FACULTY OF ENGINEERING OF THE UNIVERSITY OF PORTO

STATE OF THE ART REPORT

Implementation of a Coverage Tool

Author:
Liliana VILELA

Supervisor:
Prof. Ana PAIVA

Dissertation Planning

February 2013

Universidade do Porto
FEUP Faculdade de Engenharia

This work is financed by the ERDF - European Regional Development Fund through the COMPETE Programme (operational programme for competitiveness) and by National Funds through the FCT - Fundação para a Ciência e a Tecnologia (Portuguese Foundation for Science and Technology) within project FCOMP-01-0124-FEDER-020554.

COMPETE
PROGRAMA OPERACIONAL FACTORES DE COMPETITIVIDADE

QUADRO DE REFERÊNCIA ESTRATÉGICO NACIONAL
PORTUGAL2007.2013

UNIÃO EUROPEIA
Fundo Europeu de Desenvolvimento Regional

FCT
Fundação para a Ciência e a Tecnologia
MINISTÉRIO DA EDUCAÇÃO E CIÊNCIA

Faculty of Engineering of the University of Porto

Dissertation Planning

Implementation of a Coverage Tool

by Liliana VILELA

Abstract

Testing and quality assurance is an area that is assuming increasing importance as software occupies more critical roles. Given that the complexity of the systems themselves keeps growing, manual testing becomes increasingly more difficult. Fortunately, there are now tools that are able to automate a significant part of the testing process.

Although such automated tools allow to rapidly create a significant number of test cases, the quantity of tests does not directly correlate to quality of testing. To define that quality, and the extent to which we wish to test the system, coverage criteria (or adequacy criteria) must be previously defined and analysed.

This paper concerns the implementation of such a coverage analysis tool to a model-based testing application, PARADIGM-ME. With this coverage tool, we intend not only to provide coverage support to PARADIGM-ME, but also to explore the options in regards to coverage analysis for MBT models in particular.

Keywords: model-based testing, mbt, coverage criteria, adequacy, testing

Contents

List of Figures

Chapter 1

Introduction

As software complexity keeps growing, it becomes more difficult to assure its quality and, in some cases, even its safety. Adding this to the crescent role of software in our everyday lives, it becomes easy to see the increasing importance of software testing.

Today more than ever though, there's available an ever growing array of software testing tools that automate many of the aspects of the quality assurance process. Between tools that perform code analysis, model analysis, and tools that work solely on the base of inputs or action repetition, the generation of a bigger quantity of test cases if now easier than ever.

Still, it is widely known that the number of tests executed does not provide us with assurances of system quality by itself. Rather, more than the number of tests generated, it's the quality of those tests that matters. For one to conclude that the system has been tested thoroughly, not only it must pass the defined test cases, but assure that test suites themselves meet certain criteria. Thus, coverage criteria is the metric that allows the tester to known to what extent a system is exercised by the test suites. Or, according to Weyuker's definition:

> "*The purpose of testing is to uncover errors, (...) and the purpose of an adequacy criterion is to access how well the testing process has been performed.*" [1]

Although coverage criteria and test cases generation have overall been well-documented in literature, El-Far and Whittaker noted an interesting lack of in-depth studies concerning the coverage of models in particular [2]. Indeed, these are not the only authors to point the coverage metrics as one of the drawbacks on Model-Based Testing. Eslamimehr has also noted this as one of the drawbacks that prevents MBT to be more widely applied, claiming that common test metrics are insufficient - as stated previously, simply counting the test cases is not enough, specially when these tests are the generated kind [3]. We expect that a 'good' coverage criteria, if successfully applied to MBT, can be a much better indicator of testing progress.

However, there are yet other problems commonly faced by testing tools that propagate into analysing model coverage effectively. One example of this is the well known problem of state space explosion in non-trivial models, which propagates into all that goes into testing, including achieving adequate coverage criteria. This lead us to either work with small programs, or with weak coverage [4], with both options being sub-par.

One of the aims of this work is precisely to try to diminish this lack of information on the application of coverage metrics to MBT in particular. The proposal consists in adding coverage analysis to a Model-Based test tool, PARADIGM-ME, designed to test graphical user interfaces by use of common interface patterns [5]. This tool will be further elaborated upon Chapter 3. While developing this coverage tool to integrate into PARADIGM-ME, we expect to be able to add value to the existing body of knowledge, not only by exploring the state of the art, but specially by being able to compare the application of several different approaches. Hopefully, by the end of this project one should be able to distinguish what works and what doesn't on the application of coverage criteria to system models, and thereby be able to provide information on what are the most efficient approaches in each case.

This document is structured into four main chapters. In this first section, Introduction, we start by introducing the theme to be developed during the course of the dissertation, starting by defining the context and issue at hand.

Next, on State of the Art, we proceed to give an overview of related work done in the area. This will expand mainly to focus on coverage methodologies, being that it's application to Model-Based Testing (MBT) tools in general is also addressed where it was deemed relevant.

On the third chapter of this report, Suggested Approach, we explain the solution proposed to the problem at hand. This chapter itself is also divided into several sections. First, one intends to give a better understanding on the context in which the coverage tool to develop will be implemented. As such, a brief overview of the tool PARADIGM-ME is provided. Following this exposition, one then explains the approaches the author is considering to take, although these might be subject to change as the work progresses. To conclude this chapter, we include a detailing of the work plan itself.

Finally, to wrap up this document on the State of the Art, on the chapter pertaining to Chapter 4 we present some thoughts about the research done thus far.

Chapter 2

State of the Art

In order to find the best approach to this problem, some research on already existing methodologies and concepts was needed. As such, and in order to make this document as clear as possible, we start by defining some main concepts related to testing in general. In regards to the state of the art itself, an overview is given on several coverage criteria available, focusing specially on graph coverage. This overview is followed by a run-down of some approaches used to analyse coverage in models specifically. To conclude this chapter, we present the results of some investigation on what actual MBT tools are using to measure coverage.

2.1 Key Concepts

In the course of this report, we'll be using several terms related to key concepts in software testing. In order to make the reading as clear as possible and avoid ambiguity, we define beforehand some of these concepts.

Test Case According to ISTQB, a test case can be considered as "*a set of input values, execution preconditions, expected results and execution postconditions, developed for a particular objective or test condition, such as to exercise a particular program path or to verify compliance with a specific requirement*" [6].

In other words, a test case answers the question: "What are we going to test?". They define conditions that must be validated to ensure that the system is working as expected [7].

Test Suite A test suite can be considered simply as a collection of several test cases for a component or system under test, which are grouped for test execution purposes [6, 7].

Test Path A path in itself consists in a sequence of events of a component or system, from an entry to an exit point [6]. Hence, a test path represents the execution of a test case [8] as it runs through the several statements of a system under test.

Coverage and Adequacy Criteria To put it simply, an adequacy criteria is a set of rules used to determine if sufficient testing has been performed [1].

On the other hand, and strictly speaking, coverage criteria refers to the percentage of the system that is reached (or 'covered') by the defined test suite.

The difference between these two terms is not always clear however. The adequacy criteria used is commonly expressed as 'the degree to which the system is exercised by the tests', which, in practice, consists in the very definition of system coverage. Hence, while the terms 'adequacy criteria' and 'coverage criteria' are not exactly the same in meaning, they are often used interchangeably since, in practice, code coverage is used as a measurement of test adequacy [9].

