cannot address some interactions between the events that the user performs in the application and the system.

For GUI testing, some techniques have to be defined. One of them is model-based testing (MBT). This technique will be the main theme of this dissertation and will be compared to other GUI testing approaches.

There are inclusive some tools developed that uses MBT to test GUIs. A short list of available tools that uses MBT for GUI testing will be presented.

1.1 Context

Software testing is gaining more and more importance. With the crescent complexity, size of the software, and reduced development cycle times, more systematic and automated approaches are being seek. Model-based testing (MBT) techniques are beginning to be more and more used by software developers and test designers. MBT presents several advantages with respect to other approaches like manual testing and capture/replay testing [UtL06]; the main advantage is that test cases are generated and executed automatically from a model that

specifies the intended behavior of the system under test (SUT). However, to ensure the generation of adequate test suites, it is important that the test generation process is guided by appropriate coverage or quality criteria.

Tools using model-based testing are being created to automate as much as possible GUI testing, diminishing the testing effort and associated costs. Current tools do not present solutions to all the problems that GUI testing has (such as model coverage analysis).

This dissertation was done as a part of the research project AMBER iTest – An Automated Model-Based User Interface Testing Environment. The project is being developed by FEUP with collaboration of Critical Software (CSW). The final goal is the development of a pack of tools and techniques to automate specification-based GUI testing. The project also intends to be innovative, complementing the tools actually existing.

1.2 Motivations and goals

GUI testing is becoming more important since the arrival of GUIs. The GUIs allow the user to interact with the application in a higher level of abstraction. Actual tools using MBT cannot address all the problems presented by GUI testing. The main goal of this research work is to improve GUI testing tools, so the GUIs can be better tested, presenting a better quality when delivered to final user. More specifically, the work reported in this dissertation addresses two problems already referred: the identification of appropriate coverage criteria for the models and the development of a tool that feedbacks the user about model coverage achieved by a test suite.

The main goals of this dissertation are:

- Analyze and identify coverage criteria for model-based GUI testing. Since model coverage criteria is strongly related to the kind of model used, the work also involves the identification and analysis of models for model-based GUI testing;
- Develop a tool that analyze and represent visually the coverage of model's elements achieved by a given test suite.

1.3 Dissertation's structure

Beside this introduction, this thesis is structured as follows:

- In section 2 , the state of the art in approaches to develop tests for GUIs is presented. The comparison between approaches presented help justifying why MBT is gaining more and more strength actually;
- In section 3 , the most important kinds of modeling notations, coverage criteria and tools are presented;
- Section 4 presents an example of an alarm system to practically evaluate characteristics of the modeling notations and coverage criteria;
- Section 5 describes the tool developed, showing how it works, its architecture and algorithms and cares to take when creating both the model and the test suite to use as input of the application;
- The last section presents the conclusions and future work;
- Appendix A refers GUI interface errors usually encountered in practice;
- Appendix B shows the test suite manually derived from the UML model of the system defined in section 4 ;
- Appendix C shows the code using Spec# to implement the system defined in section 4 ;
- Appendix D demonstrates the algorithm used in the tool developed to check the coverage of the model against a test suite;
- Appendix E shows a portion of the XML file that describes the model to be imported by Enterprise Architect and the application developed.

2 GUI testing processes

This chapter describes testing techniques that can be used to perform GUI testing. The concepts and implementation methods of each technology are presented as well as the main advantages and disadvantages of each one.

2.1 Introduction

Testing software can be seen as a search problem. We will search among the infinite possibilities of input combinations available to be performed, those that cause the system to fail and those that do not.

The main goal of software testing is to produce a set of tests that, as exhaustively as practically possible, test the System Under Test (SUT) with normal or exceptional behaviors which are specified by complete and correct requirements definition. Completeness requires that enough aspects of our system are specified and correctness means that there are no ambiguities, errors or inconsistencies within requirements [BDG07].

In [Wag04], the author refers that there are a lot of problems of GUI testing. The main problems that the author refers are:

- It is hard to test GUIs with hybrid complex architectures;
- GUIs are changing for usability issues and the tests needs to be modified every time the GUI presents some change;
- Unsolicited events that can occur can interrupt a sequence of commands without notice and at any time;
- Window management operations (resize, move) can also cause bugs;
- The GUI is almost always platform-independent and the tests are also difficult to support multiple platforms;
- Interface design issues is an important factor in GUI's quality.

When testing the functionality of the SUT, we should consider some key steps that are important to be followed to achieve a good testing process. Those are [UtL06]:

- **Design Test cases**: all test cases should be designed in the beginning, taking in account system requirements and the test objectives. Each test must have a context and pass/fail criteria. Along with the test design, the test conditions for each test item, detailed test approach and associated test cases are identified [IEE98].
- **Execute tests and analyze results**: The test cases are tested on the SUT or in some model that includes SUT’s functionalities, depending on the approach. The result of tests are written in a Test Results document that will be later analyzed to see which tests

have passed or failed and, for those who failed, determine the cause of test execution failure and correct them.

- **Verify test coverage**: To achieve a good quality of the tests we must have some metrics on how well tests cover all functionalities included in the SUT. In GUI testing it is normally used requirements coverage by the test suite, done through a traceability matrix between requirements and test cases.

Some GUI testing techniques that can be used to test the GUI of applications will be presented next, showing the differences between each approach, its advantages and disadvantages.

Figure 2.1 shows the notation used in following diagrams to better understand them.

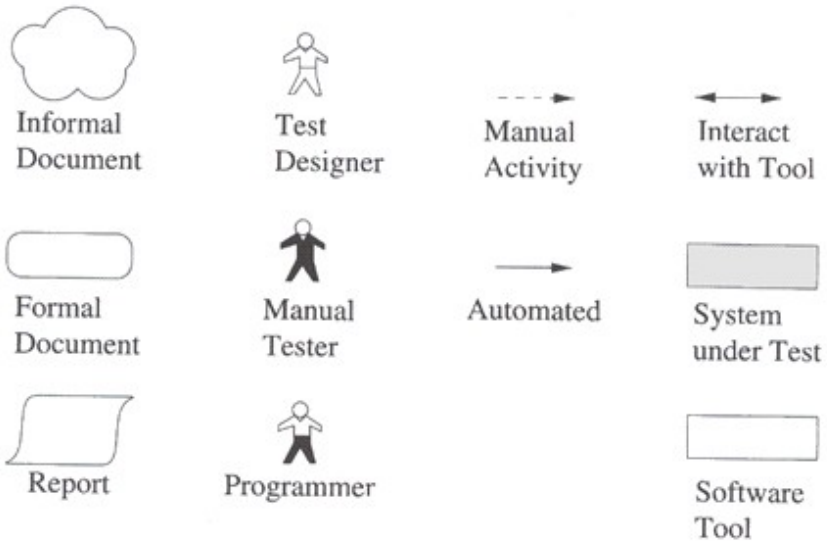

Figure 2.1: Notation used in the following figures [UtL06].

2.2 Manual testing process

This process is the oldest, but still, most of the tests are done manually [UtL06]. Figure 2.2 shows the phases of a manual testing process.

The “Test Plan” is an overview of which parts of SUT are to be tested, how often tests will be executed and the approaches to be used.

The test design phase is done manually, based on informal Requirements and Test Plan. In this phase, the test designer should document, in a human readable manner, the tests that are going to be performed in the SUT by the manual tester. This phase is a very time-consuming task and does not ensure a systematic coverage of SUT’s functionality.

After test design is complete and the SUT is ready for testing, the tester executes manually the tests defined previously, interacting directly with the SUT. Then, the actual output is compared to the expected output to check if the test passes or fails. The results of each test case are written on a Test Results document for further analysis.

The skills required from the test designer and the tester are completely different. While the test designer needs to have knowledge in test design strategies and know the SUT well, the tester only needs to know how to interact with the SUT to follow the steps defined in test case and record the output of the test.

Once a new release of the SUT needs to be tested, the tester will execute manually all tests against the SUT. This becomes a very boring, error-prone and time-consuming task, because testers can sometimes unintentionally leave some tests untested and others tested several times. It’s very time-consuming since the testing time is high and always the same on each new release

of the SUT. To keep costs within the budget stipulated, on each new release, some tests that were executed previously are not executed because they are not related with the modified portion of code. The main intention of this strategy is to reduce the test cases to be applied. With this approach, the SUT becomes incompletely tested and some unexpected errors can occur, affecting product stability and robustness.

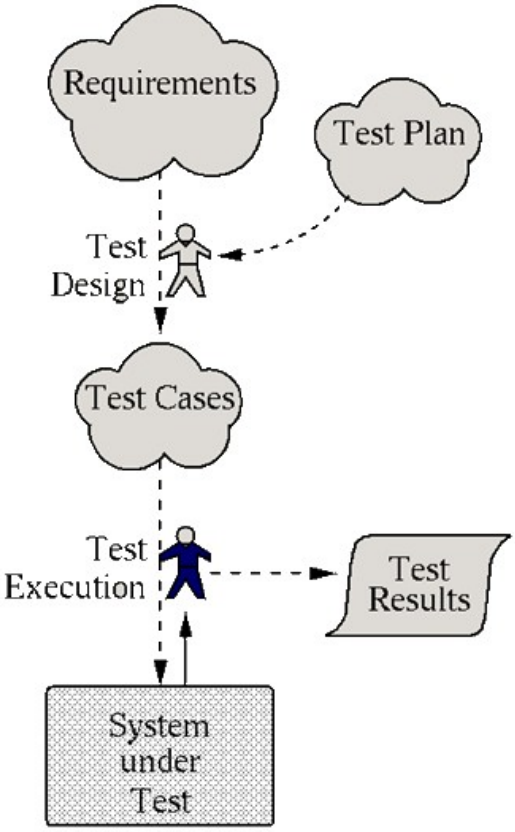

Figure 2.2: Manual testing process [UtL06].

Test coverage metrics are also computed manually. A manual analysis has to be done to ensure that all logical combinations were tested, requiring lots of time, effort and a very good knowledge of the SUT and its functionalities.

Although it is a totally manual process, it is widely used to test essentially the business logic because it is harder to teach it to machinery than for the testers to learn the logic. Intuition plays also a big role in these situations. Manual testers also have the time to see small business logic errors [Wit08].

2.3 Script-based process

All automated processes of testing are based on some kind of script. Scripts can run against the SUT with proper tools that will interpret the lines contained on the script file and translate them into actions to be performed in the SUT. So, in the execution test phase, human interactions with the SUT are unnecessary.

In [BBN04], the authors refer that all test scripts typically have a predefined sequence of steps that:

- Initializes the SUT;
- Loops through a set of test cases, and for each of them:

- Initializes the SUT (optional);
- Sets the input;
- Execute the SUT;
- Captures the output of the SUT and compares it to the expected output storing the results for further analysis.

Scripts have an advantage when test execution is to be done repeatedly because it saves a lot of time requiring only computational effort to run the tests. The processes described in the next sections use scripting to automate test execution. In the simplest way, scripts are generated manually. Techniques to generate scripts automatically will be discussed in later sections.

In Figure 2.3 we can see a diagram explaining how manual scripting usually works. The test design phase is the same as in the manual testing process. The difference comes in a later phase when executing the tests. Instead of having a tester manually testing the SUT, we have a programmer that will implement the test script manually. This process automates the test execution and further executions of the tests.

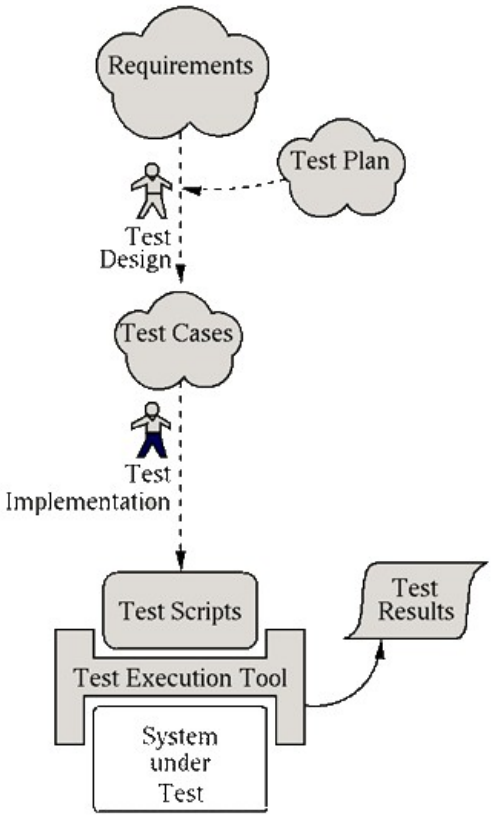

Figure 2.3: Manual scripting process [UtL06].

Most of the time required to perform the tests will be in the implementation phase where the programmer would have to write all test cases into the script manually, requiring some special skills to do it.

This approach has also some maintenance problems because if the application is changed, the script will have to be changed too. As the script can be very extensive and complex, the places where we need to change can be difficult to find and it can be a time-consuming, costly and error-prone task.

2.3.1 Keyword-driven testing

Keyword-driven testing is basically another manner to structure the script. Action keywords are used in the definition of the test cases. Each action keyword relates to a fragment of the test script. The "Adaptor" code in Figure 2.4 converts the action keyword into the respective code so the test execution tool understands it, raising the level of abstraction. Each test case will be nothing more than action keywords and test data.

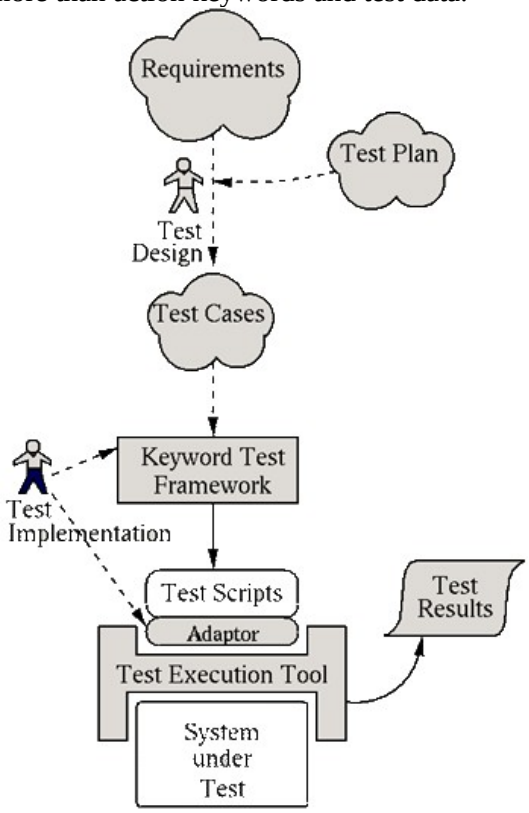

Figure 2.4: Keyword-driven testing process [UtL06].

The implementation of these keywords still requires programming skills, but the design of test cases is now at a higher level allowing other people that are not programmers to write the test cases (although programmers need to implement the adaptor code).

2.3.2 FIT

Framework for Integrated Testing (FIT) is a testing framework that promotes collaboration between the development team and the customer. It allows to know what the software actually does and describes what it should do. The main purpose of FIT is to write acceptance tests that could be validated by the customer [Cun07].

With this framework, the customer can write an HTML table with some tests and the expected result. The table is then analyzed by a "fixture" that programmers implement and then FIT tools like FitNesse [FiN09] will paint the cells of the table with green or red colors depending if the test performed against the SUT passes or fails respectively.

There are different kinds of tables that can be created. ColumnFixture is a kind of table where the first column of each row has the attribute or method and the following rows contain the test cases. Each row is executed column by column until the end of the row.

The other kind of table is the ActionFixture (Table 2.1) where each table represents a sequence of actions that together define a test case. Each ActionFixture can correspond to a user story. ActionFixtures have 4 commands although more can be defined. The predefined commands are: **start** that has the name of the class to start the feature, **enter** that executes a method with parameters, **press** that executes a method with no parameters and **check** that verifies the result of a method with no parameters. In user interface, enter action can be programmed for example to insert text in a text box; press can be used to click a button and check to see the result presented in a label.

This technique also requires programming skills as it requires that the mapping code between the application and the instructions defined in the table are implemented. This technique can be seen as a special case of keyword-driven testing.

Table 2.1: ActionFixture table sample.

Action Fixture		
start	Fixture.CountFixture	
check	counter	0
enter	counter	2
press	incrementCounter	
check	counter	3

2.3.3 Data-driven testing

Data-driven testing is the parameterization of test scripts allowing different data to be introduced to the same test procedure. With one test procedure and a data file, we can have several test cases, all differing in data values given to the attributes/methods. This makes test scripts more generic and can be applied in a lot of test cases, reducing test script maintenance problems. This approach does not require any tool except the one who will merge the data with the tests that are included in the test script. This approach can be used with keyword-driven testing, allowing the definition of more test cases with less effort.

2.4 Capture/replay testing

The capture/replay testing process allows the capture of actions performed to the SUT and saves them in a script file. Later, when testing the SUT, the tool will repeat exactly the same steps on the SUT. The actions are performed manually by a tester and the tool records the actions, inputs and outputs. This script can be edited to refine test cases, but this requires some skills from the person who edits the file.

When re-executing the tests, the test execution tool will read the script, set the inputs, execute the actions on the SUT and compare the output to the previously saved output.

The tests will be executed exactly the same way as initially saved. So the time needed to perform the tests the first time is the same as in manual testing. The advantage of this approach comes in further testing of the SUT because there is no need to execute them manually. The tool executes the tests and then gives the feedback about the tests that passed and failed.

This technique is very fragile because if some change occurs in the layout of the SUT the test will fail. This can happen because the tool expects the control in one place and it does not appear there or another control appears in its place or no control appears at all. Then the tests that are affected by the change need to be saved again manually so it can work in the next

release. So, its inability to adapt to small changes in the SUT creates a maintenance problem, causing this technique to be abandoned in many situations.

In Figure 2.5 we see all the steps of capture/replay testing. In the bottom of the figure we can see that the manual tester interacts with the tool recording the test cases for further use.

