

FACULDADE DE ENGENHARIA DA UNIVERSIDADE DO PORTO

Test Coverage Analysis

Liliana Borges Vilela



Mestrado Integrado em Engenharia Informática e Computação

Supervisor: Ana Cristina Ramada Paiva

July 29, 2013

Test Coverage Analysis

Liliana Borges Vilela

Mestrado Integrado em Engenharia Informática e Computação

Approved in oral examination by the committee:

Chair: Raul Fernando de Almeida Moreira Vidal

External Examiner: José Francisco Creissac Freitas Campos

Supervisor: Ana Cristina Ramada Paiva

July 29, 2013

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.



Abstract

Nowadays, as software tends to assume increasingly critical roles, the need to assure its quality becomes ever more crucial. Fortunately, several approaches have emerged to help overcome this need, i.e., tools and processes of software testing that help to increase quality in virtually any type of software system. One example of such tools is the so called Model-Based Testing (MBT) tools, that are able to generate test cases from system models.

Such automated tools allow to rapidly create a significant number of test cases, and yet, 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 considered.

This work concerns the implementation of such a coverage analysis tool, PARADIGM-COV, that analyses the quality of the test cases generated by an existing MBT tool, PARADIGM-ME. The tool developed produces several types of coverage information: coverage over PARADIGM graphical model elements, in the form of Test Goal Analysis and Constraint Analysis (useful to assess the quality of the test cases, as well as help to provide appropriate test input during the configuration phase); coverage of a test script in relation to an existing model; dynamic coverage (through test execution analysis, detecting discrepancies between the tests generated and those actually executed); as well as feedback on the percentage of code exercised during test execution.

Resumo

Nos dias de hoje, o software vai assumindo papéis considerados cada vez mais críticos. Como tal, a necessidade de garantir a sua qualidade torna-se cada vez mais crucial. Surgiram felizmente várias abordagens que ajudam a lidar com esta necessidade, nomeadamente, ferramentas e processos de testes de software que ajudam a melhorar a qualidade de praticamente qualquer tipo de sistema. Um exemplo de tais ferramentas são aquelas que normalmente são conhecidas como ferramentas de Testes Baseados em Modelos (*Model-Based Testing*, ou MBT), que são capazes de gerar casos de teste automaticamente a partir de modelos do sistema.

Tais ferramentas permitem gerar rapidamente um número elevado de casos de teste e, no entanto, a quantidade de testes por si só não está directamente correlacionada com a qualidade do processo de testes. De modo a definir essa qualidade, assim como a extensão do sistema que desejamos testar, devemos antes considerar a definição adequada de critérios de cobertura.

Este projecto diz respeito à implementação de uma ferramenta de análise de cobertura, de nome PARADIGM-COV, que analisa a qualidade dos testes gerados por uma ferramenta MBT já existente, PARADIGM-ME. A ferramenta aqui desenvolvida produz vários tipos de informação de cobertura: cobertura sobre elementos pertencentes a um modelo gráfico PARADIGM (útil para avaliar a qualidade dos casos de teste e ajudar o *tester* a fornecer dados apropriados durante a fase de configuração); cobertura de um *script* de teste relativamente a um modelo existente; informação sobre cobertura dinâmica (através de análise de execução de testes, permitindo detectar discrepâncias entre os testes gerados e os executados); bem como informação sobre a percentagem de código do sistema que é exercitado durante a execução dos casos de teste em si.

“Who will guard the guards themselves?”