Graph Coverage In regards to graphs, coverage criteria are applied the same way, being met by visiting particular nodes, edges, or by having the tests transverse particular paths [8].

2.2 Coverage Criteria

Given that this work concerns the implementation of a coverage analysis tool, it is important to define some of the main coverage criteria used. These typically can be divided into several groups. Here, we'll focus on the graph coverage criteria, since these are the ones most relevant to the topic at hand.

The reason for this focus lies partly on the fact we consider state machine graphs to be the most direct way to represent the system as a model. In addition, in regards to MBT in particular, the use of graph coverage criteria is recommended by Eslamimehr to be the best approach [3].

Inside the graph coverage criteria, these can be further classified according to whether they're based on the program's control flow (also known as structural criteria), or based on its data flow. Following, we'll give a brief run-down of each of them. All definitions here provided are based on the ones given by Ammann [8], unless otherwise stated.

2.2.1 Structural Graph Coverage Criteria

Node Coverage and Edge Coverage Some of the simplest types of graph based coverage criteria include Node Coverage and Edge Coverage (also known as Statement Coverage and

Decision Coverage respectively). As the very name states, the objective with each one of these criteria is to visit all reachable nodes, and to cross all reachable edges.

A variant based on the last technique is Edge-Pair Coverage. While Edge Coverage criteria's definition states that one must cover all paths with length *up to 1*, Edge-Pair Coverage is geared towards 'pairs of edges'. Putting it another way, instead of covering edges one by one, Edge-Pair covers them in paths of length *up to two*, or 'in pairs'.

Prime Path Coverage A simple path consists in a path which has no internal loops. In other words, while it's possible for the path itself to begin and end at the same node, there can be no other repeated nodes in it. Since a path can be comprised by more than one subpath, and in order to avoid redundancy when listing a graph's simple paths, only the longest simple paths are usually considered. These maximal length simple paths are known as prime paths.

The Prime Path Coverage consists then in covering all the prime paths in the system graph. The disadvantage to this approach however, is that an infeasible prime path can actually be made of several, feasible, subpaths. The solution proposed to this problem is to analyse all these feasible subpaths in place of their subsuming prime path.

Simple Round Trip Coverage and Complete Round Trip Coverage Simple Round Trip Coverage and Complete Round Trip Coverage are two special cases of Prime Path Coverage. With these, analysis of loops is done by using round trips. More specifically, round trips paths are prime paths that start and end at the same node, and whose length is bigger than zero.

While Simple Round Trip Coverage requires only a minimum of one round trip path to be analysed for each node, Complete Round Trip Coverage requires all round trip paths to be analysed.

Still, Amman recommends one to simply adopt the Prime Path Coverage in opposition to these two variants, stating the first one tends to be more practical in most situations [8].

Complete Path Coverage Complete path coverage, as the name itself indicates, concerns covering all the existing paths in a graph. Although this approach would at first seem to be the most indicated, this coverage criteria is actually infeasible when a graph has loops. The reason for this is that loops cause the number of possible paths in a graph to be infinite.

Specified Path Coverage A variant from the criteria above, instead of requiring all graph paths, the paths to be explored are defined beforehand. Since only a given, defined subset is actually analysed, this eliminates the previous problem of having an infinite number of paths. The definition of which the subset to test is usually left to the tester.

2.2.2 Data Flow Graph Coverage Criteria

Data Flow Criteria focus on the flows of the data values specifically. In other words, it is analysed by tracking where the variables are defined (*def*), and where they are used (*use*). Relying on the fact that data is carried from definitions to uses, we obtain the *def-use* associations (also called *definition-use*, or *du-pairs*) [8]. Hence, the main objective of Data Flow Criteria in general is to exercise these *def-use* associations.

Among the variable uses, we can further distinguish between *c-uses* and *p-uses*. We call a use *c-use* when the value of the variable is used in a computation. By *p-use*, we refer to the use of the variable in a condition or proposition [10]. As an example, the computation of $x = y+1$ would constitute a *c-use* of the variable *y*, while the condition *if (y > 5)* would be a *p-use* of the same variable.

In what concerns the concepts related to the paths themselves, it's also relevant to define the notion of *def-clear* and *def-use* paths. A *def-clear* path for a variable consists in any path in which the value of said variable is not changed. Or, putting it another way, there is no definition (*def*) of that variable between the beginning and ending of the path.

A *def-use* path for a variable, consists in a simple, *def-clear* path, where the variable is specifically defined at the starting node and used at the ending node.

Arising from these definitions, we obtain then several coverage criteria to address data flow, which we'll now cover.

All-Defs Coverage The All-Defs Coverage pertains to covering *def-use paths*, in a way that assures that all definitions reach at least one use.

All-Uses Coverage Similar to the All-Defs Coverage defined above, the All-Uses coverage aims to covering *def-use paths*, making sure that from each definition, we reach all of their possible uses.

This criteria in particular has several 'weaker' variants, mostly based on the use of *c-uses* and *p-uses*. Thus, we can further distinguish criteria such as All-C-Uses and All-P-Uses, as well as All-C-Uses/Some-P-Uses and All-P-Uses/Some-C-Uses.

The first two criteria, All-C-Uses and All-P-Uses, function very much as the All-Uses one, but limit the coverage to a specific use type.

All-C-Uses/Some-P-Uses is a further deviation from the All-C-Uses. If no *c-uses* are found for a path, then one must test a succeeding *p-use* instead. All-P-Uses/Some-C-Uses makes use of this same approach, but inverts the role of *c-uses* and *p-uses* [10].

All-Def-Use-Paths Coverage This criteria aims to make sure that each definition reaches every possible use, by every possible *def-use path*. This is the main distinction from the criteria above: if there are several alternative *def-use paths* connecting a *def* to a given *use*, All-Uses coverage can use just one of those paths; All-Def-Use-Paths coverage on the other hand, will cover them all.

2.2.3 Criteria Overview

In order to better analyse the 'strength' of the coverage criteria presented in this chapter, we present a simple diagram as an attempt to relate all criteria seen thus far. Figure 2.1 illustrates this subsumption relation, in which a criteria is subsumed by another, if the compliance with the second one implies the compliance with the first.

The strongest coverage criteria analysed is then Complete Path Coverage. The arrows indicate a relationship of subsumption; in this case, we can verify that Complete Path Coverage guarantees the compliance with all the other criteria, either directly or indirectly. Although not represented in this diagram due to legibility reasons, the 'weaker' variants of the All-Uses Coverage are also effectively subsumed by it (All-P-Uses, All-C-Uses, All-P-Uses/Some-C-Uses, and All-C-Uses/Some-P-Uses).