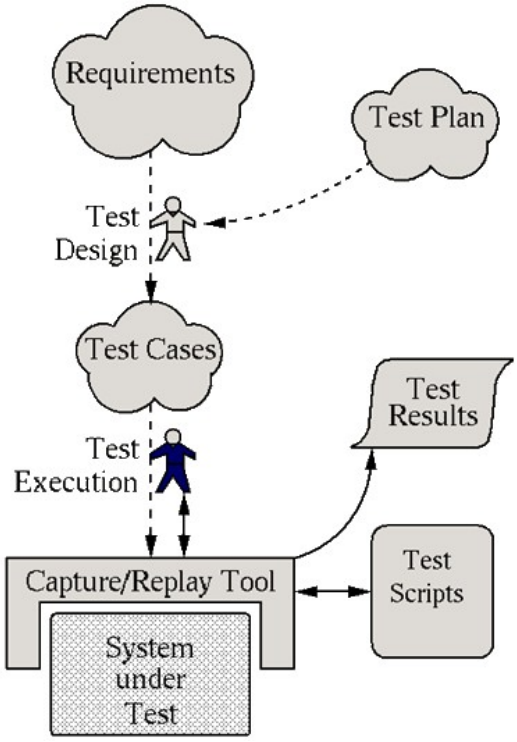

Figure 2.5: Capture/replay process [UtL06].

An example of a tool that does that is Selenium tool [Sel09]. This tool is only for testing web pages, but the concepts to test other kinds of applications are the same. To begin the test process, the tester starts the saving process. This process will save every action performed by the user including clicks and text writing. To verify if the test has completed successfully, the tool supplies ways to check the contents of a page, verifying if some word or expression is there. The tool also manages several test cases and gives feedback about those who have failed and those who have completed successfully. This will allow seeing the number of tests failed and where they have failed also. This tool has support for several browsers and can export the test suite to several programming languages and testing frameworks like JUnit [JUn09].

2.5 Model-based testing process

Model driven development plays actually a major role in reducing time and budget of software development because it introduces techniques such as automated software production [BDG07]. As models are helping in the software production, model-based testing appears as the equivalent for software testing.

Model-based testing is the automatic generation of tests, derived whole or in part from a model representing system's requirements and functionalities [WikiMBT, MBT09].

When the system has a large amount of input and state combinations, previous approaches turn to be impotent at this scale. The approaches used must systematically find relevant combinations according to defined criteria (because most of the time it is impossible to generate all the combinations of states and inputs). The approach needs also to be focused to ensure that

all the information available is used wisely to seek the source of common errors and automatic to ensure that a large amount of tests are generated and re-tested when necessary.

Model-based testing ensures all the three objectives. The number of tests that are performed is still very small in comparison with the space of inputs and states, but it is much larger than the tests that can be designed/run manually.

Binder refers in his book that "Automated testing replaces tester's slingshot with a machine gun. The model paints the target and casts the bullets" [Bin00].

Model-based testing is also dependent on scripting, but it aims to automate all phases of testing processes: test case design is done by a Test Case Generator tool that can have different test selection criteria, test case implementation is done by Test Script Generator executing the tests into the SUT and finally the verification of test output and analysis of expected values.

In MBT, the function of the test designer is to generate a simplified model of the SUT instead of test case design, and then, tools will automatically generate and run test suites depending on test selection criteria, reducing both the time of test design and execution of the tests.

As shown in Figure 2.6, MBT is divided into five different phases:

1. **Model**: This phase consists in writing an abstract model of the SUT that should be much smaller and simpler than the SUT. It must focus on the key aspects that we want to test and omit many of the details of the SUT. The model will be iteratively refined until all the behaviors desired are implemented and correct. The model can have some requirement identifiers to document the relations between the model and requirements. Besides helping in analysis of requirements coverage in a later phase, this will help to clarify the requirements and check if all requirements intended to be tested are actually implemented in the model. After writing the model and before generating test cases, it is convenient to check the consistence of the model and if it has the expected behavior using tools such as type checkers and animators.
2. **Generate**: This step generates abstract test cases from the model, taking in account defined test selection criteria that restricts the usually infinite possibilities of test cases. For example, we could be interested in testing only some part of the model, or choose between different coverage criteria. From this phase we should get a set of abstract test cases consisting on a sequence of operations generated from the model. Since the model is a simplified view of the SUT, these tests are not directly executable in the SUT because it lacks some details needed. The details will be added automatically in the next phase.

   Another output that could be generated in this phase is the requirements traceability matrix and other coverage reports. The requirements traceability matrix links the requirements and test cases. This will allow seeing if all requirements have been covered by test cases. Coverage reports consist in values that explain how well the behaviors of the model have been tested.

   These reports could be also used to identify parts of the model that aren't well covered by test cases. In this situation, we can try changing some test selection criteria and repeat the test case generation again and see if the problem was solved. In an extreme case, we can add some test case by hand to exercise one specific path of the model that has not been exercised by the automatically generated test suite.
3. **Concretize**: The next step in MBT implementation is to concretize the abstract tests into executable tests. This is done by a Test Script Generator tool that uses mappings and templates to translate the abstract tests into executable tests which are then saved in a test script. So, the main goal of this step is to fulfill the gap between abstract tests and the SUT, adding some details needed that are not present in abstract tests.

   The distinction between abstract tests and concrete tests is actually useful because abstract tests can be independent of the language used to write the tests. By changing some "Adaptor" code and/or translation templates, we can use the same tests on

multiple environments. Other advantage is that the skills required to add abstract tests are fewer because these tests can be in a high-level language.

4. **Execute**: This phase will execute the concrete tests into the SUT. There are now two approaches to do this. The first is online model-based testing which consists in executing the tests at the same time they are generated so, the execution tool will manage the execution of tests and save their results. The second approach is offline model-based testing that picks some generated test script and executes the tests into the SUT. With this approach it is easier to re-execute tests regularly, recording the results of each test.
5. **Analyze**: This is the final step in MBT. It consists in analyzing the results of the execution of tests and take needed actions. For those tests that failed in the execution we must determine their origin and perform the appropriate corrections. In the case of MBT there are more places where to look for errors. The cause can be in the SUT, in the test case itself, in the model, and ultimately in the tools that translated the model into tests, although this is more difficult because the tools are generally well tested.

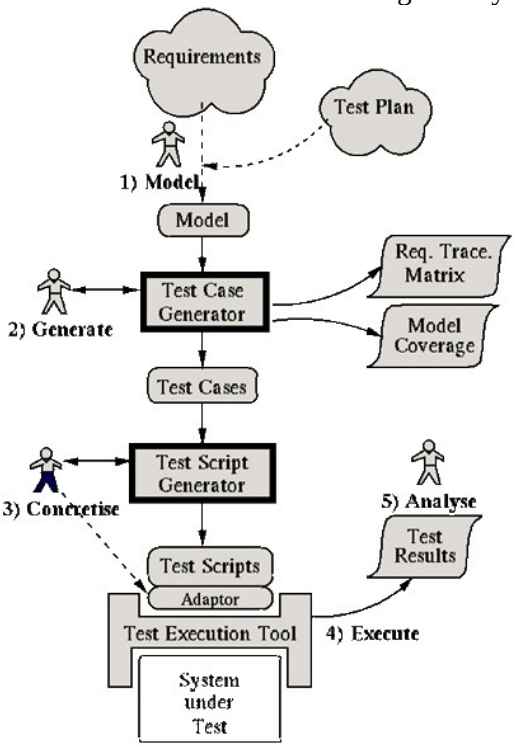

Figure 2.6: Model-based testing process [UtL06].

Utting says that the first execution of tests generally finds more errors. These are typically errors in the adaptor code. When fixed these errors the cause of the remaining ones is more difficult to find, requiring deeper analysis [UtL06].

In short, MBT consists in generating and executing tests based on some abstract model of the SUT. Since the generation and execution of tests are done automatically, this is a very fast process requiring less effort and assistance in the processes of test case generation and test execution.

MBT is also good at SUT fault detection but it depends always on the skills and experience of who writes the model and chooses the test selection criteria. Utting refers that model-based testing is as good as or better in fault detection than manually designed tests [UtL06]. Model-based testing also improves the quality of the tests because its automation based on heuristics and algorithms makes the test case design systematic and repeatable. The quality of the tests is

then measured by model coverage. It also generates many more tests than manually test design simply by giving some new test selection criteria, or simply telling the tool that we want more tests. So the time and most of the effort needed to produce tests is only computational time and effort.

By writing the model, we can expose several requirements problems. Because MBT implies the clarification of the behavior of the SUT before writing the model, some requirements could be inconsistent and may be detected in the modeling phase. Also, writing the model can raise some questions like “What if input is out of range?” that can uncover some imprecisions in requirements. Utting refers that the modeling phase is like developing a prototype of the SUT and that approximately half of the failures are due to modeling or requirements errors [UtL06]. Finding requirements errors in previous phases is always cheaper to solve than in later phases of design or implementation where the cost of changing a requirement can be large.

When requirements change, in other approaches, we need to redesign and rewrite some of the test cases which results in a lot of effort done. In model-based testing, the only thing needed is to update the model and regenerate the tests which are automatically done. Since the model is smaller than the SUT, it takes less effort to update the model.

With traceability between requirements, the model and tests, we can know which transitions of the model are not covered by any test, which requirements are not modeled and we can identify tests that are not tested yet. So, traceability gives some explanation of why the test was created or why some transition exists in the model. Traceability allows to re-execute tests that implies some transition in the model. This can be required, for example, when requirements change and the model needs to be updated. Only the tests that exercise the transitions modified are executed, saving time.

But model-based testing does not offer only advantages. The main problem is the fact that the model can be quite different from the SUT itself and does not guarantee that all errors are found. The skills required to practice MBT are also a lot different than the other approaches. The phase of model design requires people that can design models, abstract from the application and need to be expert in the area of the application. Model-based testing also does not apply to all cases, and it is needed to check if MBT is the appropriate approach to use depending on the application. There are systems that are easier to test manually because they are not easily modeled or tests are not easily generated automatically.

For model-based testing, there are several tools that generate tests based on this approach. There are ones that generate tests from abstract tests, others that generate the tests from the model and create the oracles too (oracles consists on the expected results of the tests), and others that only generate input data from domain models.

2.6 Conclusions

In this section some processes for GUI testing were presented. Manual GUI testing is a completely manual process but it is still very used because some very specific applications cannot be automatically tested. Then script based approaches were presented but they still involve some manual work and have problems when tests have to be refined due to changes in the specification or requirements. They also cannot give an estimate of the SUT’s functionalities coverage. Finally, model-based testing is presented solving the problems of test generation automation and coverage analysis. Model-based testing still requires that the model of the SUT is created manually but, as it does not have all the complexity of the SUT itself, it is easier to develop.

3 Models, coverage criteria and tools for model-based testing

This section intends to show the different modeling notations that can be used in MBT. It will also present coverage criteria that guides us determining if a test suite is adequate. At the end of this section, tools for MBT are presented.

3.1 Models

Models are not very different from programs, therefore, test generation techniques can be applied. Black-box testing [Bei95] and white-box testing [Mye04] are the most common.

Model-based testing requires a model to generate tests and measure test coverage. The model can be built in several different languages/notations. The most used models are state-transition and pre/post models [UtL06]. State-transition models include finite state machines, state charts and UML state machines. Pre/post models can be purely declarative or executable.

Most of the models presented in next sections can be used for GUI testing. Statecharts and Finite State Machines (FSM) can be used for GUI testing although the use of FSM is discouraged for medium/large systems [Hor99]. UML state machine models can also be used because they are based on a reactive behavior like statecharts.

A pre/post notation was also used in [PFT05] for GUI testing. The model was done in Spec# with the aid of Spec Explorer to automatically generate test suites.

3.1.1 State-transition models

FSM are directed graphs with a finite set of nodes and arcs representing the internal system states and possible state transitions respectively. The model must have a finite number of states and transitions. They are useful to represent very small systems with very restricted domains because the number of states and transitions quickly explodes in the presence of variables with large domains (integers, strings, decimals, etc.) or in the presence of many variables [Hor99].

Statecharts [Har87, Hor99] are also a visual model that introduces concepts of hierarchy (nested state charts) diminishing the complexity and augmenting the abstraction level of the models, orthogonality allowing parallelism between states meaning that two charts can be in execution at the same time and the actions “entry”, “exit” e “throughout”. The “entry” action is executed every time the system enters in that state, the “exit” action is executed when the

system leaves the state. A “throughout” action is an action that is executed continuously while the system remains in the same state.

UML [UML09] is a powerful standard modeling notation that eases the communication between the customer and the team [BDG07]. UML state machines are essentially an object-oriented variant of state charts; they are used to define the state related behavior of existing classes. The class attributes represent additional state variables that can be manipulated in guard conditions and actions attached to states and transitions. This way, the number of states and transitions can be kept finite. Other UML diagrams can also be used for test generation and test specification, namely activity diagrams (representing control and data flows), for white-box test generation, and sequence diagrams (representing particular execution scenarios).

For white-box testing, as the test cases need to be derived using coverage criteria over the program (UML model), UML models can use state and transition criteria for white-box testing [BDG07].

For black-box testing, as it is a functional testing of the model, it is more applied to use case diagrams where the methods and classes specify functions in the sub-system. The use cases and methods identify the test cases that need to be created in order to cover functionally the system. In this case, the focus is in the output depending on the input and execution conditions [BDG07].

UML Testing Profile (UTP) [OMG02, UML09] is a specialization of UML (extends and restricts the language). UTP provides concepts (like the addition of stereotypes) that target the development of test specifications and models. These concepts define a modeling language for visualizing, specifying, and documenting the artifacts of a test system. Mappings from UTP to JUnit can be found in [BDG07].

3.1.2 Pre/post models

For this type of models, we have two kinds of modeling: those used to complement state-transition models and those that can completely model the system. A pre/post notation that can be used together with UML models is the Object Constraint Language (OCL) [OCL09]. It allows complementing the UML models with important semantic information: pre and post conditions of class methods, class invariants and guard conditions for the UML transitions. Designed and conceived based on object oriented concepts, OCL has some basic and easily understandable concepts for programmers but has also some very complex operations leading to a big learning curve of all its features and capabilities.

Other approaches of pre/post models that describes the system completely allow the definition of the body of the actions, besides the previous definitions of pre and post conditions, using a high-level language like Spec# [SpS09] or VDM [VDM09]. The body of the actions shows how actions will affect the class attributes.

3.1.3 Other modeling notations

ConcurTaskTree (CTT) [PMM97] is a notation that uses task modeling of the system, representing its hierarchical structure in a tree form. CTT also allows the description of concurrent behaviors. This notation is used to model interactive systems but they are not usually used for MBT. Extensions of this notation for use with UML notations can be found in [MoP08].

Memon defends the use of event-flow graphs [BrM07, MSP01]. These graphs can have probabilities associated based on observed usage of the system. The nodes of the graph represent the events and the edges the event-flow relation between states. The graph gives an idea, in each state, of the events that the system can accept. This approach can be cyclic because events can be executed more than once.

Other modeling notations can include Abstract State Machines (ASM) [GaR01] for testing purposes. ASM are a well defined pseudo-code defining abstract structures. “The states of ASMs are arbitrary structures in the standard sense they are used in mathematical sciences, i.e. domains of objects with functions and predicates depend on them. The basic operations of ASMs are guarded destructive assignments of values to functions at given arguments” [GaR01]. For detailed mathematical definition on ASMs, see [Gur00].

3.2 Coverage criteria

Coverage criteria are essential in MBT, as in other systematic test techniques, to evaluate the quality of a given test suite and guide the generation of test cases [SeG05, GaR01]. “Tests that are adequate with respect to a criterion cover all the elements in the domain determined by the criterion” [AFG03]. A coverage criterion must be objectively measurable (like the portion of the model or program that is exercised by the tests) and should be a leading indicator of test adequacy, that is, the capability of the test suite to reveal “most” of the defects in the SUT [ZHM97]. In spite of some formalization efforts [Bur03], in practice testers usually consider a test suite adequate when additional tests do not find additional errors [Wei89].

If the tests derived from requirements performed to the SUT do not fail, then we can have a high degree of confidence that the system fulfills its requirements [BDG07].

In most real world applications, system requirements cover a large set of possibilities which cannot be tested because of project's time and cost. Therefore, criteria are defined to help determining what and how many tests are needed to achieve adequate testing (i.e. coverage criteria, software reliability criteria) [Cab76, Mye79].

3.2.1 Structural model coverage

In MBT one is interested in generating test cases that cover the most important model elements and, at the same time, are capable of revealing the most common errors that are introduced in their implementation. Several coverage criteria have usually to be combined for that purpose.

To verify if the model elements (states, transitions, decisions, etc.) are tested, both control-flow and transition-based coverage criteria can be used.

Control-flow oriented coverage intends to exercise decisions in the model. These decisions can be found in both modeling kinds (state-transition and pre/post). In state-transition models we can use these criteria to evaluate the guard conditions that can appear in the transitions while in pre/post models we can use it to exercise all the branching decisions, pre-conditions and post conditions of the model. Control-flow oriented coverage criteria includes [UtL06]:

- **Statement coverage** covering all the statements of the model. This is the weakest coverage criterion.
- **Decision coverage** requires that every decision in the model takes true and false values at least once, requiring only two tests for each decision. However, it is weak because, i.e., for the decision (X or Y), the test cases (X=True; Y=False) and (X=False; Y=False) will make the decision true and false but the test suite does not exercise the effect of the Y variable in the decision.
- **Condition coverage** assures that each condition in a decision takes true and false values at least once. In the previous example X can be true or false and so does Y. A possible test for this decision could be (X=true; Y=false) and (X=false; Y=true). Although all conditions assume all possible values at least once, the case where the whole decision is false is not exercised.

- **Condition/decision coverage** appears as a combination of the two previous criteria, solving the previous problem. Using the previous example, (X=true; Y=true) and (X=false; Y=false) will cover this criteria because the conditions assume both true and false values and so the final decision but it does not distinguish the decision (X or Y) from (X and Y) which in this case will have the same result.
- **Modified condition/decision coverage** increases the previous criterion's coverage with an additional requirement that is to show that each condition affects independently the result of the decision. Generally, it requires *n+1* test cases for a decision with *n* conditions. To test the example (X or Y), the tests (X=true; Y=false), (X=false; Y=true) and (X=false; Y=false) will be enough.
- **Multiple condition coverage** is the strongest and the most impracticable. It requires that all combinations of conditions are executed at least once. For a decision with *n* conditions, 2^n tests must be done. In the case of the example, we will have 4 tests because there are only two conditions. The tests to cover this criterion are (X=false; Y=false), (X=true; Y=false), (X=false; Y=true) and (X=true; Y=true)[HVC01].