Juvenal

Contents

1	Introduction	1
1.1	Objectives	2
1.2	Structure	2
2	State of the Art	3
2.1	Key Concepts	3
2.2	Coverage Criteria	4
2.2.1	Structural Graph Coverage Criteria	4
2.2.2	Data Flow Graph Coverage Criteria	5
2.2.3	Criteria Overview	6
2.3	Code Coverage	8
2.3.1	Code Instrumentation	8
2.3.2	Graph Instrumentation	10
2.3.3	Code Coverage Tools	10
2.4	Coverage Analysis in Models	12
2.5	Coverage in MBT Tools	13
2.6	Conclusions	15
3	Test Coverage Tool	17
3.1	Context: PBGT	18
3.2	Architecture and Integration	20
3.3	Model Analysis	23
3.3.1	Test Goal Analysis	23
3.3.2	Constraint Analysis	24
3.4	Script Analysis	32
3.5	Execution Analysis	34
3.5.1	Element Statistics	34
3.5.2	Test Case Statistics	35
3.5.3	Test Trace	35
3.6	Code Coverage	40
4	Case Study	45
4.1	Sample Website	45
4.2	PARADIGM-ME Model	45
4.3	Test Cases	47
4.4	Coverage Tool Output	49

CONTENTS

5	Conclusions and Future Work	55
5.1	Satisfaction of Objectives	55
5.2	Future Work	56
	References	57
A	Constraint Syntax	61
A.1	Constraint Format	61
A.2	Constraint Tables	61
A.3	Examples	63

List of Figures

2.1	Subsumption Relationship between Criteria	7
2.2	Graph Example	8
3.1	Diagram depicting PARADIGM-ME's path generation process.	19
3.2	Final PARADIGM-ME package diagram, showcasing the relations between the coverage plugin and the remainder of the system.	21
3.3	Package diagram of the <i>coverage</i> plugin (PARADIGM-COV).	22
3.4	Example of the conversion of constraint expressions to RTrees.	27
3.5	Example of a restriction table associated with a TGConf.	28
3.6	Simplified diagram of the constraints' support that was added to PARADIGM-ME.	31
3.7	Simplified sequence diagram illustrating the test execution tracking process.	37
3.8	Structure of the <i>ExecutionState</i> class.	40
3.9	Class diagram of the main classes responsible for the instrumentation process.	42
4.1	Screenshot illustrating the main page of the Museum Management website.	46
4.2	Example of the PARADIGM-ME model developed for the first 'level' of the website.	46
4.3	PARADIGM-ME's model created inside the form <i>Registry</i> (8).	47
4.4	Result displayed by the model coverage analysis feature for this website's model.	49
4.5	Result of model coverage analysis on the second level of the model, <i>Registry</i> Form (8).	50
4.6	Restriction table for a TGConf of the element <i>AdminCall</i> (5).	50
4.7	Part of the execution report with check success and test execution measurements.	51
4.8	Code coverage for a single file and overall results for the list of analyzed files.	52

LIST OF FIGURES

List of Tables

3.1	Types of elements in PARADIGM.	18
3.2	Coverage criteria for each PARADIGM UITP.	24
3.3	Definition of the coverage classes used.	25
3.4	Constraint operators.	26
3.5	Possible outcomes on criss-crossing the results of <i>Test Goal Analysis</i> and <i>Constraint Analysis</i>	29
4.1	Configurations performed on the model's first level elements.	48
4.2	Configurations performed on the model's second level, the form <i>Registry</i>	48
A.1	Literal types allowed in a constraint.	61
A.2	Constraint operators.	62
A.3	Reserved fields in use for each UITP type.	62
A.4	Current methods supported by constraints.	62

LIST OF TABLES

Abbreviations

GUI	Graphical User Interface
MBT	Model-Based Testing
PBGT	Pattern-Based GUI Testing
SUT	System Under Test
TG	Test Goal
TGConf	Test Goal Configured
UI	User Interface
UITP	UI Test Pattern
UML	Unified Modelling Language

Chapter 1

Introduction

As software complexity keeps growing, it becomes increasingly difficult to assure its quality. Adding this to the crescent role of software in our daily lives, it becomes easy to verify the ever-growing importance of software testing.