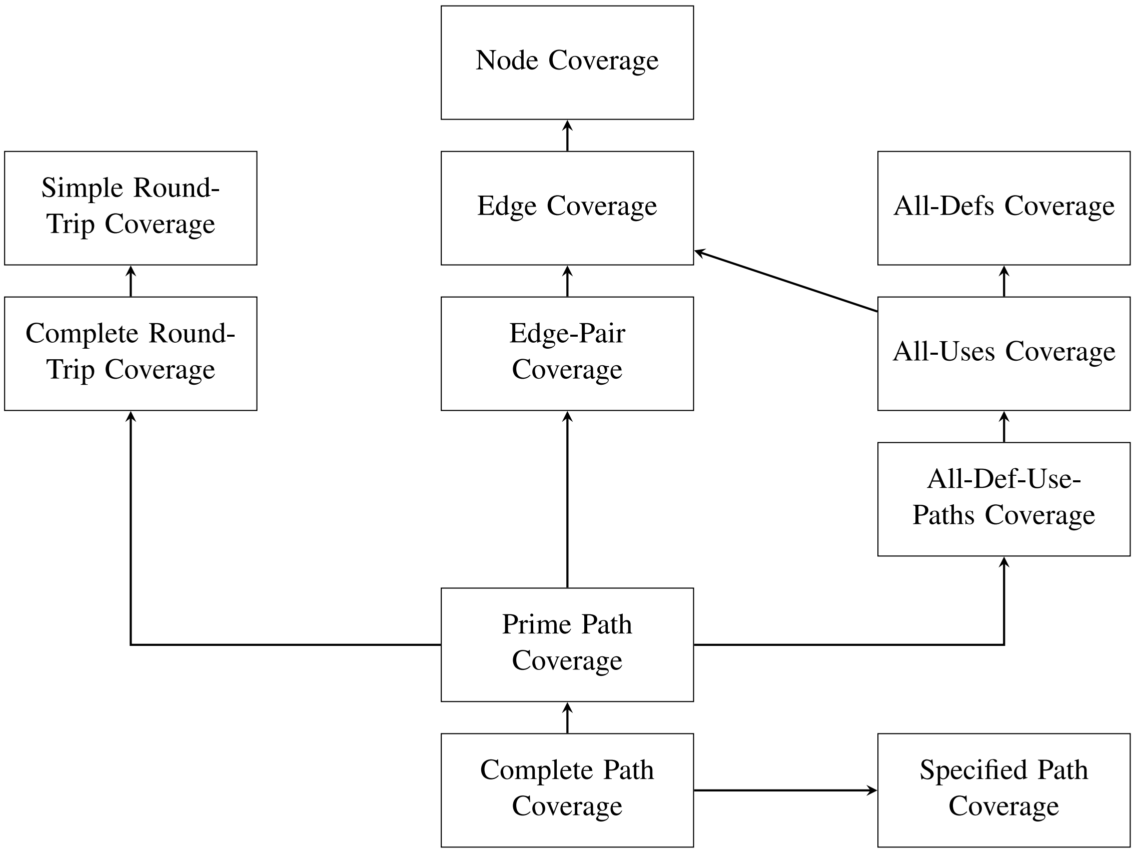

FIGURE 2.1: Diagram representing the subsumption relationship between several coverage criteria (adapted from the one provided by Amman in [8]).

This subsumption relationship holds true even when the graphs have loops, such as what can be seen in Figure 2.2. As an example, and basing ourselves in the definitions provided, we obtain the following test paths:

- Edge Coverage:
  - Test requirements: (1, 1), (1, 2)
  - Test paths: (1, 1, 2)
- Edge-Pair Coverage:
  - Test requirements: (1, 1, 2)
  - Test paths: (1, 1, 2)
- Prime Path Coverage:
  - Test requirements (prime paths): (1, 1), (1, 2)
  - Test paths: (1, 1, 2)

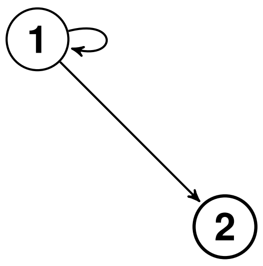

FIGURE 2.2: Example of a graph with a loop path of length one.

As a reminder, Edge Coverage covers paths with length up to one, while Edge-Pair Coverage includes those with length up to two (paths of length zero, while allowed, were not considered here). This inclusion of paths bellow one and two in length guarantees the subsumption of weaker criteria such as node coverage. As for Prime Path Coverage, while prime paths themselves don't allow internal loops, they can also be of any length. Hence, this fact allows Prime-Path Coverage to also cover paths such as (1, 1, 2), effectively subsuming Edge and Edge-Pair coverage criteria.

2.3 Code Coverage

In relation to the proposed work, it's also relevant to understand how code coverage tools function. Code coverage itself can be summed up as the application of coverage analysis to the source code of the system under test (SUT). Typically, this analysis is done using an approach dubbed 'code instrumentation', which we'll cover in this section.

2.3.1 Code Instrumentation

The main goal of code instrumentation is to enable the monitoring of a program during its execution. This is done by inserting additional code into the program that doesn't change the behaviour of the application itself, but collects information on its execution [8]. Chittimalli and Sha described code instrumentation as "*an activity which inserts a set of probes in a program to enable information gathering that would be useful for [monitoring the behaviour of a program for analysis, debugging or testing purposes]*" [11].

This additional code, usually referred to as probes, is therefore inserted at specific points in the program. By running at the same time as the remaining code, it's able to monitor the execution of the application. For instance, in a structural approach, probes register that a certain point in the program has been reached during execution. On the other hand, when using a data flow approach, probes mark that the definition or use of a certain variable has been covered. Hence, based on this, one can even state that a test plan describes where probes should be placed, and what they should do when reached [12].

In what concerns the instrumentation itself, one can distinguish among three types of approach, according to the level of abstraction where the probes are inserted [11]:

- Source code instrumentation: as the name itself indicates, it's performed at the source code level, and is therefore dependent on the programming language being used.
- Byte code instrumentation: aimed mainly at Java and .NET programming languages, relies on the fact that bytecode architectures should be subject to less change than the language itself. Due to this fact, is generally considered a better alternative to source code instrumentation.
- Binary instrumentation: performed at execution time, it's usually very dependent on the architecture and compiler used.

Though instrumentation allows an effective way to monitor the behaviour of a program, it's not without its downsides. The main problem with instrumentation lies on the fact that introducing probes means adding extra code, which in turn signifies an increase in execution time. This additional execution time, or overhead, can be classified as either offline or run-time overhead, according to whether one is referring to the overhead introduced by adding the probes, or to the one related to the execution of said probes [13].

This additional overhead problem is particular significant when wanting to add coverage to large industrial systems [14], and even more so when it comes to embedded systems, in which the execution time is a critical factor [15]. As a result, several approaches were developed as an attempt to mitigate this problem.

Dynamic Instrumentation

The instrumentation techniques mentioned thus far are usually classified as 'static instrumentation'. With static instrumentation, the probes are inserted into the code before execution, and remain there afterwards as part of the executable file. As mentioned, this increases execution time, since the original source code now contains also the inserted instrumentation code.

Another way to perform instrumentation relies on a dynamic approach. This method mitigates the overhead problem, since probes are inserted and removed from the program on-the-fly during its execution.

One example of such approach is Tikir and Hollingsworth's, that inserts instrumentation code dynamically, and removes it when it ceases to produce additional coverage information. In addition, they also introduce the concept of *incremental function instrumentation*. Instead of integrating all probes at once, the relevant instrumentation code is only introduced when a function is called by the first time during execution. Coupling this with the code removal, the result is that instrumentation code is kept on the paths that are actually executed by the application, unlike what usually happened with traditional static instrumentation techniques [16].