Transition-based coverage criteria: this is used to state-transition models to cover the elements of the model. Transition-based coverage criteria includes [UtL06]:

- **All-states coverage** requires that all states are visited at least once.
- **All-transitions coverage** requires that all transitions of the model have been covered by some test. It can be different from all-states coverage for example when there are two transitions that have the same source and target state. The first criteria require only using one of the transitions to visit the states while the all-transitions require that both are used in the tests.
- **All-transition-pairs coverage** requires that every pair of adjacent transitions is executed at least once in the test suite.
- **All-loop-free-paths coverage** says that every loop free path must be executed at least once.
- **All-one-loop-paths coverage** is the same as the previous with the small difference that it must visit at most two loop executions.
- **All-paths coverage** is the most extensive testing criteria and requires that all paths are traversed at least once. This is almost impracticable because the model can have too many paths and loops to test requiring a giant or infinite test suite.

Brooks and Memon refers that for event-flow graphs, the coverage criteria defined for test generation are exercising the most probable sequence of events [BrM07].

3.2.2 Data coverage

Data coverage has the purpose of choosing some good and representative data to include in the test case since there are infinite possibilities (taking for example some text box where the input is a decimal number). A data coverage criterion helps cutting a portion of the possible values depending on the approach selected. This criteria is usually combined with structural model coverage [UtL06]. The most representative criteria to select values includes:

One-Value: One value criterion requires that only one value is chosen from the domain to perform the test. This criterion might seem useless but it becomes more powerful when joined with some other criteria, reducing the amount of tests to perform.

All-Values: This criterion tests every possibility of the variable. The variable can assume every value included in the domain. This is impracticable if the size of the domain is huge (i.e. any decimal interval). This criterion is good for a specific domain where all possibilities are listed and its size is small (for example: an enumeration of sexes available to choose: “Male” and “Female”).

The previous approaches are not always the best to consider because we may want to have more than one test but not the whole domain being tested. Also, the approaches only are good for a single variable that ranges in a domain. The following criteria can not only be applied to a single variable but also to a set of dependent variables combining their values intelligently, reducing the number of tests to perform.

Boundary Value Testing: This criterion consists in choosing the values that are in the extremes of the domain. Utting refers that boundary testing has some justification because there are lots of faults located at the frontier between two behaviors present in the SUT [UtL06]. This approach is applied to ordered domains whether they are numbers or some user-defined domain since there is some form of ordering the data contained in the domain. In case that we have multiple variables to test together, other decision criteria can be chosen such as:

- **All-boundaries** coverage that retrieves all possible combination of the variables. Using the (x, y) variables and their domains of integer values (Figure 3.1) all-boundaries criteria will require all (17) points drawn in the figure. This can become a huge set of values because it tests every point that satisfies the boundary.
- **Multidimensional-boundaries** coverage requires that each variable has assigned its maximum and minimum value at least once in the tests. For the example in Figure 3.1 the points (-3,-3), (0, 3) and (3, 0) would suffice.
- **All-edges** coverage requires that at least one point of each edge is tested. In this case it only will require two points to satisfy the condition (-3, 0) and (0, -3).
- **One-boundary** coverage requires only one test and it should be a random point of a random boundary. We can choose for example the point (2, 2) to cover this criterion.

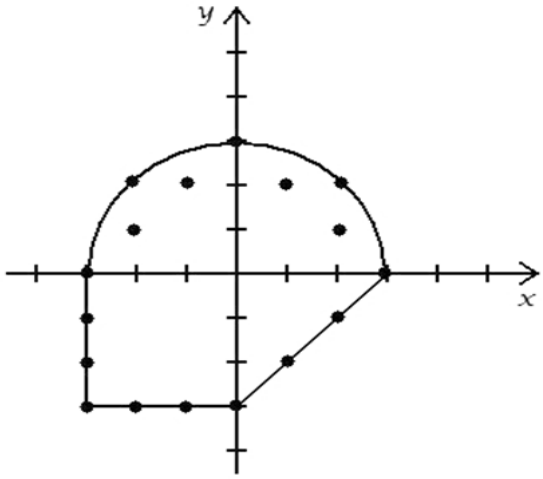

Figure 3.1: Integer boundary points of a complex domain.

Statistical data coverage: this method uses a distribution *D* to select random values to be tested. With this approach we have to give the distribution of the variable (normal distribution, uniform distribution, among others) and then we can obtain the amount of tests that we desire. For example with boundary value coverage we have a set of tests and we want to add some more. We can implement statistical data coverage to complement test suites generated using previous criteria, giving more values to execute the test suite.

Since choosing the values is done randomly (depending on a distribution that can change the probability of some values being selected) this approach can select data even from unordered domains. In case of specified domains (example of sexes Male or Female) we can weight each possible value giving more or equivalent weight to one of the variables (i.e. Male=45% and Female=55%).

In **Pairwise testing**, two related variables are tested to verify if the behavior of them working together is the expected. This assumes that in most cases, the faults come when trying

to combine two variables containing certain values. There are also other approaches that differ from pairwise testing, instead of duets of variables, the use of triplets and quadruples of variables are used. Those are less used because the number of tests would increase significantly.

3.2.3 Fault model criteria

The main goal of software testing is to find errors/faults in the application. Fault based testing is used to test the absence of some common errors. They are generally based on specific models that are used to specify test data.

Appendix A was taken from [KFN99] and shows common GUI errors encountered in several applications.

Fault testing is sometimes supported by mutation testing. Mutation testing tries to solve the problem of not being able to measure the quality of the tests. Mutation testing assumes that we have a test suite and that we have a program that passes the test suite. Then, one minor change in the code is done (creating a mutant program) and the test suite is run against the mutant program. This step is done several times. One of two things can happen to each mutant: the test suite detects the change in the code and the mutant is said killed; it can also happen that the test suite does not find the change in the code and then we have an equivalent mutant or an inadequate test suite.

The ratio of killed mutants gives an idea on how robust to changes is the code and how good our test suite is [Mar06].

3.2.4 Standard testing checklists

The coverage of predefined checklist like the one presented in [Baz09] can also be used as coverage criteria.

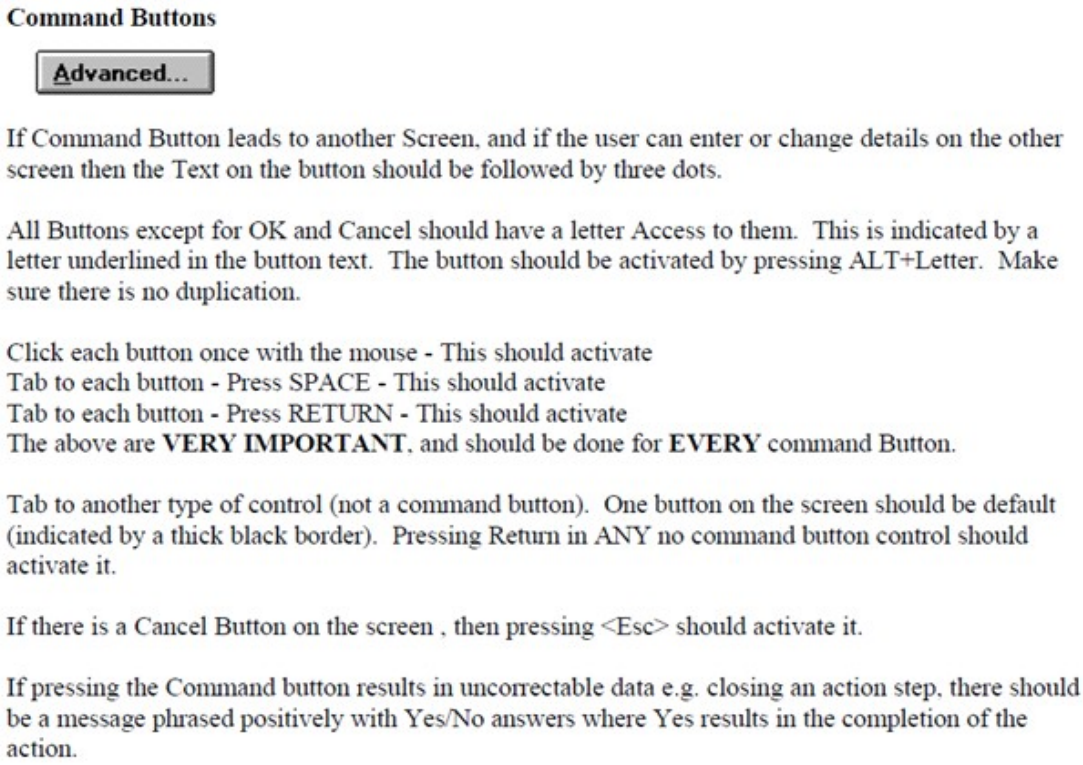

Figure 3.2: Steps for button testing [Baz09].

In this checklist the basic steps to test controls in the applications are detailed so testers know what to do without doubt. Figure 3.2 is a portion of the file only for exemplification purposes showing how to test a button on interfaces, showing a lot of possibilities to activate it. It does not say what should be the behavior of the button activations because that is dependent on the application itself. Ways to test other controls can be found in the complete checklist.

3.3 Tools for MBT

There are some tools that actually supports MBT to model and test the systems. Here will be introduced a short list of the tools found.

Spec Explorer [SpE09] is a tool that uses pre/post models to model the system. The model is implemented in a language called Spec# [SpS09]. This language extends C# with contracts (invariants, pre/post conditions, actions, probe actions), value-based collection types and other features. The methods annotated with the "Action" keyword represent the externally callable update to the variables, for testing purposes. These actions can be of two kinds. The first is a probe action and must be annotated with "Action (Kind=ActionAttributeKind.Probe)". This kind of action represents queries to the class attributes that can be called for testing purposes. The other kind of action is the Scenario action. This is annotated using "Action (Kind=ActionAttributeKind.Scenario)" and allows the definition of test scenarios. Besides allowing the definition of manual tests, the tool is also able to generate a test suite. But, it cannot generate the tests directly from the pre/post model. To generate the test suite, the tool generates first an FSM and then generates the test suite using transition-based coverage criteria.

An interesting tool to implement MBT is ParTeG [Par08]. This tool is a free plugin for the Eclipse Framework [Ecl09] that allows the creation of UML state machine models visually and then generates a test suite using selected coverage criteria. The tool provides the basic controls to create a complex model. The test suite generated can be executed using JUnit [JUn09]. The tool can also generate mutation tests and, when executed, shows how many of the mutants were killed and how many remain alive, revealing how robust the model is. The tool also includes two examples (elevator control in Figure 3.3 and sorting machine) to better understand how to use it.

Conformiq Qtronic [QTr09] is a commercial MBT tool that uses as input a UML state machine model, complemented with textual specifications in a Java [Jav09] like language named QML. The tool supports several coverage criteria for test generation (transition coverage, boundary value analysis, requirements coverage among others). This tool can also be integrated with Eclipse to generate test suites to the model defined. The model has to be done in Conformiq Modeler tool. This tool also provides traceability matrices between coverage goals (structural features and high-level testing requirements) and test cases. It also provides message sequence charts of the test cases.

Esterel Technologies also commercialize a tool named Model Test Coverage (MTC) that "measures the coverage of the design reached by a high-level requirements-based test suite. MTC thus verifies if every element of the SCADE model (i.e., every software design feature) has been dynamically activated, and makes it possible to detect unintended functions" [MTC09].

The last framework presented is the GUITAR (GUI Testing frAmewoRk) project. "GUITAR is a suite of models, components, and tools for automated testing of software applications that have a Graphical User Interface (GUI) front-end" [GUI09]. This framework allow to create test cases from Java and Windows applications detecting software crashes. It also allows to consult the structure of the application, its properties and capture test sessions to execute later.

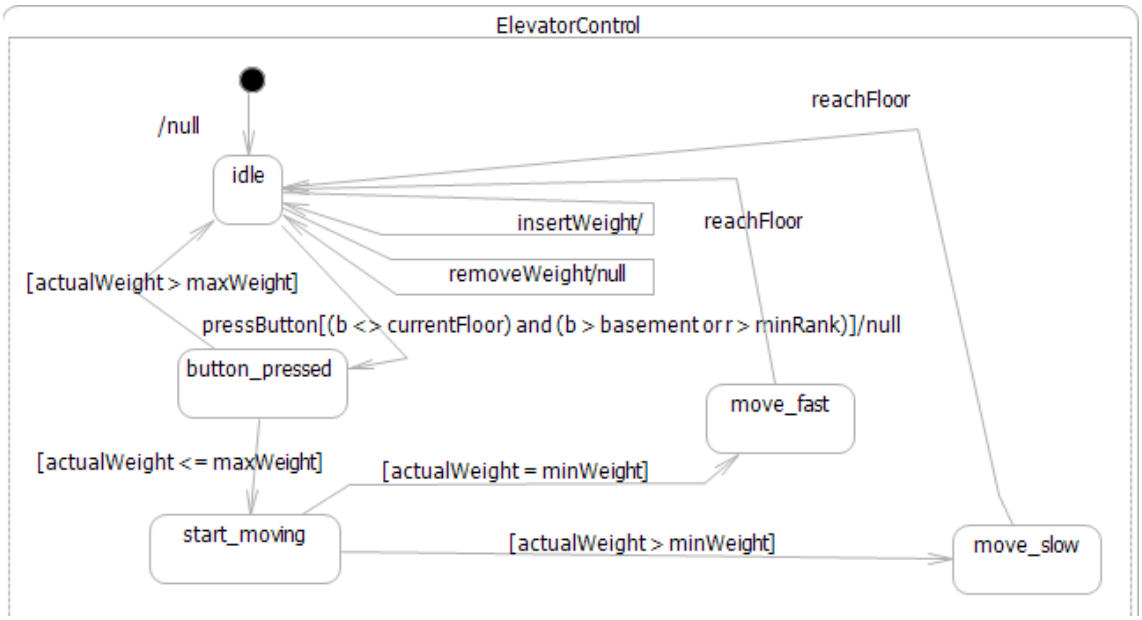

Figure 3.3: Elevator control sample included in ParTeG tool.

3.4 Conclusions

In this chapter were presented different kinds of models to use with MBT. The best models are state-transition models because they have a visual component and can be understand by most of the people. In these models, UML is valued because it is a standard. Pre/post models require more skills to implement and to understand the whole model.

Different coverage criteria can be combined to obtain a better test suite. Control-flow can be used in all kinds of models as well as data coverage and fault-based criteria. Only transition-based criteria are applied to state-transition models. There are also checklists that give the tester all the steps required to test successfully a specified control/element.

Finally, several tools were presented. Some of the tools use pre/post models and other use state-transition models. They all lack model coverage analysis features, although most of them have test generation based on selected criteria.

4 Experimentation and comparative assessment

This chapter has the intention to present a comparative assessment of some modeling notations and related test coverage criteria. The notations used are pre/post model using Spec#, finite state machine, using Spec Explorer to automatically generate it from pre/post model, and UML notation. Finally, the results of comparing the systems are presented. The main goal of this section is to gain quantitative insight into the strengths and weaknesses of each approach used.

4.1 System requirements

The example proposed to illustrate and assess the modeling notations and coverage criteria previously presented will consist of a simple alarm system with the following requirements:

- The system has an interactive control panel that is connected to a siren and multiple sensors;
- When the system is deactivated, the only thing that the user can do is to activate it by pressing a button in the control panel of the system;
- If a sensor is activated while the system is activated and the system is not deactivated in the meanwhile, the siren rings after a specified wait time, shutting down only when the system is deactivated;
- To deactivate the system, the user has to insert a PIN code in the control panel;
- If the user consecutively inserts a wrong PIN code a certain number of times, the siren is turned on and the system becomes locked (cannot be deactivated).

4.2 UML state machine model

A UML model for this system is shown in Figure 4.1 and 4.2. The alarm class in Figure 4.1 contains the attributes and methods that are used in the state machine of the Figure 4.2. The exception are functions that are included in the UML definition such as “after(time)” time event. In this case, the methods represent events of the system. Since the distinction between impossible and no-effect events is very important for test generation, it was followed the convention of showing all the events that can occur in each state, even if they have no effect on

the system state. The transition triggered by a temporal event that departs from state “Sensor Activated” was drawn with the assumption that the time counter is not reset by transitions from that state to itself. Also, all the attributes of the class were considered changeable although maxTry could have been constant. Also, methods to change the expected PIN code could be added to the class, but they will not interfere with the model and as it is not a requirement it was not added.

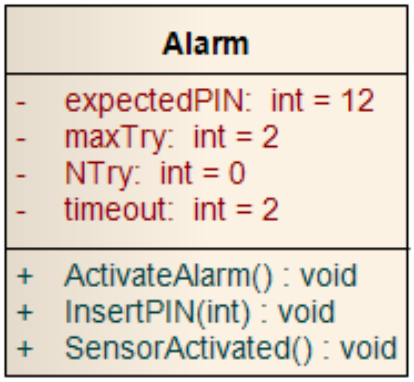

Figure 4.1: UML alarm class.

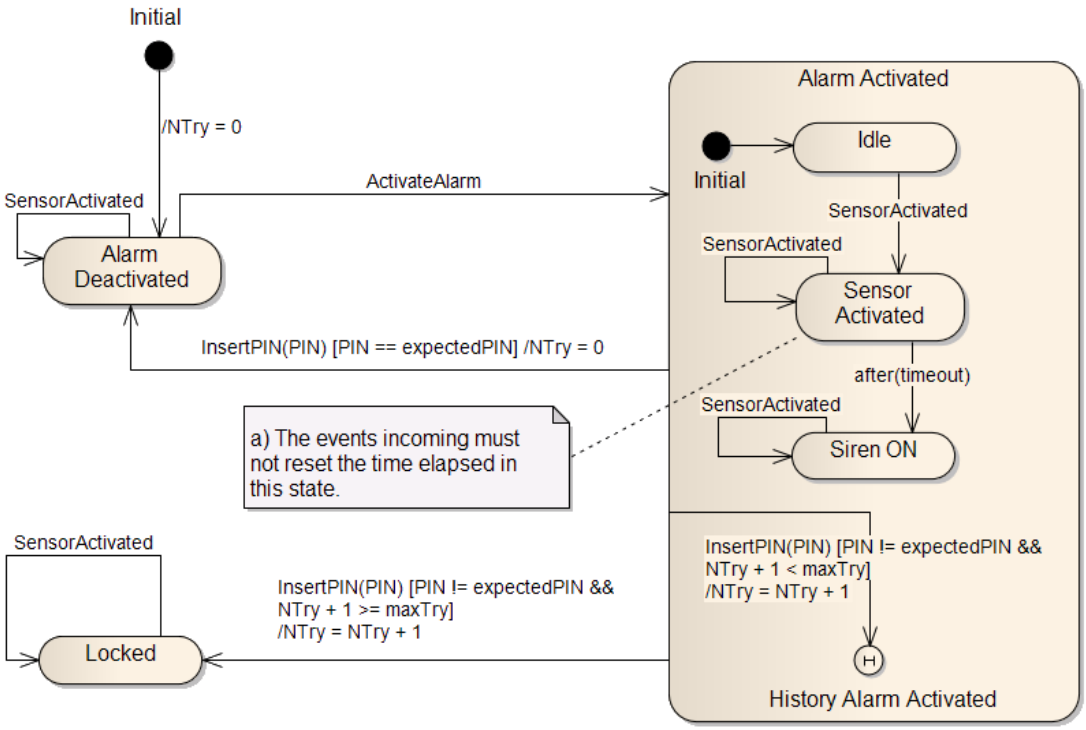

Figure 4.2: State machine diagram of alarm system.