Today more than ever though, we have a wide range of software testing tools available that automate many of the aspects of the quality assurance process. Among 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 is 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 test cases generated, it is 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 are metrics that allow the tester to know the extent to which 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 assess how well the testing process has been performed." [Wey86]

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 [EFW02]. 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, especially when these are automatically generated [Esl08].

The work developed tries precisely to diminish this lack of information on MBT coverage metrics in particular.

1.1 Objectives

The objective of this work is to build a tool for test coverage analysis, PARADIGM-COV, which provides several types of coverage information in order to: aid the tester to provide appropriate input test data during the test configuration phase, help access the quality of PARADIGM-ME's test suites, evaluate how those tests fare when actually executed against the target system, as well as provide a more exact indicator on the progress of testing itself.

This tool is developed in the context of the Pattern Based GUI Testing (PBGT) project [MP13], which aims to build a Model-Based Testing environment designed to support automated testing of graphical user interfaces by use of common interface patterns. With the Model-Based Testing strategy, the test cases are generated from abstract representations (models) of the system under test. In PBGT, these models are constructed using a highly abstract graphical modelling language (PARADIGM), with elements dubbed UI Test Patterns defining generic test strategies that can be applied for testing UI patterns over their different implementations, thereby promoting reuse. As such, the developed PARADIGM-COV is to be integrated into PBGT's Model-Based Testing environment, PARADIGM-ME, effectively equipping this MBT tool with coverage analysis facilities.

Regarding coverage analysis in particular, the objectives set out for this tool include: provide feedback on model coverage, both through evaluation of UI Pattern test requirements and analysis of test preconditions; perform script coverage analysis, allowing to compare custom-made test scripts with an existing PARADIGM-ME model; inclusion of dynamic coverage analysis, evaluating the relation between the tests defined in the model and those actually executed on the target system; and to feature code coverage analysis, as to provide the tester with precise indicators on the percentage of code covered during the testing process.

In order to achieve these goals, the first objective was firstly towards gaining an understanding of the PARADIGM modelling language, as well as PARADIGM-ME's system itself. In order to obtain seamless integration between this system, PARADIGM-COV was to be developed using the same technologies, of which stands out the Eclipse Modelling Framework.

1.2 Structure

This document is divided into 5 chapters. Following this introduction, a research on the state of the art regarding coverage tools and methodologies is presented (Chapter 2). Following, Chapter 3 proceeds then to present the work developed, its context, and the methodologies used, while providing also some information on the implementation details of the work in question. Throughout Chapter 4 a case study is presented, illustrating the coverage tool developed at work with a real-world application. Finally, in Chapter 5 we sum up the work developed, presenting some conclusions as well as some possible paths of future work.

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 will 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*” [IST12].

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 [IBM07].

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 [IST12, IBM07].

Test Path A path in itself consists in a sequence of events of a component or system, from an entry to an exit point [IST12]. Hence, a test path represents the execution of a sequence of test cases [AO08] 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 [Wey86].

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 [ZHM97].

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 [AO08].

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 will 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 [Esl08].

Inside the graph coverage criteria, these can be further classified according to whether they are based on the program's control flow (also known as structural criteria), or based on its data flow. Following, we will give a brief run-down of each of them. All definitions here provided are based on the ones given by Ammann [AO08], 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* (a single edge or a 'pair of nodes'), 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 2*, or 'in pairs'.

Prime Path Coverage A simple path consists in a path which has no internal loops. In other words, while it is 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 [AO08].

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*) [AO08]. 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 [Str]. 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 is 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 will 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 cover *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* [Str].