A similar technique is used with Jazz tool, also using a dynamic approach. In fact, Misurda et al. salient the main difference to the previous approach, which lies in the method used to remove the instrumentation code. While the previous technique relies on a periodic garbage collection to remove probes, Jazz removes instrumentation as soon as possible. They report this method as providing better results in reducing overhead [12].

2.3.2 Graph Instrumentation

In regards to graphs in particular, they too serve a purpose in instrumentation. Indeed, Najumudheen states that many instrumentation approaches use either tables or graphs to represent the targeted program [17].

As for how this is achieved, Ammann presents several examples on how instrumentation for coverage analysis can be accomplished through graphs [8]. For instance, when instrumenting for node coverage in particular, each node is given a unique identifier. These identifiers are then used to create an array, which will associate each node with a counter. The probes, placed at each node location, increment the array value of the respective node by one whenever they're executed. By the end of the execution, we'll have an array which contains the number of times each node was traversed, and hence, our coverage information. A similar process is used for edge coverage, being that here identifiers are attributed to edges instead of nodes.

Coverage analysis based on data-flow, although a bit more complex, is performed in much the same way. The main difference in this case lies in using two arrays instead of one: an array to keep track of the variable definitions, and another to track their uses [8].

2.3.3 Code Coverage Tools

In relation to code coverage tools available, we found a wide array of them. Most are restricted to a single programming language and, as such, the list of the tools found would be too extensive to describe here in its entirety. As such, we'll give a brief overview of some of the tools regarding what we consider to be widely used languages, such as Java and .NET languages. Since the application we'll be extending, PARADIGM-ME, is geared specially to web interface testing, we'll focus also on the tools that can be more readily applied to web testing, namely, PHP and Javascript coverage tools. For a broader study on the subject, Yang et al. performed a more comprehensive comparison of some of the code coverage tools available [13].

In regards to the PHP code coverage tools available, we find software such as PHPCoverage[1], which adopts a 'per-line' approach, or, in other words, it records how many times each line of the target script is executed. In fact, the only coverage criteria currently supported by the tool seems to be precisely line coverage [18]. Another tool, Xdebug[2], not only does it provides code coverage, but it also seems to be the basis for the PHP coverage functionality of many tools found, such as PHPUnit[3] [19] and the library PHP_CodeCoverage[4]. As with PHPCoverage, probes are placed as to track the execution of each line.

As for Javascript, JSCover[5] functions much as the other tools seen thus far. As with the previous ones, line coverage is based on the actual, physical lines of code rather than logical statements. Here, instrumentation itself can currently only be performed on Javascript files (.js extension), as opposed to inline Javascript embedded in HTML pages [20]. Saga[6] on the other hand, provides coverage information for both standalone and inline scripts. Other examples of tools providing code coverage for this language include Istanbul[7], and the Blanket library[8].

In what concerns the Java programming language, there is a wide choice in tools. For example, Emma[9] can instrument individual .class files or .jars, hence, it does not require access to the application's source code. Additionally, it also supports both offline and on-the-fly instrumentation options. A second tool, Cobertura[10], functions much in a similar way, instrumenting Java bytecode after compilation. There exist plenty of other Java oriented code coverage tools, some of which include Clover[11], CodeCover[12] and Coverlipse[13], with these last two being available as Eclipse plugins.

[1]http://sourceforge.net/projects/phpcoverage/
[2]http://xdebug.org/
[3]https://github.com/sebastianbergmann/phpunit/
[4]https://github.com/sebastianbergmann/php-code-coverage
[5]http://tntim96.github.com/JSCover/
[6]http://timurstrekalov.github.com/saga/
[7]https://github.com/gotwarlost/istanbul
[8]http://migrii.github.com/blanket/
[9]http://emma.sourceforge.net/
[10]http://cobertura.sourceforge.net/
[11]http://www.atlassian.com/software/clover/overview/groovy-code-coverage
[12]http://codecover.org/index.html
[13]http://coverlipse.sourceforge.net/

For the .NET platform, we have available standalone tools such as NCover[14], OpenCover[15], and dotCover[16], being that code coverage is also supported natively by IDEs such as Visual Studio[17] [21]. Although we were not able to find much information in regards to instrumentation tactics used the tools mentioned, line coverage seems to be the most commonly applied here as well. While some tools do provide alternative coverage options, we found these to be the minority, and even then we consider the coverage options provided to be somewhat scarce.

Even if for the most part code coverage tools are language dependent, there are some exceptions to this rule. For instance, in addition to both Java and C++ support, PurifyPlus[18] supports Basic and .NET as well, while Semantic Designs (SD)[19] extends its functionality to languages as diverse as C#, PHP, PL/SQL and COBOL. Both these tools achieve this through different means however. In PurifyPlus, this seems to be accomplished with dynamic instrumentation at byte or object level, which enables the analysis of third party libraries without having its source code, and without the need for recompilation [22, 23]. On the other hand, Semantic Designs coverage tool inserts language specific probes at source code level before its compilation/execution. This leads us to believe it functions much like other languages dependent tools, with the difference of packaging several language parsers in a single tool.

With regards to the coverage criteria supported, simpler structural coverage criteria seem to be the ones most implemented, with line coverage being by far the most widely used. In fact, for many of the tools described above, it was the sole coverage criteria being provided. Other more complex criteria are not as widely represented in commercial code coverage tools, possibly due to the increased complexity of development these imply [13].

2.4 Coverage Analysis in Models

As we've seen in the previous sections, defining coverage criteria is one thing, analysing if such criteria are met is another altogether. The same holds true for analysing model coverage in particular. This is a problem that has spawned several different approaches.

At times, instead of analysing the model directly, the test coverage of a model is found by generating code from said model, making then a coverage analysis on that generated code [24].

[14]http://www.ncover.com
[15]https://github.com/sawilde/opencover
[16]http://www.jetbrains.com/dotcover
[17]http://www.microsoft.com/visualstudio/eng
[18]http://www-01.ibm.com/software/awdtools/purifyplus/
[19]http://www.semdesigns.com/Products/TestCoverage/

Other times, model coverage itself isn't even considered, since coverage can be used as merely a means to guide automatic test generation, and not an end by itself [25, 26].

Generation of code from the model is not always possible though. In some situations, a tool has its own model definition, which makes the use of third-party tools for generating code a very difficult task, if not even an impossible one. Although the tool responsible for designing the model could provide its translation into system code, we believe that is not the most cost and time-effective solution. The alternative is what most MBT tools seem to implement, that is, relying on the model itself to retrieve all the necessary data. Typically, models used can range from design models to UML ones. The approach used is still very much the same however, relying mostly on state space exploration. In order to achieve that, we believe one possible solution may lie in the application of graph transversal algorithms.

Among this category, we find widely applied algorithms, such as Breadth-first Search and Depth-first search, as well as the several variants built upon these. In addition, there's also the family of what is broadly designed as Best-first Search, of which A* is probably the most widely used algorithm. Unlike the former two though, Best-first search and its variants rely on heuristics to find the most promising path, giving it priority. Still, when analysis of the state machine graph as a whole is required, it's not clear if the use of such heuristics would bring us any advantage.