4.2.1 Test cases derived from the UML model (using transition coverage)

Six tests have been manually derived from the UML model with a total size of 31 steps. The tests created cover all transitions of the exploded state machine (Figure 4.3) that results from exploding the composite state present in the model of the Figure 4.2. The complete test

suite can be consulted in Appendix B . The “Wait(timeout)” action represented in the tests simulates a wait behavior by the test drive or user. All other actions are considered atomic with a negligible execution time. The assert command checks if the system is in the expected state of the state machine or else it fails, leading the test to fail too.

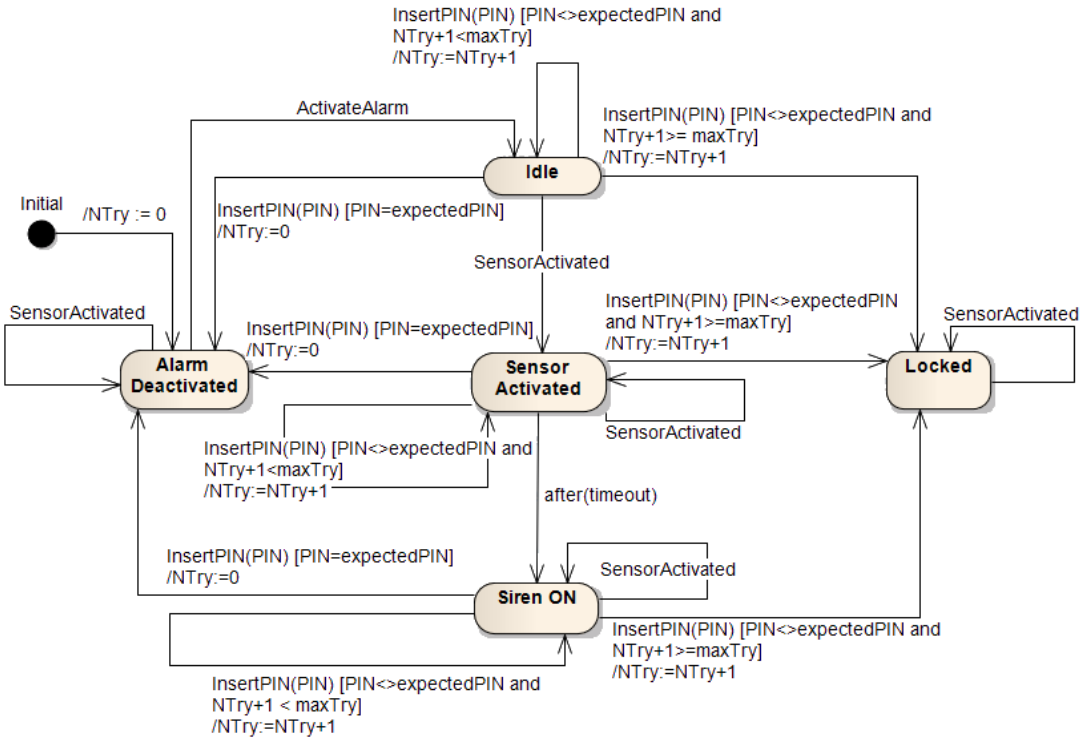

Figure 4.3: Exploded UML model of the alarm system.

4.3 Pre/post model

A corresponding model of the system in Spec# can be consulted in Appendix C . The first lines of the model define some constants and class attributes. Restrictions to the model are defined in the invariant declaration section.

The methods preceded by the “[Action]” keyword represent the events that the system accepts: user events (“ActivateAlarm”, “InsertPIN(PIN)”), sensor events (“SensorActivated”) and time events (“Wait(time)”). Each method has a pre-condition that indicates when the event is available to the system and can occur. The body of the methods represents the effect produced on the class attributes. As we could not handle time in Spec#, it was created an additional action named “Wait(time)”. This action has the purpose of simulating elapsed time during the execution of the model.

Also were defined probe actions that allow us to “peek” the internal state of the system variables. So, we have defined probe actions for all the attributes that change their value during the execution of the system. These will generate assert verifications in the test cases generated.

4.3.1 Manually derived test suite from pre/post model (using condition/ decision coverage)

As Spec# allows the definition of test cases together with the model definition, we manually derived and defined a set of test methods that cover all the conditions and decisions of the model. This criterion was applied to the control-flow instructions and to the pre-condition of

all the actions. To achieve this coverage criterion, only 5 tests were needed. Some of the tests defined were supposed to throw an exception because they were created with the intention to violate the pre-condition of the action. The complete test suite can be consulted in Appendix C together with the model.

4.3.2 Automatically generated test suite from pre/post model (using transition coverage)

To automatically generate a test suite, Spec Explorer requires that a finite state machine has been generated. The FSM generated in this case is shown in Figure 4.4. Due to the very large domain of integers, the PINs that the user can enter were restricted to {10, 12} (one correct and one incorrect value). Also the parameter of the "Wait" event was reduced to {1, 2}. Since it is a small model with a small FSM, it was easy to see that the behavior represented is the expected one. Next, the test suite has been generated. Spec Explorer uses transition coverage to automatically generate the test suite. The test suite generated contained 16 test cases with a total size of 73 steps without assertions.

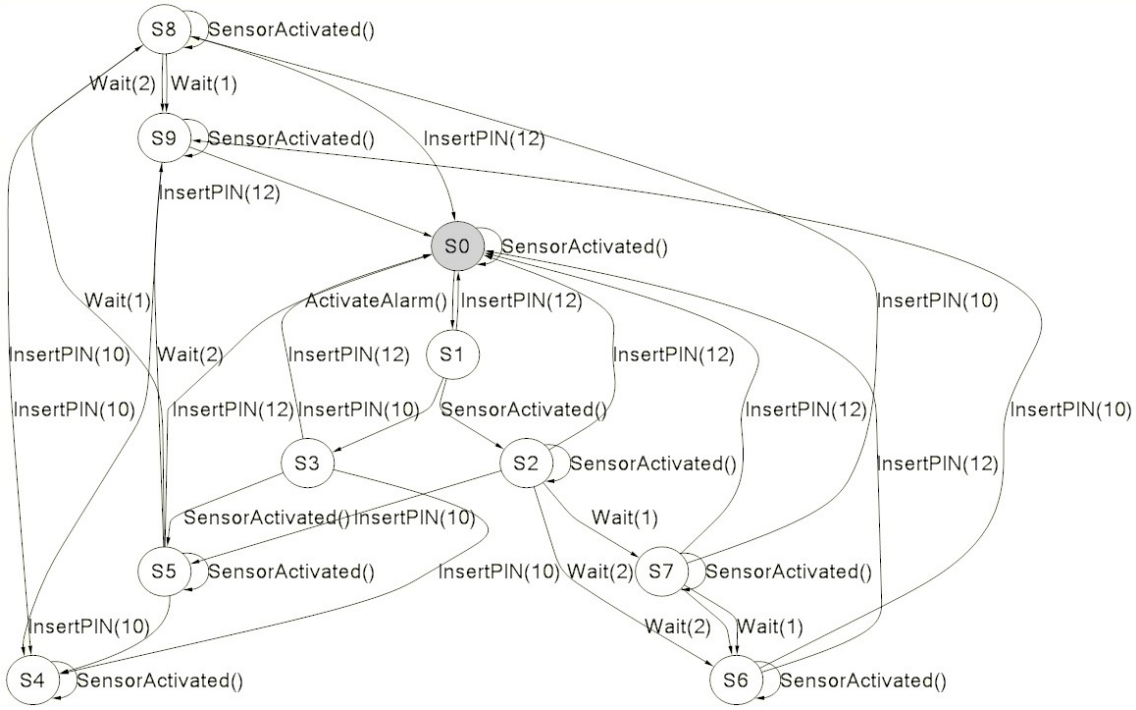

Figure 4.4: FSM of the alarm system.

4.4 Comparative assessment

This section intends to compare the approaches used previously to model the system defined. The comparison criteria are also referred.

4.4.1 Assessment criteria

The comparison criteria used in this assessment were the total size of the test suite, the effort needed to create the models, their size and the coverage of test goals defined empirically. The goals were defined taking into account the initial requirements and most probable faults that can happen in this system.

4.4.2 Comparative assessment

To assess the adequacy of the test suites derived from the models, a set of empirical objectives that the model should implement and the tests should exercise were defined. The main objectives are presented in Table 4.1. The other columns represent if the tests done to the model cover these objectives.

We can see that all approaches are quite similar in terms of coverage of test objectives. The worst approach was white box testing that could not test the objective 4 and the objective 5 was only partially covered because not all the possibilities to lock the system were tested. For all approaches there were two objectives that were not tested. Checking if the siren is always on is almost impossible because we cannot check that the siren remains on forever. As for the test objective 9, this would require chaining test cases without resetting the internal state of the system. An option could be creating a test case that deactivates the system twice.

Another interesting comparison between the models could be the size of the test suites obtained. Approaches that generates smaller test suites, while covering the same test objectives are better for scalability reasons. Table 4.1 shows the number of test cases and its total size (number of steps) of the approaches used. The FSM has originated much more tests because the number of transitions to cover is also larger than in UML model. Also, in the FSM approach, the domains of the variables were restricted so the FSM does not become too large but can cover all possibilities. The advantage of the FSM is that it is completely automatic, after choosing the right domain values to explore. The main problem is the quick state and transition explosion. For example, if one have used a more realistic value of 3 for “maxTry” (instead of pre-defined 2), the number of states would increase from 10 to 13 and the number of transitions would increase from 35 to 51. The size of the test suite will increase because the number of transitions to cover is now larger. To limit this explosion, the domains of the variables has to be strongly restricted, raising the risk of not testing important features.

Table 4.1: Coverage of empirical test objectives in the approaches used (V-fully covered, P-partially covered, X-not covered). Numbers in parenthesis indicate section numbers.

Empirical test objectives	**White box (4.3.1)**	**FSM (4.3.2)**	**UML (4.2.1)**
Number of tests cases / Size of test suite	5/15	16/73	6/31
1. Siren rings after a sensor was activated and “timeout” seconds passed.	V	V	V
2. Alarm is deactivated when correct PIN is inserted.	V	V	V
3. Possibility of inserting PIN after sensor has activated and “timeout” seconds hasn’t passed.	V	V	V
4. Deactivate system before a sensor is activated.	X	V	V
5. System becomes locked when wrong PIN is inserted “maxTry” times.	P	V	V
6. Check that siren is always ON until deactivation of alarm.	X	X	X
7. Timer is not reset when sensor is activated again.	V	V	V
8. Attempts left (“NTry”) aren’t reset when a sensor is activated.	V	V	V
9. Attempts left (“NTry”) are reset when a correct PIN is inserted (deactivating the alarm).	X	X	X

Another comparison between the different approaches has to do with the effort to create the models (the effort to generate the test suite is not considered, because test generation could

be automated in all approaches). Since the effort is usually related with the size, the size of the models were also compared. In this example, after deciding which actions the system must implement and its internal structure, the pre/post model's size and development effort were not significantly different from the UML model.

4.5 Conclusions

In this section a comparison between modeling notations was performed using a common system. In UML and white-box models, the test suite has been manually generated. For the FSM, Spec Explorer generated the test suite automatically.

The evaluation criteria used in this assessment were the total size of the test suite, the effort needed to create the models, their size and the coverage of test goals defined. These test goals were defined using the requirements as a basis and usual faults that this system can have. Other evaluation methods could be used such as fault injection.

Analyzing the final results according to the defined criteria, we can see that the UML modeling approach was the best in this example. The UML approach needed less tests than FSM to completely cover the model defined. As for white-box approach, it covers less test goals than the other approaches. The size of the models are quite similar, and so their creation effort.

This assessment, although it is small and not all functionalities of the modeling notation are represented, it allowed us to see some situations where automatic test case generation could have problems generating test cases. One of the issues is related with temporal actions. They are difficult to simulate. The case of the 6th test objective defined previously is a special case because it would be impossible to fulfill completely, even with manual testing (this objective is untestable). Other issues are related with the “intelligence” of the test case generator. For example, the 9th test objective would be easily covered if the generation of test cases does not stop when reaching the initial state. This objective could be easily tested activating and deactivating the system twice (inserting wrong PIN codes while activated). Because of this, some manually defined test cases can be defined to complement the automatically generated test suite. The model can also contain errors and can be impossible to reach some elements of the model. The tool developed and presented in section 5 allow the tester to see which elements were not tested, allowing him to add tests to exercise those elements or modify the model in case the elements are unreachable.

With other examples we could probably find more issues that automated test generation has. This example can be scaled to other sizes and, sometimes, the use of UML state machines will be discouraged because it can become more complex to create and test than other of the modeling notations.

5 Development of a model coverage analysis tool

As this project requires Spec Explorer to generate the test suite and it is necessary to restrict the domains so it can generate the minimum test suite, it is needed to assure that the test suite generated cover the initial model. The tool presented here will address that problem.

So, this chapter intends to present the coverage analysis tool implemented. The basic requirements of the tool, its working principles and algorithms are presented. Also, the organization of the input models and a reference manual to use the application are shown. Finally a more detailed architecture of the system is explained.

5.1 Context, objectives and requirements

The application developed is a part of a bigger project presented in Figure 5.1. The whole project aims to improve software testing. This project adopts the concepts of model-based testing to deliver better software. For this application specifically (Test Coverage Analysis), we will need as input a model of the system to test. This model of the SUT will then be converted automatically to a Spec# model so Spec Explorer tool can automatically generate the test suite. To see when test generation is enough Test Coverage Analysis tool paints the model to see if the test suite has already covered the whole model or if the addition of more tests does not cover more parts of the original model. When the test suite is marked as completed, it will then be tested against the SUT. The tool developed, in a first iteration of the application, will not interact directly with Spec Explorer but receives directly the test suite as input to the application as well as the model of the system. In a later phase, the integration with Spec Explorer is required to guide the process of test generation.

As seen previously, there are no tools that can give feedback about the coverage of a model, so the application developed intends to give some feedback about the coverage of the test suite generated to see if the test suite is appropriate or if there is the need to generate some tests to cover as much as possible the model. Another use of the tool could be to select the minimum test cases that cover the model because, as seen previously, Spec Explorer generates more tests than needed to cover all the transitions of the UML model.

The main requirements to this project includes:

- (Essential) The coverage criteria selected should be adequate for GUI testing
  - In order to assure confidence in the ability of the test suite to discover defects
- (Essential) The coverage criteria should be expressed in terms of model coverage

    - The coverage criteria will be used to guide test generation, even before an implementation exists
    - Yet, we recognize that code coverage is also important, e.g., for covering exceptions
    - Depends on the modeling notation selected in the project
- (Essential) Represent mainly in a visual way the coverage of the model
    - It is easier to visually analyze the painted model than a coverage report
    - The main priority is to feedback as much as possible painting the model. For the parts of the model that can't be painted, we can generate a report specifying which parts of the model were not covered and painted.
- (Essential) Coverage should be computed automatically
- (Desirable) Coverage criteria should be fulfilled automatically
    - Ideally, the test generation algorithm should be guided by the coverage goals
    - This is desirable because it is hard to generate test suites that completely cover the models

Essential and Desirable are keywords to shows the relevance of the topic. In the first topic there is the question of definition of adequacy. Adequacy refers to the capability of the test suite to reveal “most” of the defects in the SUT.

Also there is the problem of ambiguity in the requirements definition. This ambiguity can become a barrier to achieve a good test suite adequacy.

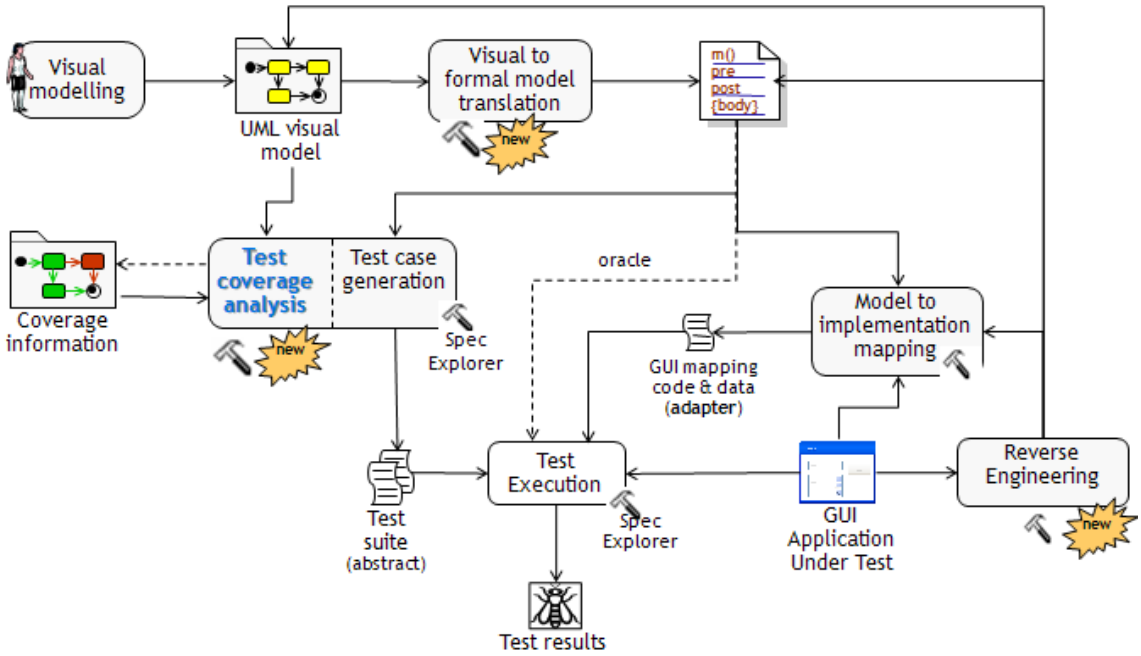

Figure 5.1: AMBER iTest project.

5.2 Tool overview

This tool has the intention to represent the model coverage achieved by a given test suite in the initial model. To do this, its elements will be painted with colors representing its coverage. Enterprise Architect is the tool used to create the models and export them into a XMI file. The models loaded by this tool are state machine models and class diagram models. Each state machine must be associated with a class from the class diagram. The language used in expressions is OCL. Some extensions were added such as the attribution operator (“:=”) for the actions of the transitions. Some functions and constants were also added to give a wider range of possibilities to the user. The constants added were *pi* (“PI”) and *neper* number (“EXP”). The

functions are square root (“sqrt(value)”), cosine (“cos(value)”), sine (“sin(value)”), and string length (“size(string)”).

The test suite is generated and exported by Spec Explorer. The tool will then receive both files (the model and the test suite) and generate a file that can be imported back in Enterprise Architect for further analysis. Figure 4.2 shows an example of initial model and Figure 5.9 and Figure 5.10 are examples of painted models. From them, we can see if more tests are needed to completely cover the model or not and which elements needs to be exercised by the added tests.

5.3 Working principle

As the main purpose of this application is to analyze and identify the portion of the model exercised by a test suite, this tool needs to have as input both files (the exported model file from Enterprise Architect that is the model editor and the test suite exported from Spec Explorer that will automatically generate the test suite).

The model will then be executed by a given test suite. The tool will seek the first state of the state machine and starting from there the execution of the tests will explore the model through its transitions and states. Finally, the tool is able to export the modified model to analyze its coverage in Enterprise Architect.

Other approach to obtain the same result could be using the states defined in Spec Explorer’s FSM and map them to states present in the UML model. This is actually possible because with each transition, Spec Explorer associates start and destiny states. The main problem with this approach will be to match the states in the FSM with the states in the UML state machine model. The matching can be achieved through the values of the class attributes. In the case of the example presented in the previous chapter, the mapping could be done through the “state” attribute used in Spec# that has the name of the corresponding state in the UML model.

For this application we have chosen the first approach because, in a later phase of development, the tool can also include an extra module able to generate a test suite through the analysis of the transitions present in the model. Also, if a test case is manually created, there will be no need to worry about the states of the state machine. The only thing required is the steps that the test case has.

5.4 Enterprise Architect model organization

For the tool to interpret the model, it must be created considering a few aspects. These will ensure that the file model is correctly created and that the application can correctly analyze and interpret the file. The aspects to take into account during the creation of the model will be described now.

The first precaution to have is that the main state machine of the model must be under a class. This class will give the attributes and methods that can be used inside the state machine model. The application supports several classes but only the state machine of the first one will be tested. Additional state machines can be used but all the states that compose this state machine have to be under it (in the tree definition). This will allow distinguishing the state machine where the state belongs. As an example, the tree of the alarm model should be like in the Figure 5.2.

The transitions of the state machine model can contain methods with parameters, guard conditions and actions. Transition methods are calls to the methods defined in the class. Besides those, only “after” is interpreted by the application. Each transition has only one method call. To represent the methods in the model, they do not need to appear with its parameters. The only restriction will be to the “after” function that needs to know previously the time to wait. The parameter of this method can be a simple expression using class attributes and predefined

keywords. To restrict the parameter of the transitions method, just add the condition to the guard condition of the transition.

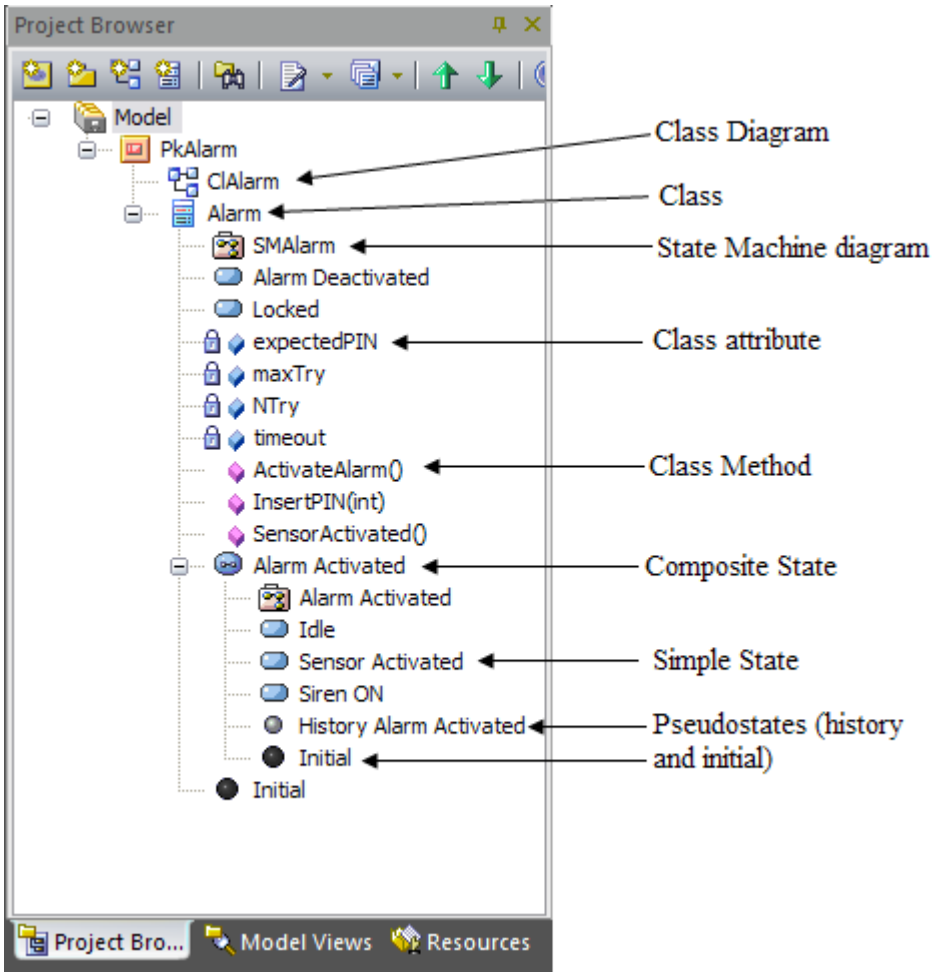

Figure 5.2: Organization of the input model.

Guard conditions can have references to the values of the attributes of the class and to names of the method call parameters. Guard conditions are expressed in OCL. The conditions can also have predefined variables such as “PI” and “EXP” representing respectively the PI number and *neper* number. It also can have references to predefined functions such as “sin(*value*)”, “cos(*value*)”, “size(*string*)” and “sqrt(*value*)”. The transition action allows the modification of the value of a class attribute. To assign a value to a variable is used the symbol “:=” to distinguish from the equality symbol.

The initial state of each state machine must have the name “Initial” to be interpreted as being the initial node. This state must also be under the proper state machine. With history states is the same thing. The name must contain “History” and the state must be located under the respective state machine. Other symbols that aren’t states, transitions, state machines, initial states, history states and classes are ignored when loading the model. These can be inserted without any effect in the application.

Another important care is about the coherence with the methods calls defined in the test suite. If the test suite has been automatically generated by Spec Explorer, then the coherence must be with the actions of the model. Here, the methods must have the parameters without any expression presenting only the values required. When testing the model, the tests are case sensitive so, the method name has to be exactly the same as defined in the UML model. In case the test suite is done manually, it must respect the template of the XML file exported from Spec Explorer tool shown in section 5.8.

5.5 Instructions to use the application

Before we can use the application, the model and the test suite must be exported to a XML file. After the creation of the model using Enterprise Architect, we need to export the model using "Project->Import/Export->Export package to XMI" (see Figure 5.3). The window in Figure 5.4 will appear.

The options selected are the required to successfully export the file. After the model is exported, we need to export the test suite using Spec Explorer. To export in Spec Explorer, the FSM has to be generated and so the test suite to that FSM. After that, select the option "Tools->Export XML as…" select the name of the file and its location and, when asked about what to save, the required is the "Test Suite" although the states and transitions can be saved too without influencing the file.

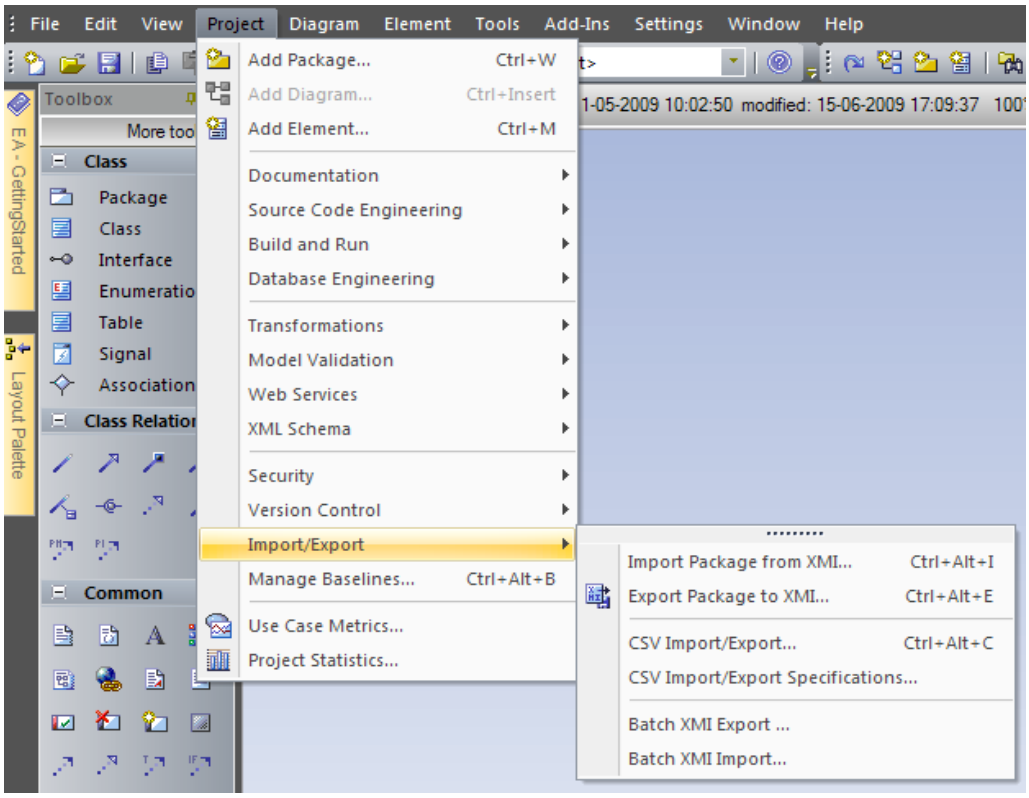

Figure 5.3: Import/Export menu in Enterprise Architect.

After both models are exported, the only thing left is to load them into the application Figure 5.5. To do that click in the button with "…" that gives respect to the model and to the test suite. This will open a File dialog where we select the file pretended. After both files are selected, the "Load!" button gets enabled, allowing the user to click on it Figure 5.6. This action will load both models, check the coverage of the model and paint the model. The "Save!" button will then become enabled (Figure 5.7). This button will open a File dialog to select the file to save. If the file already exists, it will prompt if you wish to replace it.

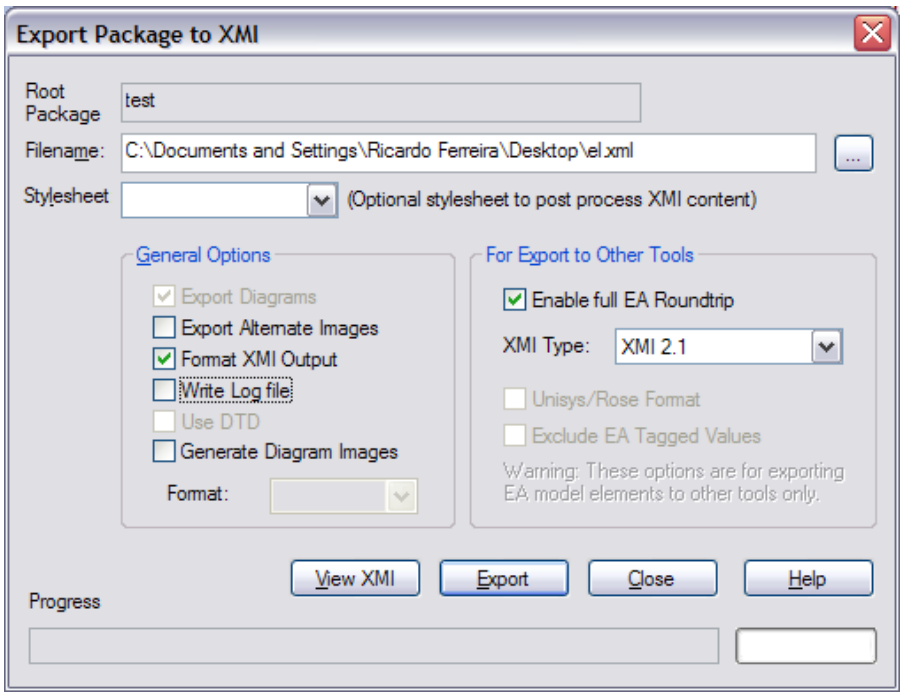

Figure 5.4: Export dialog in Enterprise Architect.

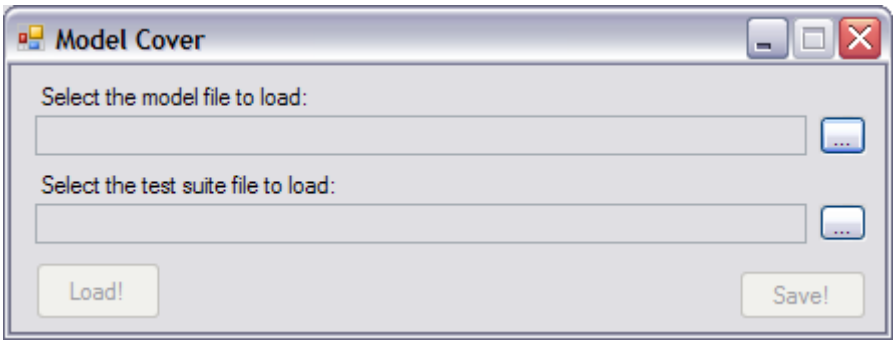

Figure 5.5: Initial window of the application developed.

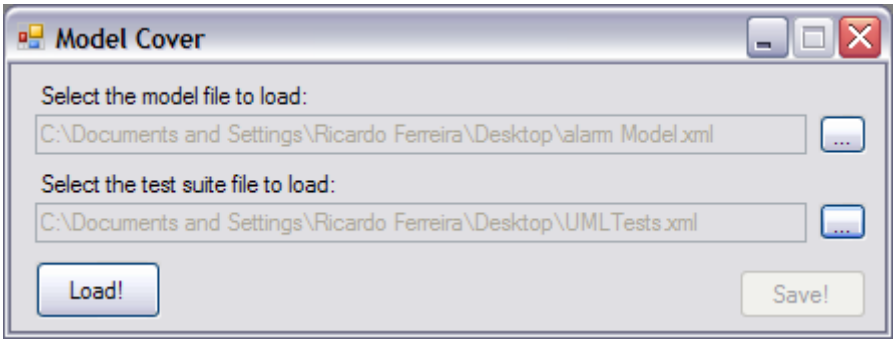

Figure 5.6: Load files in the application.

After the generation of the painted model, we need to import it back into the Enterprise Architect to see what has been covered. Before importing the file, we need to create a root node like the node “Model” shown in Figure 5.2. To import the file, we select the option “Project-

>Import/Export->Import Package from XMI" (Figure 5.3). The window in the Figure 5.8 will appear. Again, the option selected is the only needed to import successfully the model into Enterprise Architect.

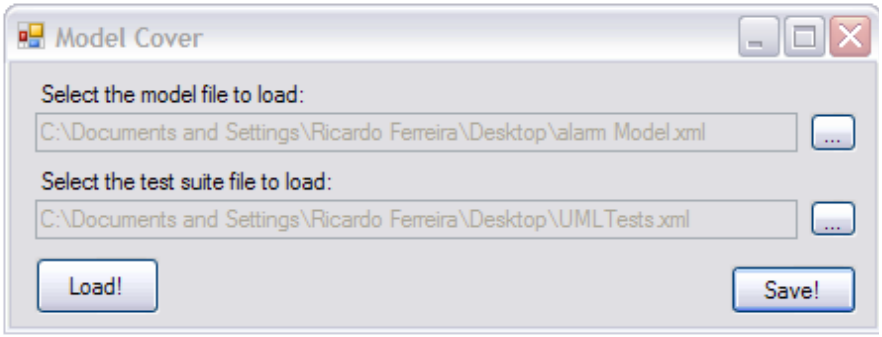

Figure 5.7: Save file in the application.

To identify which elements of the model were covered, they were painted with different colors. The transitions use a darker color to better identify them within the diagram. The following color schema was used to paint the model:

- Green: this color represents an element that was completely covered;
- Yellow: this color represents an element that was only partially covered. This can occur in several situations (see Figure 5.10):
  - All states and transitions of a composite state weren't covered;
  - Transitions starting in a composite state weren't tested from all states inside the composite state;
- Red: the element was not exercised by the test suite.