Unfortunately, such approach can easily be considered lacking in efficiency, specially when trying to combine stronger coverage criteria with more complex system models.

Model Transformations

As a way to ease this type coverage analysis on models, Weißleder suggested an alternative method, by exploring the concept of *simulated satisfaction* on UML state machines [27, 28]. His approach is based on model transformations: instead of trying to achieve coverage for a complex model, one can try to achieve coverage for a different, but equivalent model. In other words, the stronger coverage criteria can be simulated with the use of a weaker one. He dubs these as *semantic preserving state machine transformations*. The basic premise is to obtain transformations for which the satisfaction of a coverage criteria on the transformed model is at least equal to the satisfaction of a stronger coverage criteria on the original model. In this line of work, the same concept was also applied to restricting coverage criteria as a way to show both their strength and dependency on the model [29]. Coverage Simulator is a tool prototype that intends to demonstrate this approach [30].

Although Weißleder created this approach mainly as a way to circumvent restrictions in common model-based tools, namely the limited coverage criteria choices these often provided [28], we believe it can be adapted to other uses. More precisely, use model transformations as a facilitator to analyse the coverage of any model in general. One issue here might be defining new transformation rules, since in the original work there was no need for the transformations to be bidirectional. A second aspect to consider refers to the fact that the impact of any transformation is ultimately linked with the coverage criteria being used, and this impact isn't always a positive one.

Furthermore, we believe to also be important to consider the trade-off in what concerns the performance of a tool applying this technique. As mentioned by the author, not all transformations result in an improvement in efficiency over applying the stronger criteria directly. Since the research at hand was focused on the overall impact of model transformations and not directly on the efficiency of each one, such considerations must be taken into account when adapting these approaches to any other process. In the same note, there's also the need to consider the scalability of such approach in regards to the complexity growth in the original models.

2.5 Coverage in MBT Tools

In regards to the practices in the industry concerning MBT tools, we found it somewhat hard to differentiate between the techniques used to evaluate coverage and the ones used to generate the test cases automatically. In fact, since most tools focus on generating the test cases, the coverage criteria itself serves as no more than a guideline to generate them. Such is the case at Microsoft with the *Abstract State Machine Language Tool (AsmL/T)*, in which the definition of *queries* guides the test cases. One example of such queries include the *Shortest Path*, where the tester specifies the beginning and ending point in a path to be tested (hence, what the generated tests ought to cover) [25].

A similar case can be found with Spec Explorer[20]. Here, what is defined as coverage criteria is effectively used as test selection criteria, being Transition Coverage the most used one [26, 31]. A third MBT tool we looked into, Simulink Design Verifier[21], uses formal methods to identify design errors in the models. In this one, model coverage is once again used to generated additional tests. In regards to coverage criteria though, it supports not only commonly used criteria, such as Branch Coverage, but also reports coverage on metrics like test objectives or constraints [32]. Other tools that seem to use the coverage criteria to generate test cases include Conformiq Qtronic[22] and Smartesting CertifyIt[23] [33].

[20] http://visualstudiogallery.msdn.microsoft.com/271d0904-f178-4ce9-956b-d9bfa4902745
[21] http://www.mathworks.com/products/sldesignverifier/
[22] http://www.conformiq.com/
[23] http://www.smartesting.com/index.php/cms/en/product/certify-it

As for the specific criteria mentioned in this document, Statement Coverage and Transition Coverage appear to be the most commonly used. Still, while there are other tools implementing the other types of criteria, we have found that most don't seem to offer a vast array of choice on this matter. Qtronic is the only one we've found to supply All-Paths coverage for acyclic graphs for instance [34, 35], along with some other additional coverage criteria not mentioned in this report [33].

In what concerns the analysis of the model itself, the overall idea throughout relies on exploring the state-machine derived from the model. The exact process by which this is achieved however, varies. With Spec Explorer, the model itself is explored by using transformations, in which a new graph has at most one outgoing transition for each state [26, 31]. Other tools make use of *coverage checkpoints* in the model, also known as *coverage items*, of which Conformiq Qtronic is also an example [36]. With SCADE Suite[24], the coverage analysis is done by verifying the activation of each element in the model as the system is executed. It's important to note though, that the main objective of coverage analysis in SCADE is to check compliance to higher-level requirements, rather than the correctness of the overall system itself. This technique was inclusively further explored to provide support for test cases for Java generic classes [37].

Another approach in regards to using model exploration to generate test cases was developed by Andrade et al.. By using algebraic specifications, expressed in ConGu, one is able to generate an Alloy model that obeys said specifications. This model is then coupled with a associated algebraic mapping, and Java unit test cases are generated [38]. This technique was inclusively further explored to provide support in generating test cases for Java generic classes in particular [37].

While most commercial MBT tools found at this point do specify the basic type of functionalities provided, including coverage, they do not make it clear in what way exactly is the criteria being analysed. Such we have found to be the case with IBM's Rational Rhapsody[25] for instance [39]. We attribute this limited information in most commercial tools to not wanting to disclose proprietary information. A more comprehensive comparison of MBT tools in general can be found in Shafique and Labiche's report on the subject [33].

[24]http://www.esterel-technologies.com/products/scade-suite/
[25]http://www.ibm.com/developerworks/rational/products/rhapsody/

Chapter 3

Suggested Approach

3.1 PARADIGM-ME

In this section, we'll provide a brief overview of the project in which the proposed tool will be integrated, PARADIGM-ME (extended from the application previously known as PetTool (*Pattern Based GUI Testing Tool*)) [5].

PARADIGM-ME aims to provide a more effective way to test graphical user interfaces (GUI). The approach relies on the fact that most GUIs end up having having similar elements, and therefore, a test based on these elements (or *patterns*) can be reused across several applications. For each pattern, there is an expected behaviour from which test cases can be defined.

Since PARADIGM-ME uses a Model-Based testing approach, a model of the system under test (SUT) can be built using the previously mentioned GUI patterns as building blocks. The tester can then design and configure the test cases on top of said model, and map it to the target application when time comes to execute the tests [5]. Although PARADIGM-ME already allows to evaluate the 'correctness' of an application, such is not enough if the tester doesn't know where to stop testing, or how much the system is actually covered by the defined tests. This is where the proposed Test Coverage Tool comes in: to give test coverage indicators to PARADIGM-ME, aiding therefore the tester to better evaluate the quality of the test suits.

3.2 Solution Proposal

The proposed work consists then in the implementation of a coverage analysis tool into PARADIGM-ME, a model based testing program. Given that the models and respective test configurations are already handled by PARADIGM-ME itself, we propose to display coverage information to the tester by simply colouring the already provided models. So, each element of the model would be coloured with red, yellow or green, following common colour notation in testing. This way, an element that would be considered as achieving complete coverage would be coloured in green, while one for which tests had not yet been defined would be marked as red. All other intermediate cases of partial coverage would present elements coloured in yellow. A rough prototype of what is intended can be seen in the comparison between Figure 3.1 and Figure 3.2. We believe this strategy to be able to provide a quick and intuitive comprehension of the current coverage of the system under test.