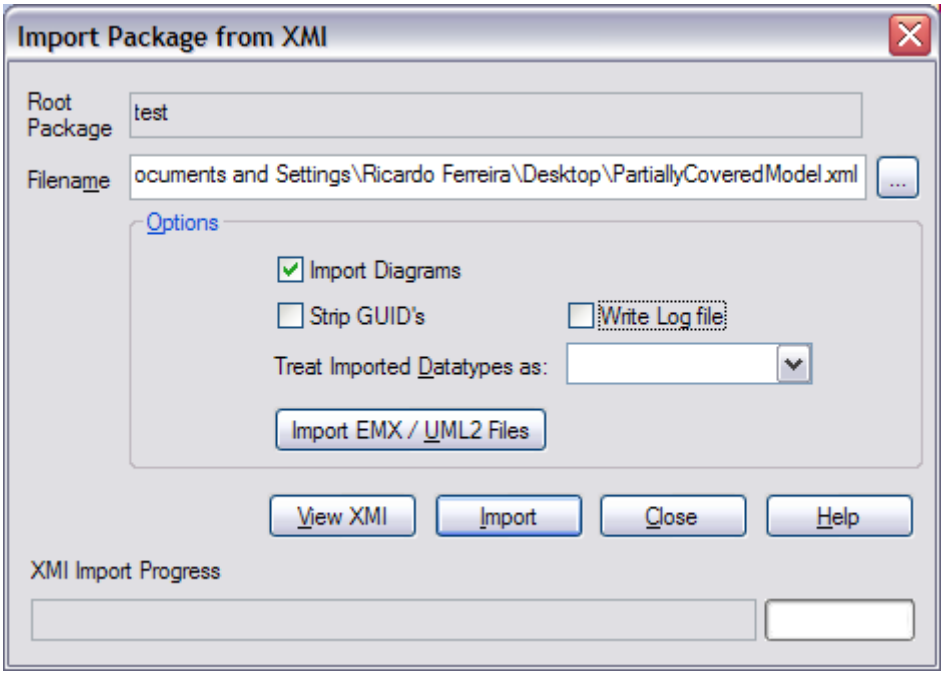

Figure 5.8: Import dialog in Enterprise Architect.

To test the program we used the example of the alarm defined in Section 4 . The initial model is shown in Figure 4.2. To test this model we have used the test suite generated by Spec Explorer and the manual test suite for the UML model. Both cover all the states and transitions of the model and the model looked like presented in Figure 5.9. To generate the incompletely covered model we have removed some tests and altered the sequence. The model got like

presented in Figure 5.10. Now, it is very simple to analyze the coverage of the model simply looking to the color of each state and transition. We can see in Figure 5.10 that the transition to deactivate the system was only partially exercised. This happen because it was not tried to deactivate the system when it was in “Siren ON” state which was not covered.

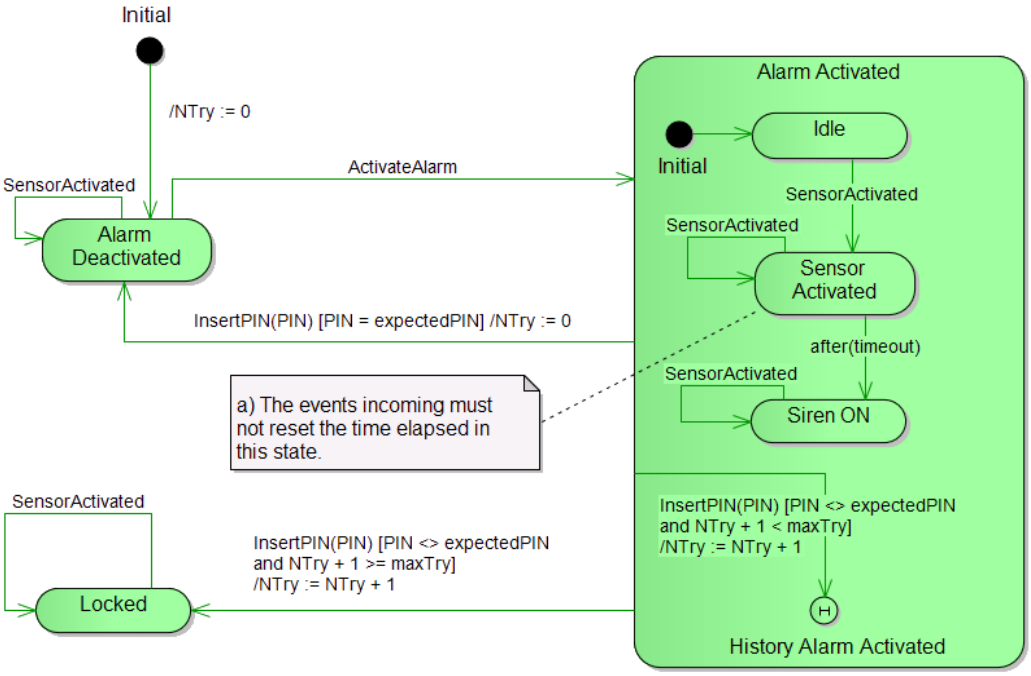

Figure 5.9: Fully covered UML model.

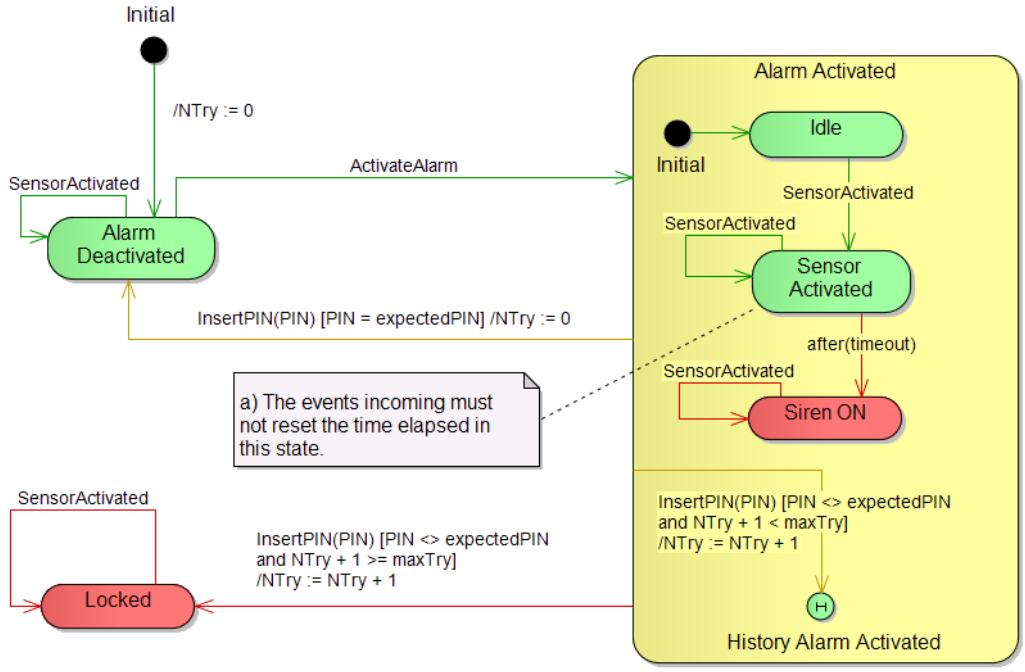

Figure 5.10: Partially covered UML model.

5.6 Internal structure

In this chapter the internal structure of the application will be detailed a little more. The Figure 5.11 shows the package diagram of the tool developed. The application consists in three different file classes (each one corresponds to a package of the Figure 5.11). Each file has a specific purpose and the main one (Model.cs) can access to methods of the other files. The main file is the one that supports in memory the structure of the model drawn. This file has several classes that are grouped as shown in Figure 5.12.

The only public class accessible in the application is the Model class. This includes functions that are used by the application to read, execute and paint the model and save it in the end. The other classes within this package allow the application to maintain a copy of the model in memory for coverage analysis. Each of these classes represents an element type in the model. The less obvious class is the Variable class. This class represent either class attributes or method/event parameters. The class diagram shown in Figure 5.12 (except for the Model class) is a simplification of the UML superstructure [USS07]. The Variable class is not also in UML superstructure. Instead they split this class in two different ones: one for the attributes of the class and other to the parameters of the methods. Here, these two classes were grouped into only one.

Other of the files is the TestSuite Figure 5.13. This file gives support for loading test suite files. The structure of the test suite loaded will then be passed to “CheckCoverage” method of the model class, so it can check witch states and transitions were covered by the test suite. The final file is a parser of expressions (Figure 5.14) that is used to interpret the expressions, actions and sometimes, the parameters of functions. This is provided by the “EvaluateExpression” function present in the Parser class. The “Operand” class is used when we want to add variables to the expression. The array with the variables is used to evaluate the expression and, when found a reference to a variable, the variable is replaced by its value instead of its name. All the operands implements an interface that provides all the functions available for each of the classes.

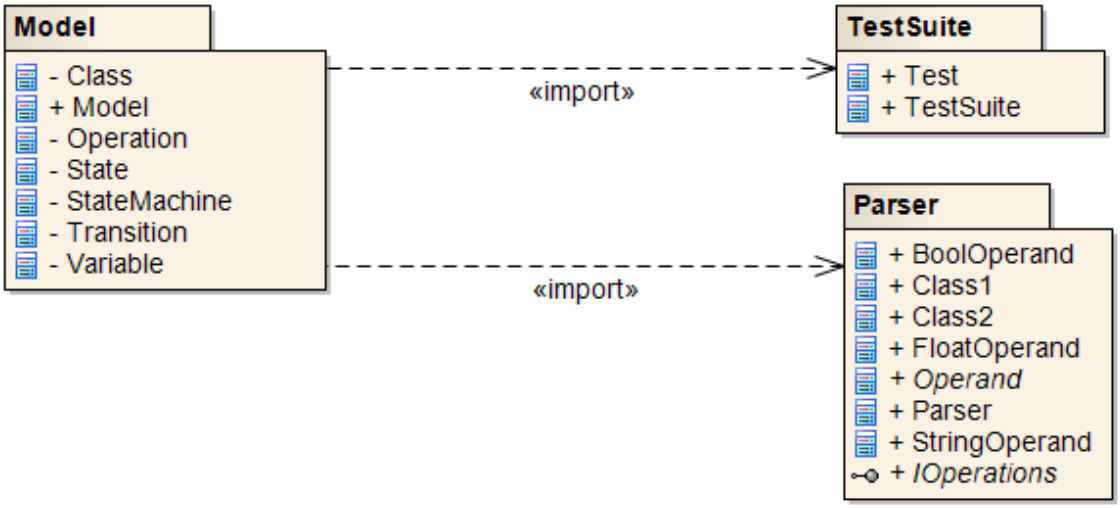

Figure 5.11: Package diagram of the application.

To better understand the whole functioning of the application, a sequence diagram was produced. This diagram intends to show the most important calls starting when the “Load!” button is pressed until the file is saved. The diagram can be consulted in Figure 5.15.

First, both files are loaded. When checking the coverage, for each transition crossed, the “EvaluateExpression(expression)” method is called. Details about how this function works are presented in the algorithms used in section 5.7. After the model has been checked, it is painted and saved to a file, ready to be imported in Enterprise Architect.

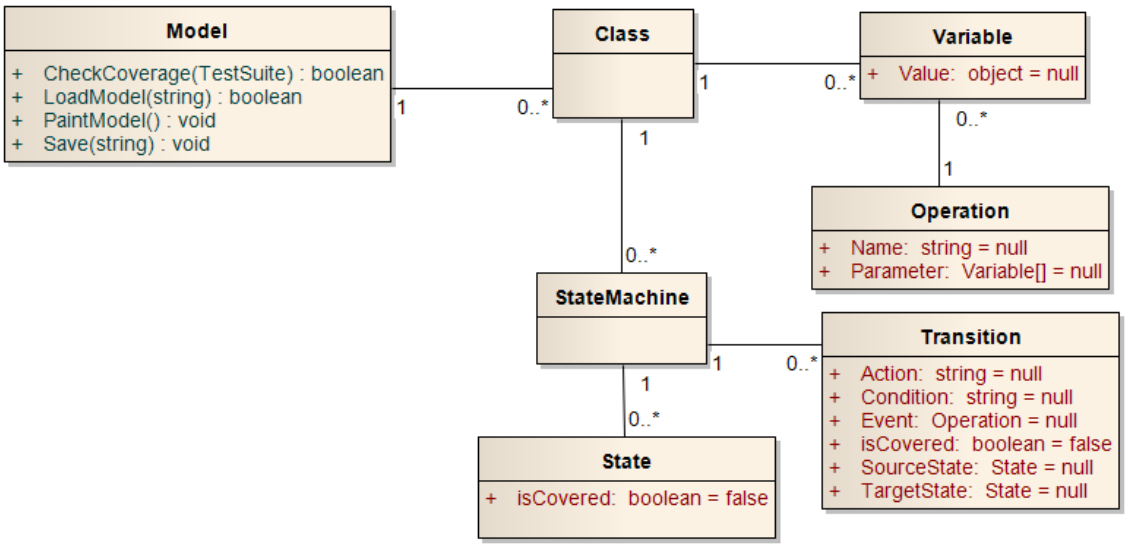

Figure 5.12: Model package class diagram.

Figure 5.13: Test Suite package class diagram.

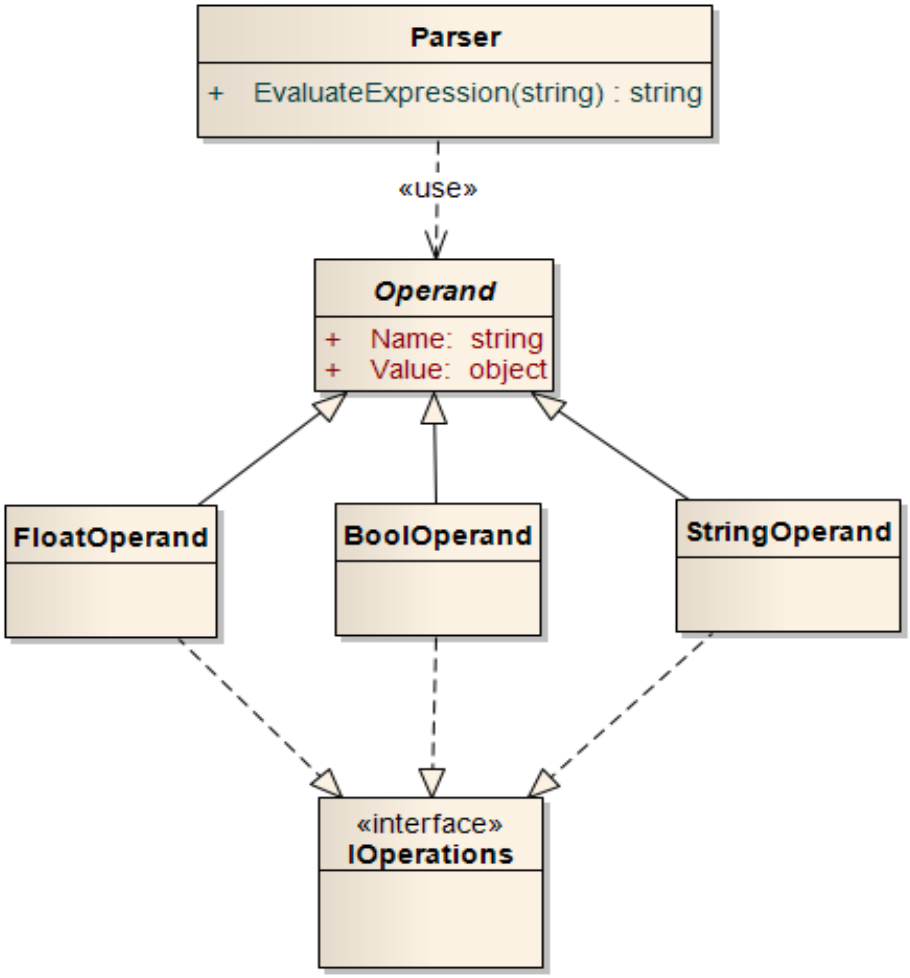

Figure 5.14: Parser package class diagram.

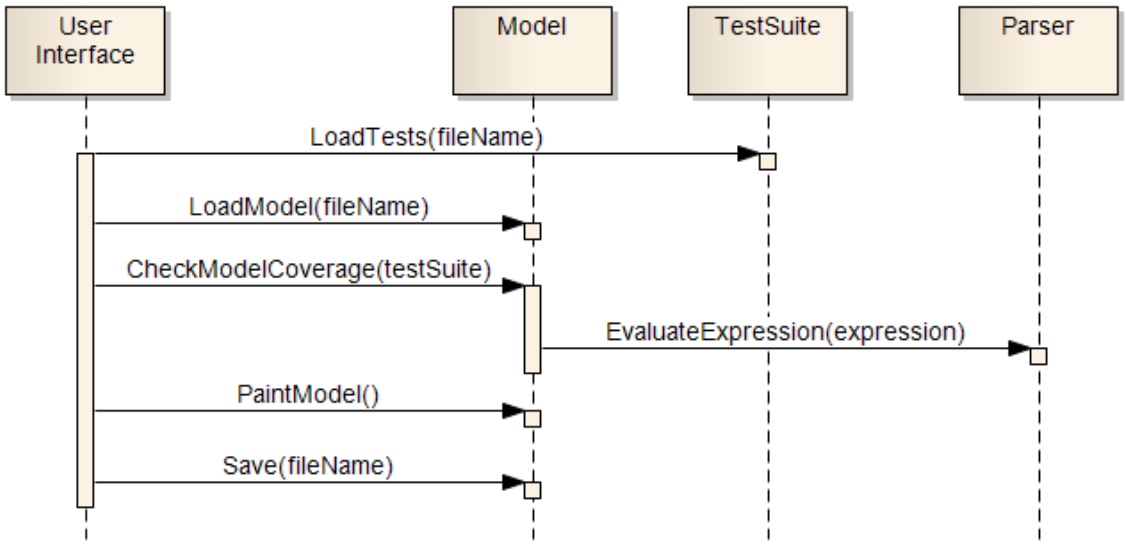

Figure 5.15: Sequence diagram of the application.

5.7 Algorithms and important technologies used

As for the algorithms and technologies used in this project, the most important will be described next.

5.7.1 LINQ

To load both files, as they are using XML to describe the model and the test suite, LINQ to XML [Mic09] was used. LINQ appears only in .Net 3.5 framework and is used to query SQL tables, XML files and collections that implement a specific interface (IEnumerable).

LINQ is very easy to use because it is based on queries that are performed to the file, looking for what we want. The query returns a list of results that can be accessed using some cyclic function like "foreach" instruction. LINQ also provides manipulation of the elements (this is required to change colors of the states and transitions of the model). The example shown in Table 5.1 returns in "qryMet" all methods of each class. Other approaches could require reading line by line the file and then using the lines that we want. LINQ automatically does this. Also, we can make queries to previously defined queries.

Table 5.1: Use of LINQ to XML.

```
XDocument modelDoc = XDocument.Load(fileName);

var qryClass = from clasM in modelDoc.Descendants()
               where (clasM.Attribute(XMI + "type") != null) &&
                     clasM.Attribute(XMI + "type").Value == "uml:Class")
               select clasM;

foreach (var XMLClass in qryClass)
{
    var qryMet = from attrib in XMLClass.Descendants()
                 where (attrib.Name.ToString() == "ownedOperation")
                 select attrib;
}
```

5.7.2 Model transformation

The first algorithm that needs to be explained is when we load the model. The initially loaded model needs some refinements before it can be executed by the test suite. The first update occurs in the transitions. If the transition has as a target state a state machine, the target state of the transition is set to the initial state of the state machine. If the source of the transition is a state machine, a transition for each state of the state machine is created and copied from the initial transition (except for the target state that will be the actual state of the state machine). Also, when a transition points to a history state, there are created as many transitions as the number of states in the state machine and the target state is set to each of the states of the state machine.

If the transition of the model is an "after" transition, the application behaves differently. First, for each class is added an internal attribute "elapsedTime" that cannot be created in the model definition because then we will have two attributes with the same name. This "elapsedTime" will increment every time an "Wait" instruction is called by the test case. If the instruction of the test case is not an "Wait" event, the "elapsedTime" will return to value 0.

Together with transitions that has no method, will be evaluated the transitions that have an "after" call. To evaluate this call is added to the guard condition a condition. If the guard condition does not exists, the condition "*time*<=elapsedTime" is added (time is the parameter of the after event and is initialized while loading the model). If a guard condition already exists, the previous condition is added in series together with the rest of the guard condition. So, if this condition fails, the whole guard condition fails. An example is shown in Table 5.2. Also, an action is added to the transition to update the "elapsedTime" in case of some after event occurs in the next state. Another change that needs to be done occurs in transition event. This event is set to null and the transition gets like those that do not have transition event being traversed automatically if the guard condition is verified.

Table 5.2: Treatment given to after event.

	Event	**Guard Condition**	**Action**
Transition	after(*time*)	size("abc")=3 or cos(PI)<>-1	null
Modified Transition	null	(size("abc")=3 or cos(PI)<>-1) and *time*<=elapsedTime	elapsedTime = elapsedTime - *time*

5.7.3 Model coverage analysis algorithm

Another algorithm is present when checking the coverage of the model. Basically speaking, the algorithm does the following: for each test case in the test suite and for each test instruction in the test case, the program first runs every transition that has no event, updating the current state (remember that in no-event transitions are included the after transitions that were altered). When there are no more to traverse, a list of possible transitions starting in the current state is filled and checked if some has the event correspondent to the current test instruction. In this case, the current state must be updated if the expression evaluation returns "True" or if there is no expression at all. In both cases, the action of the transition must be executed. After each test case, the loop through the transitions with no event is performed. The detailed algorithm can be consulted in Appendix D .

5.7.4 Expression representation and evaluation

Another important algorithm is the one used to parse expressions of the model. To interpret and evaluate each expression, it is used a Reverse Polish Notation (RPN) [MHP09, WikiRPN]. The code of the application is based on a portion of code found in the web [TCP04]. A sample of the notation using functions is shown in Table 5.3. To interpret the expression converted, it is used a stack to save the operands (the operands can be values and variables) until an operator is found. When an operator is found, if it is unary, only the first operand is popped from the stack, calculated its value and pushed again to the stack. In the other case, the operands are popped in inverse order (the first to pop is the second operand and the second value to be popped is the first operand), calculated its value and then pushed to the stack again. At the end of the converted expression, in the stack only remains the final value of the expression.

Table 5.3: Sample of expression, its RPN notation and return value.

Expression	EXP + -2 * cos(PI)^2 <> 0 and size("abc")=3
Expression in RPN	EXP 2 – PI cos 2 ^ * + 0 <> "abc" size 3 = and
Return value	True

5.8 Input file structure

The input files that the system accepts have a predefined structure as exported by Spec Explorer (the test suite file) and Enterprise Architect (the model). Here, an overview of its structure will be presented.

Starting with the test suite, the exported file by Spec Explorer follows a simple structure. The structure depends on what is exported. The example shown above requires that the states, transitions and test suite have been exported. If other parts are not exported, they will not be included in the exported file. Remember that the application developed requires that at least the test suite has been exported. The example was generated by exporting an FSM with only 2 states and 3 transitions. The tags that the file uses are very explicit. The “states” tag represents the beginning of the states declaration. Here will be all the states that the FSM contains as well as their attribute values. The “transitions” tag follows the same method. Here are included all the transitions present in the FSM, and for each of them, its source and target state and the invocation to go from the source to target state. Finally we have the “testsuite” tag. Here are the test cases. In this case, there are two test cases (“TESTSEGMENT0” and “TESTSEGMENT1”). Each test case includes several transitions defined previously.

Table 5.4: Test Suite sample file.

```
<?xml version="1.0"?>
<database>
  <states>
    <state id="S0">state: expectedPIN=12, maxTry=2, timeout=2,
status=ALARM_DEACTIVATED, NTry=0, remainingTime=0, hashcode=0</
state>
    <state id="S1">state: expectedPIN=12, maxTry=2, timeout=2,
status=IDLE, NTry=0, remainingTime=0, hashcode=1</state>
  </states>
  <transitions>
    <transition id="T0">
      <source>S0</source>
      <invocation id="I0">ActivateAlarm()</invocation>
      <target>S1</target>
    </transition>
    <transition id="T1">
      <source>S0</source>
      <invocation id="I1">SensorActivated()</invocation>
      <target>S0</target>
    </transition>
    <transition id="T2">
      <source>S1</source>
      <invocation id="I2">InsertPIN(12)</invocation>
      <target>S0</target>
    </transition>
  </transitions>
  <testSuites>
    <testSuite id="TESTSUITE0" name="Test Suite #0"
kind="AllLinks" expectedCost="1" cost="0">
      <segment id="TESTSEGMENT0"/>
      <segment id="TESTSEGMENT1"/>
    </testSuite>
    <testSegment id="TESTSEGMENT0">
```

```
      <transition id="T0">
        <source>S0</source>
        <invocation id="I0">ActivateAlarm()</invocation>
        <target>S1</target>
      </transition>
      <transition id="T2">
        <source>S1</source>
        <invocation id="I2">InsertPIN(12)</invocation>
        <target>S0</target>
      </transition>
    </testSegment>
    <testSegment id="TESTSEGMENT1">
      <transition id="T1">
        <source>S0</source>
        <invocation id="I1">SensorActivated()</invocation>
        <target>S0</target>
      </transition>
    </testSegment>
  </testSuites>
</database>
```

The model file is much more complex to be shown completely. So, the most relevant parts of it will be shown in Appendix E . In the example presented is a portion of the file related with the UML state machine used in the example of the section 4 .

The XML file has basically the same structure defined in Figure 5.2. First come the model package that includes classes defined and the associations between them. On each class, there can exist several attributes, events with associated parameters and state machines, being the first one the default and cannot be painted. Other state machines have a visual representation and can be painted. Some of the model transitions appear directly down in the class tree or inside the corresponding state machine. After the model definition, comes the tag “xmi:Extension”. Inside this tag, elements, connector, and diagram definitions appear. Inside the elements there are few data to get. From connector, we can get the method, guard condition and the actions defined in the transition. Also, within connector tag, we can change the color of the transition. Inside diagram tag, we can change the color of the state and the state machine.

5.9 Conclusions

In this chapter was presented an explanation of the functioning of the application developed. The main algorithms used to calculate the coverage of the model by the test suite was also referred.

The tool developed is still being developed, but it can by now read, update and save the updated model exercised by the test suite. It already has some limitations that are expected to vanish with the development progress. The tool also were designed to be as extensible as possible since there are a few elements that the tool needs to load. The parser, because it was reused from existing code, is not as extensible as it could be. If required, a new parsing solution can be created. This is possible since it is already a separate module used by the tool.

Although the tool is not fully developed, the requirements proposed in the beginning of this chapter were satisfied. The coverage criteria for model-based GUI testing is presented in section 3 . These coverage criteria can be automatically computed although sometimes it is hard to fulfill. The representation of the model coverage is done using the tool that is being created and was presented in this chapter.

6 Conclusions and future work

In this report several approaches for GUI testing were analyzed. From them, we have concluded that a good choice for the most of applications is the use of MBT although it can have applications where other testing approaches can be better than MBT. After that, modeling notations, coverage criteria and tools used in MBT were presented. The main modeling notations include finite state machines (FSM) that are good to test small systems with very restricted domains. On the other hand, UML state machines can be scaled for larger systems. Textual pre/post models can be used for all sorts of systems but are less familiar and visually appealing than UML models. Unfortunately, except for FSMs, the automatic generation of adequate test suites of manageable size from models is hard.

As for the tools that support MBT, they lack a model coverage analysis feature. Since it is impossible to achieve automatically 100% coverage for all coverage criteria and sorts of systems, it is important to be able to measure the degree of coverage achieved by the tests generated automatically and by additional tests added manually.

To overcome this problem, the tool presented in this report intends to cover this hole in the existing tools. Although the tool has already some functionalities implemented, they have to be refined for better performance and some others must be added. Also the tool has to be submitted to more tests to evaluate its quality.

The tool developed still needs some improvements in the following areas:

- Paint guard conditions according to its coverage;
- Validate input files (their correctness in format);
- Insert the option to reset the model at the end of each test case;
- Support for multiple actions in one transition;
- Invariant checking;
- Communication between classes through call methods.

Also, automatic test case generation from the UML model can be an important feature of this tool but it is not required for this project.

As for the dissertation's requirements, they were fulfilled. The coverage criteria for model-based GUI testing was presented in chapter 3 . These criteria depends on the modeling notation used to model the system. As for the tool developed, by now it has its basic requirements covered although some improvements can also be performed.

References

[AFG03] Andrews, A.; R. France; S. Ghosh; G. Craig. *Test adequacy criteria for UML design models*. In Software Testing, Verification and Reliability, pp. 95-127, 2003, **13**.

[Baz09] Bazman. *GUI Testing Checklist.* [cited 2009; Available from: http://members.tripod.com/~bazman/checklist.html].

[BBN04] Blackburn, M.; R. Busser; A. Nauman. *Why Model-Based Test Automation is Different and What You Should Know to Get Started*. In Software Productivity Consortium, 2004.

[BDG07] Baker, P.; et al. *Model Driven Testing - Using the UML Testing Profile*. Springer, First Ed., 2007.

[Bei95] Beizer, B.. *Black-Box Testing: Techniques for Functional Testing of Software and Systems*. Wiley, 1995.

[Bin00] Binder, R. V.. *Testing Object-Oriented Systems – Models, Patterns and Tools*. Addison-Wesley Professional, 1999.

[BrM07] Brooks, P. A.; A. M. Memon. *Automated GUI Testing Guided By Usage Profiles*. In *ASE '07: Proceedings of the 22nd IEEE international conference on Automated software engineering* , pp. 333-342, 2007.

[Bur03] Burnstein, I.. *Practical Software Testing*. Springer, First Ed., 2003.

[Cab76] McCabe, T. J.. *A Complexity Measure*. In *IEEE Transactions on Software Engineering*, pp. 308-320, 1976.

[Cun07] Cunningham, W.. *Fit: Framework for Integrated Test.* 16/09/2002 [cited 2009; Available from: http://fit.c2.com/].

[Ecl09] Eclipse, F.. *Eclipse Foundation*. 2009 [cited 2009; Available from: http://www.eclipse.org/].

[FiN09] FitNesse. *FitNesse*. [cited 2009; Available from: http://fitnesse.org/].

[GaR01] Gargantini, A.; E. Riccobene. *ASM-based Testing: Coverage Criteria and Automatic Test Sequence Generation.* In *Journal of Universal Computer Science*, pp. 1050-1067, 2001, **7**(11).

[GUI09] GUITAR. *GUITAR.* [cited 2009; Available from: http://guitar.sourceforge.net/index.shtml].

[Gur00] Gurevich, Y. *Sequential Abstract State Machines - Capture Sequential Algorithms*. In *ACM Transactions on Computational Logic (TOCL)*, pp. 77-111, 2000, **1**(1).

[Har87] Harel, D.. *Statecharts: A visual Formalism for complex systems*. In *Science of Computer Programming*, pp. 231-274, 1987, **8**(3).

[Hor99] Horrocks, I.. *Constructing the User Interface with Statecharts*. Addison-Wesley, 1999.

[HVC01] Hayhurst, K. J.; et al. *A Practical Tutorial on Modified Condition/Decision Coverage*. 2001.

[IEE98] IEEE. *Standard for Software Test Documentation*. IEEE Std 829-1998, 1998.

[Jav09] Sun Microsystems. *Java.* 2009 [cited 2009; Available from: http://java.sun.com/].

[JUn09] JUnit. *Junit.* [cited 2009; Available from: http://www.junit.org/].

[KFN99] Kaner, C.; J. Falk; H. Q. Nguyen. *Testing Computer Software*. Wiley, Second Ed., 1999.

[Mar06] Marathe, M.. *Basics of Mutation Testing.* 2006 [cited 2009; Available from: http://developer.spikesource.com/wiki/index.php/Basics_of_Mutation_Testing].

[MBT09] Gold Practices Website. *Model Based Testing.* 2009 [cited 2009; Available from: https://www.goldpractices.com/practices/mbt/].

[MeS03] Memon, A. M.; M. L. Soffa. *Regression testing of GUIs*. In *Foundations of Software Engineering - Proceedings of the 9th European software engineering conference held jointly with 11th ACM SIGSOFT international symposium on Foundations of software engineering*, pp. 118–127, 2003.

[MHP09] Museum of HP Calculators. *RPN.* [cited 2009; Available from: http://www.hpmuseum.org/rpn.htm].

[Mic09] Microsoft. *LINQ to XML.* msdn 2009 [cited 2009; Available from: http://msdn.microsoft.com/en-us/library/bb387098.aspx].

[MoP08] Moreira, R. M. L. M.; A. C. R. Paiva. *Visual Abstract Notation for Gui Modelling and Testing - VAN4GUIM*. In *3rd International Conference on Software and Data Technologies*, pp. 104-111, 2008.

[MSP01] Memon, A. M.; M. L. Soffa; M. E. Pollack. *Coverage Criteria for GUI Testing*. In *Foundations of Software Engineering - Proceedings of the 8th European software engineering conference held jointly with 9th ACM SIGSOFT international symposium on Foundations of software engineering*, pp. 256-267, 2001.

[MTC09] Esterel Technologies. *Model Test Coverage*. 2009 [cited 2009; Available from: http://www.esterel-technologies.com/products/scade-suite/model-test-coverage.html].

[Mye04] Myers, G. J.; et al. *The Art of Software Testing*. Wiley, Second Ed., 2004.

[Mye79] Myers, G. J.. *The Art of Software Testing*. John Wiley & Sons, First Ed., 1979.

[OCL09] OMG. *Object Constraint Language (OCL).* 08/01/2009 [cited 2009; Available from: http://www.omg.org/spec/OCL/].

[OMG02] OMG. *UML Testing Profile - Request For Proposal*. OMG Document: AD/01-07-08, 01/04/2002 [cited 2009; Available from: http://www.omg.org/docs/ad/01-07-08.pdf].

[Par08] Weißleder, S.. *ParTeG – Partition Test Generator*. 19/10/2008 [cited 2009; Available from: http://parteg.sourceforge.net/].

[PFT05] Paiva, A. C. R.; et al. *A Model-to-implementation Mapping Tool for Automated Model-based GUI Testing*. In *ICFEM05: International Conference on Formal Engineering Methods*, 2005.

[PMM97] Paternò F.; C. Mancini; S. Meniconi. *ConcurTaskTrees: A Diagrammatic Notation for Specifying Task Models*. In *INTERACT '97: Proceedings of the IFIP TC13 Interantional Conference on Human-Computer Interaction*, pp. 362-369, 1997.

[QTr09] ConformiQ. *Qtronic™*. 2009 [cited 2009; Available from: http://www.conformiq.com/qtronic.php].

[SeG05] Seifert, D.; C. Gaston. *Evaluating Coverage Based Testing*. Springer, p. 293, 2005.

[Sel09] SeleniumHQ. *Selenium*. 2009 [cited 2009; Available from: http://seleniumhq.org/].

[SpE09] Research, Microsoft. *Model-based Testing with SpecExplorer*. 2009 [cited 2009; Available from: http://research.microsoft.com/en-us/projects/SpecExplorer/].

[SpS09] Research, Microsoft. *Spec#*. 2009 [cited 2009; Available from: http://research.microsoft.com/en-us/projects/specsharp/].

[TCP04] The Code Project. *C# Expression Parser using RPN*. 2004 [cited 2009; Available from: http://www.codeproject.com/KB/cs/rpn_expressionparser.aspx].

[UML09] OMG. *UML® Resource Page*. 2009 [cited 2009; Available from: http://www.uml.org/].

[USS07] OMG. *OMG Unified Modeling Language (OMG UML), Superstructure, V2.1.2*. 09/2007 [cited 2009; Available from: http://www.omg.org/spec/UML/2.1.2/Superstructure/PDF/].

[UtL06] Utting, M.; B. Legeard. *Practical Model-Based Testing - A Tools Approach*. Morgan Kaufmann, First Ed., 2006.

[VDM09] VDM. *VDM information web site*, 2009 [cited 2009; Available from: http://www.vdmtools.jp/en/].

[Wag04] Wagner, Sebastian. *GUI Testing and Automated Test Generation*. 2004.

[Wei89] Weiss, S. N.. *Comparing test data adequacy criteria*. In *ACM SigSoft, Software Engineering Notes*, pp. 42-49, 1989, **14**(6).

[WikiMBT] Wikipedia. *Model-based testing*. 2009 [cited 2009; Available from: http://en.wikipedia.org/wiki/Model_based_testing].

[WikiRPN] Wikipedia. *Reverse Polish notation*, 2009 [cited 2009; Available from: http://en.wikipedia.org/wiki/Reverse_Polish_notation].

[Wit08] Wittaker, J.. *manual v. automated testing again*. 29/10/2008 [cited 2009; Available from: http://blogs.msdn.com/james_whittaker/archive/2008/10/29/manual-v-automated-testing-again.aspx].

[ZHM97] Zhu, H.; P. A. V. Hall; J. H. R. May. *Software Unit Test Coverage and Adequacy*. In *ACM Computing Surveys (CSUR)*, pp. 366-427, 1997, **29**(4).