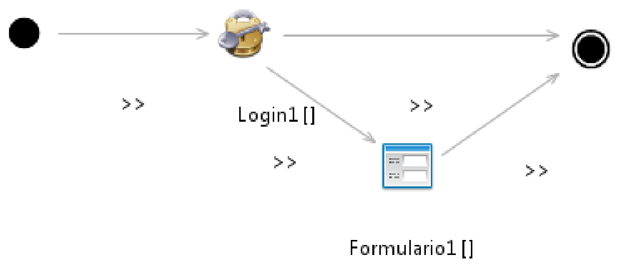

FIGURE 3.1: Example of a simple model created with PARADIGM-ME.

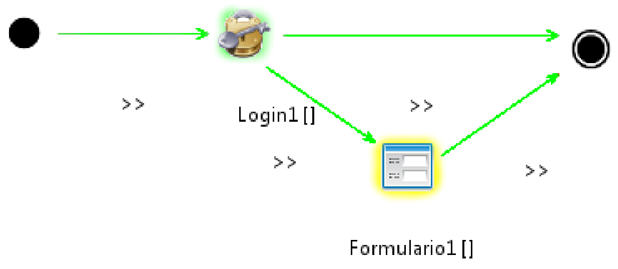

FIGURE 3.2: Example of a possible way to convey coverage information through the use of colours.

Nevertheless, one should also consider that not all people see colours the same way; in fact, some people have trouble in distinguishing colours all together. Due to this, we consider it would be a good approach to also include some kind of notation beyond colour, whether this feature be optional or not. The way to provide extended information or coverage options is still yet to be considered.

In addition to analysing the model and tests defined on top of PARADIGM-ME, it's also intended that this coverage tool is able to provide coverage information for test scripts. To achieve this, the tester should provide the tool with a system model, and an independent test script to be compared. Among other things, this could allow the tester to evaluate several different test suites against the same model.

So far, we’ve only covered the coverage approach regarding static testing, which doesn’t imply the execution of the system itself. However, there are special cases which can only be detected by running the system. One example of this is when ’false positives’ occur when testing for error conditions. Taking the example of a test case which aims to insert invalid information in an input field, such test case would likely be expecting some form of error handling on the part of the system (displaying an error message could be one way). It could happen that the system, instead of handling such error, would not allow the invalid information to be introduced in the first place. Such situation would maintain the system in a correct state, but the test case in question would fail. Typically, the tester would have to check the failed tests one by one, and discard the false positives manually. If a tool could perform this check automatically, not only the ’quality’ presented for the system would be much more accurate in the first place (less false positives), but it could hopefully serve as a time saver for the tester himself.

Possible Approach

At this stage, more research and experimentation is still needed to elaborate a full path to developing the proposed tool. Although aspects such as the way to implement the dynamic coverage referred in the last section are still undefined, some possible paths of exploration are beginning to appear. One of these paths passes roughly by the following steps:

1. Mapping of models into state-machine graphs
2. Define model transformation rules to apply to graphs
3. Analyse coverage on the graph
    - (Optionally) Define best algorithm to apply based on model structure
    - Implement graph search algorithms
    - Store data regarding coverage for each element
4. Map results from the graph to the original model

First and foremost, one needs to learn the inner workings of the environment in which the coverage tool will be integrated. The understanding on how to interact with PARADIGM-ME is clearly crucial to the development of its coverage support, specially in what concerns storage and treatment of data related to the models and defined tests.

Since we’re only basing the analysis on models, and not on code, our approach to evaluate coverage must depend heavily on a structural approach. Making use of some of the ideas and recommendations mentioned in Chapter 2, the implementation of the coverage tool will most likely pass by the use of graphs and graph coverage criteria. Thus, the first logical step will be to find a way to transform the PARADIGM-ME models into finite state machine (FSM) graphs. Though we intend to provide options for several coverage criteria, the first coverage criteria to implement is not yet clear at this moment. We consider the hypothesis to start with Edge Coverage, since it's relatively simple to implement and still provides an adequate coverage in most graphs. Furthermore, if we're able to successfully apply model transformations to graphs, it will enable us to apply weaker coverage criteria to simulate the effects of stronger one. In this way, model transformations could serve as a facilitator for the next step, the application of the coverage criteria analysis in itself. We believe the main challenge here consists in defining what are the most adequate transformation rules to implement.

In regards to the graph analysis, one possible approach could consist in using a graph search algorithm to run through each node and, for each, retrieve information on its associated test cases. At this stage, each graph node would be associated with a single model element. By analysing the elements of the model one by one, one can calculate on-the-fly the coverage data for each, and associate that information with the respective node. The downside is that this is very much a greedy procedure, hence it will suffer in terms of efficiency. As a result, for the tool to be able to analyse the coverage of more complex models, an adapted solution will probably need to be produced.

An idea we'd like to implement would be to make use of not one, but several heuristics. Ideally, if we could define a way to apply the most adequate algorithm based either on model type or complexity, the performance and efficiency of the coverage tool could be greatly enhanced. This step will be again considered once a more in-depth study of the several algorithms is carried through. Namely, one needs to take several measures of efficiency in regards to several models to achieve this, task that could prove to be too time-consuming to be able to implement in its entirety.

The final step, albeit of equal importance, consists in mapping the results obtained in the graph back into the original model. Here, the coverage results are finally displayed to the tester. This is another phase where the understanding of the PARADIGM-ME will be crucial, for one needs to convey the coverage information on top of the constructed model.

Although this first approach will likely change as the development unfolds, we believe this to be a good basis from which to start building the proposed coverage tool. The route defined will be subjected to corrections as experimentations accumulate, and other alternatives are discovered during the course of the project.

3.3 Work Plan

The work plan for the proposed project can be divided into the following major tasks:

- State of the Art Research
- Study of the PARADIGM-ME tool
- Implementation of the Test Coverage Tool
- Period dedicated to testing and validating the coverage tool
- Writing of a scientific report
- Writing of the dissertation document

Whilst the dates for each work section are defined by this point, they could be subject to change along the course of the project. Since the periods in which each task is set to be worked upon are not independent, we believe the overall work structure in relation to the time available can be better understood by use of a Gantt diagram (Figure 3.3).

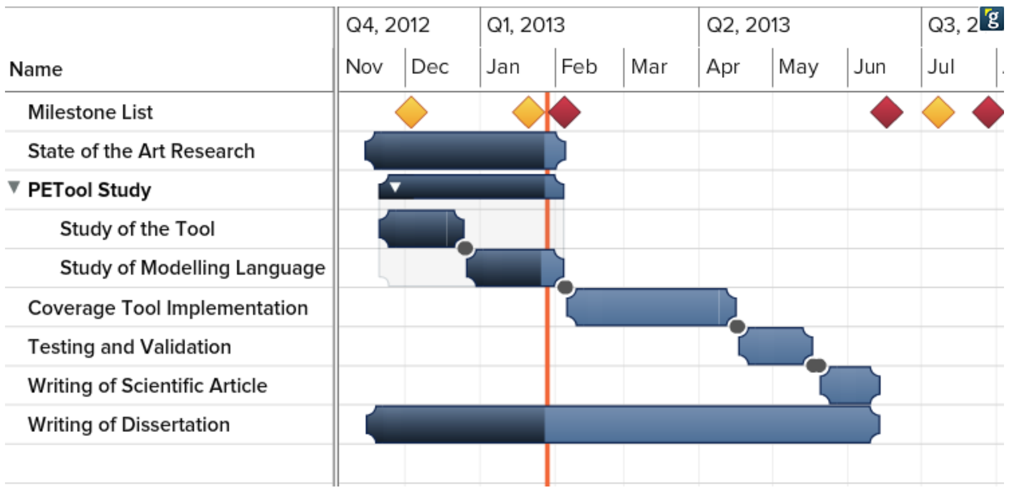

FIGURE 3.3: Gantt chart representing the proposed work plan. Milestones in red refer to documents' delivery, orange ones represent work presentations.