Appendix A Common user interface errors

FUNCTIONALITY

Excessive functionality
Inflated impression of functionality
Inadequacy for the task at hand
Missing function
Wrong function
Functionality must be created by the user
Doesn't do what the user expects

COMMUNICATION

Missing information
No onscreen instructions
Assuming printed documentation is readily available
Undocumented features
States that appear impossible to exit
No cursor
Failure to acknowledge input
Failure to show activity during long delays
Failure to advise when a change will take effect
Failure to check for the same document being opened more than once
Wrong, misleading, or confusing information
Simple factual errors
Spelling errors
Inaccurate simplifications
Invalid metaphors
Confusing feature names
More than one name for the same feature
Information overload

When are data saved?
Poor external modularity

Help text and error messages
Inappropriate reading level
Verbosity
Inappropriate emotional tone
Factual errors
Context errors
Failure to identify the source of an error
Hex dumps are not error messages
Forbidding a resource without saying why
Reporting non-errors

Display bugs
Two cursors
Disappearing cursor
Cursor displayed in the wrong place
Cursor moves out of data entry area
Writing lo the wrong screen segment
Failure to clear part of the screen
Failure to highlight part of the screen
Failure to clear highlighting
Wrong or partial string displayed
Messages displayed for too long or not long enough

Display layout
Poor aesthetics in the screen layout
Menu layout errors
Dialog box layout errors
Obscured instructions
Misuse of flash
Misuse of color
Heavy reliance on color
Inconsistent with the style of the environment
Cannot get rid of onscreen information

COMMAND STRUCTURE AND ENTRY

Inconsistencies
“Optimizations”
Inconsistent syntax
Inconsistent command entry style
Inconsistent abbreviations
Inconsistent termination rule
Inconsistent command options
Similarly named commands
Inconsistent capitalization
Inconsistent menu position
Inconsistent function key usage
Inconsistent error handling rules
Inconsistent editing rules
Inconsistent data saving rules

Time-wasters
Garden paths
Choices that can’t be taken

Are you really, really sure?
Obscurely or idiosyncratically named commands

Menus

Excessively complex menu hierarchy
Inadequate menu navigation options
Too many paths to the same place
You can't get there from here
Related commands relegated to unrelated menus
Unrelated commands tossed under the same menu

Command lines

Forced distinction between uppercase and lowercase
Reversed parameters
Full command names not allowed
Abbreviations not allowed
Demands complex input on one line
No batch input
Can't edit commands

Inappropriate use of the keyboard

Failure to use cursor, edit, or function keys
Non-standard use of cursor and edit keys
Non-standard use of function keys
Failure to filter invalid keys
Failure to indicate keyboard state changes
Failure to scan for function or control keys

MISSING COMMANDS

State transitions

Can't do nothing and leave
Can't quit mid-program
Can't stop mid-command
Can't pause

Disaster prevention

No backup facility
No undo
No Are you sure?
No incremental saves

Error handling by the user

No user-specifiable filters
Awkward error correction
Can't include comments
Can't display relationships between variables

Miscellaneous nuisances

Inadequate privacy or security
Obsession with security
Can't hide menus
Doesn't support standard O/S features
Doesn't allow long names

PROGRAM RIGIDITY

User tailorability

Can't turn off the noise

Can't turn off case sensitivity
Can't tailor to hardware at hand
Can't change device initialization
Can't turn off automatic saves
Can't slow down (speed up) scrolling
Can't do what you did last time
Can't find out what you did last time
Failure to execute a customization command
Failure to save customization commands
Side-effects of feature changes
Infinite tailorability

Who's in control

Unnecessary imposition of a conceptual style
Novice-friendly, experienced-hostile
Artificial intelligence and automated stupidity
Superfluous or redundant information required
Unnecessary repetition of steps
Unnecessary limits

PERFORMANCE

Slow program
Slow echoing
How to reduce user throughput
Poor responsiveness
No type-ahead
No warning that an operation will take a long time
No progress reports
Problems with time-outs
Program pesters you
Do you really want help and graphics at 300 baud?

OUTPUT

Can't output certain data
Can't redirect output
Format incompatible with a follow-up process
Must output too little or too much
Can't control output layout
Absurd printed level of precision
Can’t control labeling of tables or figures
Can’t control scaling of graphs

Appendix B Manually defined test suite for UML model

```
void Test1
{
  int wrPIN = 10;

  SensorActivated
  assert(Alarm in
ALARM_DEACTIVATED)

  ActivateAlarm
  assert(Alarm in IDLE)

  SensorActivated
  assert(Alarm in
SENSOR_ACTIVATED)

  Wait(1)
  assert(Alarm in
SENSOR_ACTIVATED)

  SensorActivated
  assert(Alarm in
SENSOR_ACTIVATED)

  Wait(1)
  assert(Alarm in SIREN_ON)

  SensorActivated
  assert(Alarm in SIREN_ON)

  InsertPIN(wrPIN)
  assert(Alarm in SIREN_ON)
  InsertPIN(expectedPIN)
  assert(Alarm in
ALARM_DEACTIVATED)
}

void Test2
{
  ActivateAlarm
  assert(Alarm in IDLE)

  SensorActivated
  assert(Alarm in
SENSOR_ACTIVATED)

  InsertPIN(expectedPIN)
  assert(Alarm in
ALARM_DEACTIVATED)
}

void Test3
{
  ActivateAlarm
  assert(Alarm in IDLE)

  InsertPIN(expectedPIN)
  assert(Alarm in
ALARM_DEACTIVATED)
}
```

```
void Test4
{
  int wrPIN = 10;

  ActivateAlarm
  assert(Alarm in IDLE)

  SensorActivated
  assert(Alarm in
SENSOR_ACTIVATED)

  InsertPIN(wrPIN)
  assert(Alarm in
SENSOR_ACTIVATED)

  Wait(2)
  assert(Alarm in SIREN_ON)

  InsertPIN(wrPIN)
  assert(Alarm in LOCKED)

  SensorActivated
  assert(Alarm in LOCKED)
}

void Test5
{
  int wrPIN = 10;

  ActivateAlarm
  assert(Alarm in IDLE)

  InsertPIN(wrPIN)
  assert(Alarm in IDLE)

  SensorActivated
  assert(Alarm in
SENSOR_ACTIVATED)

  InsertPIN(wrPIN)
  assert(Alarm in LOCKED)
}

void Test6
{
  int wrPIN = 10;

  ActivateAlarm
  assert(Alarm in IDLE)

  InsertPIN(wrPIN)
  assert(Alarm in IDLE)

  InsertPIN(wrPIN)
  assert(Alarm in LOCKED)
}
```

Appendix C Pre/post model of the alarm specification

```
//enumerations and constant attributes definition
enum Status {ALARM_DEACTIVATED, IDLE, SENSOR_ACTIVATED,
SIREN_ON, LOCKED};
int expectedPIN = 12;
int maxTry = 2;
int timeout = 2;

//class attributes definition
Status status = Status.ALARM_DEACTIVATED;
int NTry = 0;
int remainingTime = 0;

//restrictions to the class
invariant NTry >=0 && NTry <= maxTry;
invariant status==Status.ALARM_DEACTIVATED ==> NTry == 0;
invariant remainingTime > 0 ==> status==Status.SENSOR_ACTIVATED;

//actions definition
//Activate alarm event
[Action]
void ActivateAlarm()
requires status == Status.ALARM_DEACTIVATED;
{
  status = Status.IDLE;
}

//Insert PIN code event
[Action]
void InsertPIN(int PIN)
requires status==Status.IDLE || status==Status.SENSOR_ACTIVATED
|| status==Status.SIREN_ON;
```

```
{
  if (PIN == expectedPIN)
  {
    remainingTime=0;
    NTry=0;
    status = Status.ALARM_DEACTIVATED;
  }
  else
  {
    NTry++;
    if(NTry >= maxTry)
    {
      remainingTime=0;
      status = Status.LOCKED;
    }
  }
}

//Wait action. Only because time can't be handled in Spec#
[Action]
void Wait (int waitTime)
requires waitTime > 0 && status==Status.SENSOR_ACTIVATED;
{
  if (remainingTime - waitTime <= 0)
  {
    remainingTime=0;
    status=Status.SIREN_ON;
  }
  else
  {
    remainingTime -= waitTime;
  }
}

//Sensor activation can always occur
[Action]
void SensorActivated()
{
  if (status == Status.IDLE)
  {
    status=Status.SENSOR_ACTIVATED;
    remainingTime = timeout;
  }
}

//Probe actions
[Action (Kind=ActionAttributeKind.Probe)]
int getStatus()
{ return status;}

[Action (Kind=ActionAttributeKind.Probe)]
int getNTry()
{return NTry;}
```

```
[Action (Kind=ActionAttributeKind.Probe)]
int getRemainingTime()
{return remainingTime;}

//Test #1
void Test1()
{
  ActivateAlarm ();
  assert(getStatus()==Status.IDLE);

  InsertPIN(10);
  assert(getStatus()==Status.IDLE);

  SensorActivated();
  assert(getStatus()==Status.SENSOR_ACTIVATED);

  Wait(1);
  assert(getStatus()==Status.SENSOR_ACTIVATED);
  assert(getRemainingTime()==1);

  SensorActivated();
  assert(getStatus()==Status.SENSOR_ACTIVATED);
  assert(getRemainingTime()==1);

  Wait(1);
  assert(getStatus()==Status.SIREN_ON);
  assert(getRemainingTime()==0);

  InsertPIN(12);
  assert(getStatus()==Status.ALARM_DEACTIVATED);

  WriteLine("Test1 Completed.");
}

//Test #2
void Test2()
{
  ActivateAlarm();
  assert(getStatus()==Status.IDLE);

  InsertPIN(10);
  assert(getStatus()==Status.IDLE);

  SensorActivated();
  assert(getStatus()==Status.SENSOR_ACTIVATED);
  assert(getRemainingTime()==timeout);

  InsertPIN(10);
  assert(getStatus()==Status.LOCKED);
  assert(getRemainingTime()==0);

  WriteLine("Test2 Completed.");
```

```
}

//Test #3
void Test3()
{
  ActivateAlarm();
  assert(getStatus()==Status.IDLE);

  ActivateAlarm();
}

//Test #4
void Test4()
{
  Wait(-1);
}

//Test #5
void Test5()
{
  InsertPIN(12);
}

//Main function will execute the tests.
//try catch used if the test case is supposed to violate the
//precondition of the action
void Main()
{
  Test1();
  Test2();

  try
  {Test3();}
  catch(Exception)
  {WriteLine("Test3 Completed.");}

  try
  {Test4();}
  catch(Exception)
  {WriteLine("Test4 Completed.");}

  try
  {Test5();}
  catch(Exception)
  {WriteLine("Test5 Completed.");}
}
```

Appendix D Algorithm to check model coverage

```
bool CheckCoverage(TestSuite testSuite)
{
  //for now, GetTestClass returns the first class of the model.
  //As future work it could detect the class where the tests
  //are being performed
  Class testClass = GetTestClass();

  foreach (Test testCase in testSuite)
  {
    //Finds the initial state of the first state machine
    State actualState = FindInitialState(testClass);

    //testCase contains a list of testSteps
    foreach (string testStep in testCase.TestSteps)
    {
      //loops through null events and returns the last state
      //this function also marks as covered the states and
      //transitions executed
      actualState = LoopAutomatedTransitions(actualState);

      //the corresponding method is returned with the value of
      //its parameters updated
      Operation testMethod =
               BindEventToMethodParameters(testClass, testStep);

      //gets the class attribute elapsedTime
      Variable elapsTime = GetClassAttribute(testClass,
                                             "elapsedTime");

      //if the test case is wait, increment the elapsed time
      if (testMethod.Name == "Wait")
```

```
{
  //in the wait event, the first parameter is the value
  //to increment
  elapsTime += testMethod.Parameter(0).Value;
  continue;
}

//gets the possible transitions that start in actualState
possibleTransitions=FindPossibleTransitions(actualState);

//searches the transition that corresponds to the
//testStep
foreach (Transition actTrans in possibleTransitions)
{
  if (actTrans.Event.Name == testMethod.Name)
  {
    //gets the acceptable vars to parse the expression
    //this includes class attributes and method
    //parameters
    ArrayList acceptableVars = GetPossibleVars(testClass,
                                               testMethod);

    //if the transition have condition defined
    if (actTrans.Condition != null)
    {
      Parser parse = new Parser(acceptableVars);

      //if the evaluation of the condition is true
      if (parse.EvaluateExpression(actTrans.Condition)==
                                                  "True")
      {
        //evaluate the action, set the actualState and
        //update elapsedTime with 0
        parse.EvaluateExpression(actTrans.Action);
        actualState = actTrans.TargetState;
        elapsTime = 0;
        //marks as covered the transition, source and
        //target state
        actTrans.isCovered = true;
        actTrans.TargetState.isCovered = true;
        actTrans.SourceState.isCovered = true;
        break;
      }
    }
    //else
    else
    {
      //evaluate the action, set the actualState and
      //update elapsedTime with 0
      parse.EvaluateExpression(actTrans.Action);
      actualState = actTrans.TargetState;
      elapsTime = 0;
      //marks as covered the transition, source and
```

```
                //target state
                actTrans.isCovered = true;
                actTrans.TargetState.isCovered = true;
                actTrans.SourceState.isCovered = true;
                break;
            }
        }
    }
    }
    //loops through null events and marks as covered the states
    //and transitions executed
    LoopAutomatedTransitions(actualState);

    return true;
  }
}
```

Appendix E Extract of the exported UML model file

```
<?xml version="1.0" encoding="windows-1252"?>
<xmi:XMI xmi:version="2.1"
xmlns:uml="http://schema.omg.org/spec/UML/2.1"
xmlns:xmi="http://schema.omg.org/spec/XMI/2.1">
  <xmi:Documentation exporter="Enterprise Architect"
exporterVersion="6.5"/>
  <uml:Model xmi:type="uml:Model" name="EA_Model"
visibility="public">
    <packagedElement xmi:type="uml:Package"
xmi:id="EAPK_5FEC29A8_CA24_4f5b_B767_B8803E7D12F3" name="test"
visibility="public">
      <packagedElement xmi:type="uml:Class"
xmi:id="EAID_2B851A4A_ED95_4fac_BC0D_D5F039403273" name="Alarm"
visibility="public">
        <ownedAttribute xmi:type="uml:Property"
xmi:id="EAID_D69671FD_050F_4ef3_8743_D7108AB6222C"
name="expectedPIN" ...>
                ...
        </ownedAttribute>
        <ownedOperation
xmi:id="EAID_F053DA5F_3C69_47c7_B22F_855FAD13628C"
name="ActivateAlarm" ...>
          <ownedParameter
xmi:id="EAID_RT000000_3C69_47c7_B22F_855FAD13628C" .../>
        </ownedOperation>
        <ownedBehavior xmi:type="uml:StateMachine"
xmi:id="EAID_SM000001_CA24_4f5b_B767_B8803E7D12F3"
name="AlarmStateMachine" ...>
          <region xmi:type="uml:Region"
xmi:id="EAID_SR000001_CA24_4f5b_B767_B8803E7D12F3"
name="EA_Region1" visibility="public">
```

```
            <subvertex xmi:type="uml:State"
xmi:id="EAID_045EF6A4_8EB7_4c9a_A7D9_3FC4DB9E5029" name="Alarm
Deactivated" ...>
                ...
            </subvertex>
                ...
          </region>
        </ownedBehavior>
        <ownedBehavior xmi:type="uml:StateMachine"
xmi:id="EAID_C3ADE581_1E2D_47d7_BD52_4A436F9676BC" name="Alarm
Activated" ...>
                ...
        </ownedBehavior>
        <transition xmi:type="uml:Transition"
xmi:id="EAID_2C44FCC8_4506_459e_8C27_47E3DCA17697" ...>
          <trigger xmi:type="uml:Trigger"
xmi:id="EAID_TR000000_4506_459e_8C27_47E3DCA17697"
name="InsertPIN(PIN)"/>
          <guard xmi:type="uml:Constraint"
xmi:id="EAID_CO000000_4506_459e_8C27_47E3DCA17697">
            <specification xmi:type="uml:OpaqueExpression"
xmi:id="EAID_OE000000_4506_459e_8C27_47E3DCA17697" body="PIN
&amp;lt;&amp;gt; expectedPIN and NTry + 1 &amp;lt; maxTry"/>
          </guard>
          <effect xmi:type="uml:OpaqueBehavior"
xmi:id="EAID_OB000000_4506_459e_8C27_47E3DCA17697" body="NTry :=
NTry + 1"/>
        </transition>
                ...
      </packagedElement>
    </packagedElement>
          ...
  </uml:Model>
  <xmi:Extension extender="Enterprise Architect"
extenderID="6.5">
    <elements>
      <element
xmi:idref="EAID_045EF6A4_8EB7_4c9a_A7D9_3FC4DB9E5029"
xmi:type="uml:State" name="Alarm Deactivated" scope="public">
          ...
      </element>
    </elements>
    <connectors>
      <connector
xmi:idref="EAID_CD28FB98_C76A_470c_87BB_CB7D6E44DD6D">
        <source
xmi:idref="EAID_C3ADE581_1E2D_47d7_BD52_4A436F9676BC">
                ...
        </source>
        <target
xmi:idref="EAID_045EF6A4_8EB7_4c9a_A7D9_3FC4DB9E5029">
                ...
        </target>
```

```
        <model ea_localid="18"/>
        <properties ea_type="StateFlow" direction="Source -&gt;
Destination"/>
        <documentation/>
        <appearance linemode="3" linecolor="-1" linewidth="1"
seqno="0" headStyle="0" lineStyle="0"/>
        <labels mt="InsertPIN(PIN) [PIN = expectedPIN] /NTry :=
0"/>
        <extendedProperties virtualInheritance="0"
privatedata1="InsertPIN(PIN)" privatedata2="PIN = expectedPIN"
privatedata3="NTry := 0"/>
        <style/>
        <xrefs .../>
        <tags/>
      </connector>
    </connectors>
    <diagrams>
      <diagram
xmi:id="EAID_D000D0D0_EE65_4be9_884B_1D15933B22E6">
        <model
package="EAPK_5FEC29A8_CA24_4f5b_B767_B8803E7D12F3" .../>
        <properties name="test" type="Statechart"/>
          ...
      </diagram>
        ...
    </diagrams>
  </xmi:Extension>
</xmi:XMI>
```