As of the moment of this writing, the research on both the state of the art regarding coverage tools and the PARADIGM-ME itself will carry on being done until the beginning of February. The study of the PARADIGM-ME will comprise both the tool itself and the modelling language being used. Following these steps, the effective work on the coverage tool is then ready to be started, comprising a phase which should last until roughly the middle of April. By then, the work developed thus far will undergo a testing and validation phase, making adjustments as needed. We expect to have the tool concluded halfway through the month of May at most. This deadline aims make time to write a scientific paper, as well as to wrap up the dissertation document, which will be progressing in parallel with the previous phases. All the work here detailed is expected to be done by 15 of June of the current year.

Chapter 4

Conclusions

It doesn't cease to be interesting that, although the types of coverage criteria available are plenty, Model-Based Testing tools offer a relatively small range of them. In the same note, we were able to observe that the vast majority of MBT tools available do not support coverage analysis with basis on pre-existing tests. Instead, the bulk of them is aimed at generating test cases on-the-fly. If on one hand, it's more efficient this way, on the other we believe it somewhat limits the ability of such tools to integrate with other types of test generation software.

In this state of the art report, we were concerned mainly with identifying the main coverage criteria used with MBT in particular, hence the predominant focus on graph coverage. As for the approaches used to analyse the application of those criteria, the several methodologies end up having much in common. As expected, one aspect common to all tools assessed thus far is their heavy reliance on state space exploration. Indeed, this is pretty much a required step in what comes to analysing software models based on control flow. Other model types exist that are based on more high-level, abstract concepts, but usually, those tend to aim for requirement validation more than for overall system validation.

Yet, some unexpected approaches were found during the course of this report. For instance, the discovery that state machine transformations are not only a valid approach, but actually used in real, industrial software testing products. This fact strengthened our conviction to try to see how far we can take this approach to solve the current problem. In particular, we consider it an ingenious way to circumvent the usual restrictions of tools in what concerns their supported criteria range. In addition, we believe transformations can be further explored as way to obtain stronger coverage more efficiently as well.

Finally, we conclude that this state of the art research was fundamental to sketch our first approach to solve the problem of coverage criteria analysis on MBT. For future work, we intend to start developing the solution exposed in this report, but always keeping an eye open for other alternative ways that might improve upon it. We hope that by the end of this project, we'll be able to provide PARADIGM-ME with adequate, multifaceted coverage analysis, as well as finding new ways to analyse coverage on models for pre-existing tests and test scripts.

References

[1] E.J. Weyuker. Axiomatizing software test data adequacy. *IEEE Transactions on Software Engineering*, SE-12(12):1128 –1138, December 1986. ISSN 0098-5589. doi: 10.1109/TSE.1986.6313008.

[2] Ibrahim K. El-Far and James A. Whittaker. Model-based software testing. In *Encyclopedia of Software Engineering*. John Wiley & Sons, Inc., 2002. ISBN 9780471028956. URL http://onlinelibrary.wiley.com/doi/10.1002/0471028959.sof207/abstract.

[3] M.M. Eslamimehr. The survey of model based testing and industrial tools. Master's thesis, Master Thesis, Linköping University, 2008.

[4] M. Dhingra, A. Arora, and R. Ghayal. Achievements and challenges of model based testing in industry. In *2012 CSI Sixth International Conference on Software Engineering (CONSEG)*, pages 1 –5, September 2012. doi: 10.1109/CONSEG.2012.6349473.

[5] M. Cunha, A. C. R. Paiva, H. S. Ferreira, and R. Abreu. PETTool: a pattern-based GUI testing tool. In *Software Technology and Engineering (ICSTE), 2010 2nd International Conference on*, volume 1, page V1âĂŞ202, 2010. URL http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=5608882.

[6] ISTQB standard glossary of terms used in software testing, 2012. URL http://www.istqb.org/downloads/viewcategory/20.html. Accessed on February 2013.

[7] IBM information center masthead, May 2007. URL http://pic.dhe.ibm.com/infocenter/clmhelp/v4r0/topic/com.ibm.help.common.resources.doc/banner/banner.htm. Accessed on February 2013.

[8] Paul Ammann and Jeff Offutt. *Introduction to Software Testing*. Cambridge University Press, January 2008. ISBN 9780521880381.

[9] Hong Zhu, Patrick A. V. Hall, and John H. R. May. Software unit test coverage and adequacy. *ACM Comput. Surv.*, 29(4):366–427, 1997. ISSN 0360-0300. doi: 10.1145/267580.267590. URL http://doi.acm.org/10.1145/267580.267590.

[10] Georg Struth. Software verification and testing. URL http://staffwww.dcs.shef.ac.uk/people/G.Struth/COM6854.html. Accessed on February 2013.

[11] P.K. Chittimalli and V. Shah. GEMS: a generic model based source code instrumentation framework. In *2012 IEEE Fifth International Conference on Software Testing, Verification and Validation (ICST)*, pages 909 –914, April 2012. doi: 10.1109/ICST.2012.195.

[12] J. Misurda, J.A. Clause, J.L. Reed, B.R. Childers, and M.L. Soffa. Demand-driven structural testing with dynamic instrumentation. In *27th International Conference on Software Engineering, 2005. ICSE 2005. Proceedings*, pages 156 – 165, 2005. doi: 10.1109/ICSE.2005.1553558.

[13] Qian Yang, J. Jenny Li, and David M. Weiss. A survey of coverage-based testing tools. *The Computer Journal*, 52(5):589–597, August 2009. ISSN 0010-4620, 1460-2067. doi: 10.1093/comjnl/bxm021. URL http://comjnl.oxfordjournals.org/content/52/5/589.

[14] Yong Woo Kim. Efficient use of code coverage in large-scale software development. In *Proceedings of the 2003 conference of the Centre for Advanced Studies on Collaborative research*, CASCON '03, page 145ÃćâĆňâĂIJ155. IBM Press, 2003. URL http://dl.acm.org/citation.cfm?id=961322.961347.

[15] X. Wu, J.J. Li, D. Weiss, and Y. Lee. Coverage-based testing on embedded systems. In *Second International Workshop on Automation of Software Test , 2007. AST '07*, page 7, 2007. doi: 10.1109/AST.2007.8.

[16] Mustafa M. Tikir and Jeffrey K. Hollingsworth. Efficient instrumentation for code coverage testing. *SIGSOFT Softw. Eng. Notes*, 27(4):86ÃćâĆňâĂIJ96, July 2002. ISSN 0163-5948. doi: 10.1145/566171.566186. URL http://doi.acm.org/10.1145/566171.566186.

[17] E.S.F. Najumudheen, R. Mall, and D. Samanta. A dependence graph-based test coverage analysis technique for object-oriented programs. In *Sixth International Conference on Information Technology: New Generations, 2009. ITNG '09*, pages 763 –768, April 2009. doi: 10.1109/ITNG.2009.284.

[18] PHPCoverage - an open-source code coverage measurement tool for php applications, . URL http://phpcoverage.sourceforge.net/?source=navbar. Accessed on February 2013.

[19] PHPUnit manual, . URL http://www.phpunit.de/manual/3.8/en/code-coverage-analysis.html. Accessed on February 2013.

[20] Jscover user manual, . URL http://tntim96.github.com/JSCover/manual/manual.xml. Accessed on February 2013.

[21] Code coverage basics with Visual Studio team system, . URL http://msdn.microsoft.com/en-us/library/dd299398%28v=vs.90%29.aspx. Accessed on February 2013.

[22] Data sheet: Rational PurifyPlus, . URL ftp://public.dhe.ibm.com/software/rational/web/datasheets/version6/pplus.pdf. Accessed on February 2013.

[23] Develop fast, reliable code with IBM Rational PurifyPlus. . URL ftp://public.dhe.ibm.com/software/rational/web/whitepapers/2003/PurifyPlusPDF.pdf. Accessed on February 2013.

[24] A. Pretschner. Model-based testing. In *27th International Conference on Software Engineering, 2005. ICSE 2005. Proceedings*, pages 722 – 723, 2005. doi: 10.1109/ICSE.2005.1553582.

[25] Keith Stobie. Model based testing in practice at microsoft. *Electronic Notes in Theoretical Computer Science*, 111(0):5–12, January 2005. ISSN 1571-0661. doi: 10.1016/j.entcs.2004.12.004. URL http://www.sciencedirect.com/science/article/pii/S1571066104052296.

[26] Wolfgang Grieskamp, Nicolas Kicillof, Keith Stobie, and Victor Braberman. Model-based quality assurance of protocol documentation: tools and methodology. *Software Testing, Verification and Reliability*, 21(1):55–71, 2011. ISSN 1099-1689. doi: 10.1002/stvr.427. URL http://onlinelibrary.wiley.com/doi/10.1002/stvr.427/abstract.

[27] S. Weißleder. Simulated satisfaction of coverage criteria on UML state machines. In *2010 Third International Conference on Software Testing, Verification and Validation (ICST)*, pages 117 –126, April 2010. doi: 10.1109/ICST.2010.28.

[28] S. Weißleder. *Test models and coverage criteria for automatic model-based test generation with UML state machines*. PhD thesis, Humboldt-University Berlin, Germany, 2010.

[29] S. Weißleder and T. Rogenhofer. Simulated restriction of coverage criteria on UML state machines. In *2011 IEEE Fourth International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, pages 34 –38, March 2011. doi: 10.1109/ICSTW.2011.78.

[30] S Weißleder. Coverage simulator. URL http://covsim.sourceforge.net/. Accessed on January 2013.

[31] Margus Veanes, Colin Campbell, Wolfgang Grieskamp, Wolfram Schulte, Nikolai Tillmann, and Lev Nachmanson. Model-based testing of object-oriented reactive systems with spec explorer. In Robert M. Hierons, Jonathan P. Bowen, and Mark Harman, editors, *Formal Methods and Testing*, number 4949 in Lecture Notes in Computer Science, pages 39–76. Springer Berlin Heidelberg, January 2008. ISBN 978-3-540-78916-1, 978-3-540-78917-8. URL http://link.springer.com/chapter/10.1007/978-3-540-78917-8_2.

[32] Model coverage analysis - simulink design verifier, . URL http://www.mathworks.com/products/sldesignverifier/description5.html. Accessed on February 2013.

[33] M. Shafique and Y. Labiche. A systematic review of model based testing tool support. Technical report, Technical Report SCE-10-04, Department of Systems and Computer Engineering, Carleton University, Ottawa, Canada, 2010.

[34] Conformiq Qtronic SG: Semantics and algorithms for test generation, 2008. URL http://www.verifysoft.com/ConformiqQtronicSemanticsAndAlgorithms-3. Accessed on February 2013.

[35] Conformiq Designer 4.4 user manual, 2011. URL http://www.verifysoft.com/ConformiqManual. Accessed on February 2013.

[36] Antti Huima. Implementing conformiq qtronic. In Alexandre Petrenko, Margus Veanes, Jan Tretmans, and Wolfgang Grieskamp, editors, *Testing of Software and Communicating Systems*, number 4581 in Lecture Notes in Computer Science, pages 1–12. Springer Berlin Heidelberg, January 2007. ISBN 978-3-540-73065-1, 978-3-540-73066-8. URL http://link.springer.com/chapter/10.1007/978-3-540-73066-8_1.

[37] Francisco Rebello de Andrade, JoÃčo Faria, AntÃşnia Lopes, and Ana Paiva. Specification-driven unit test generation for java generic classes. In John Derrick, Stefania Gnesi, Diego Latella, and Helen Treharne, editors, *Integrated Formal Methods*, volume 7321 of *Lecture Notes in Computer Science*, pages 296–311. Springer Berlin / Heidelberg, 2012. ISBN 978-3-642-30728-7. URL http://www.springerlink.com/content/a22k88w613346781/abstract/.

[38] F.R. Andrade, J.P. Faria, and A.C.R. Paiva. Test generation from bounded algebraic specifications using alloy. *ICSOFT (2)*, pages 192 – 200, 2011.

[39] Safety-related software development using a model-based testing workflow, January 2013. URL http://www.ibm.com/developerworks/rational/library/

safety-related-software-development/index.html. Accessed on February 2013.

[40] Mark Utting and Bruno Legeard. *Practical Model-Based Testing: A Tools Approach*. Morgan Kaufmann, 2006. ISBN 9780123725011.

[41] José L. Silva, José Creissac Campos, and Ana C.R. Paiva. Model-based user interface testing with spec explorer and ConcurTaskTrees. *Electronic Notes in Theoretical Computer Science*, 208:77–93, 2008. ISSN 1571-0661. doi: 10.1016/j.entcs.2008.03.108. URL http://www.sciencedirect.com/science/article/pii/S1571066108002132.

[42] Ana Paiva, João Faria, and Raul Vidal. Specification-based testing of user interfaces. In Joaquim Jorge, Nuno Jardim Nunes, and João Falcão e Cunha, editors, *Interactive Systems. Design, Specification, and Verification*, volume 2844 of *Lecture Notes in Computer Science*, pages 243–254. Springer Berlin / Heidelberg, 2003. ISBN 978-3-540-20159-5. URL http://www.springerlink.com/content/62g3k4p93h74dm0q/abstract/.

[43] A.C.R. Paiva. Automated specification based testing of graphical user interfaces. *PhD, Engineering Faculty of Porto University, Department of Electrical and Computer Engineering*, 2006.

[44] Ana Barbosa, Ana C.R. Paiva, and José Creissac Campos. Test case generation from mutated task models. In *Proceedings of the 3rd ACM SIGCHI symposium on Engineering interactive computing systems*, EICS '11, pages 175–184, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0670-6. doi: 10.1145/1996461.1996516. URL http://doi.acm.org/10.1145/1996461.1996516.