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

State of the Art

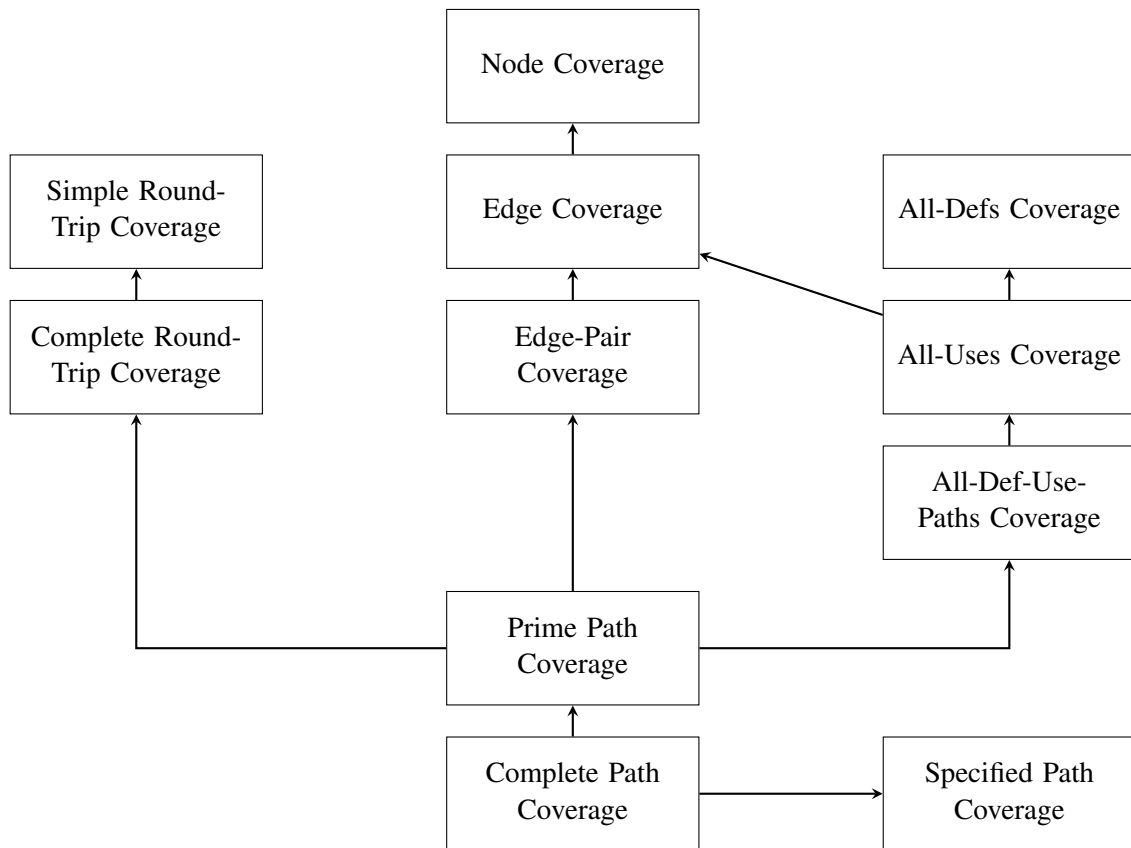


Figure 2.1: Diagram representing the subsumption relationship between several coverage criteria (adapted from the one provided by Amman in [AO08]).

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).

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 coverage requirements:

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

Which actually results in the coverage of the same test path:

- Test Paths: (1, 1, 2)

As a reminder, Edge Coverage covers paths with length up to one (all edges), while Edge-Pair Coverage includes those with length up to two (all edges and pairs of edges) (paths of length zero,

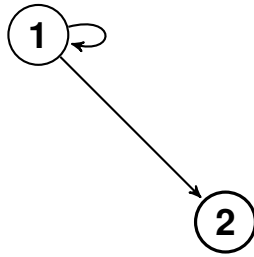


Figure 2.2: Example of a graph with a loop path of length one.

while allowed, were not considered here). This inclusion of paths below one and two in length guarantees the subsumption of weaker criteria such as node coverage. As for Prime Path Coverage, while prime paths themselves do not 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 is 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 will 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 does not change the behaviour of the application itself, but collects information on its execution [AO08]. 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]*" [CS12].

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 is 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 [MCR⁺05].

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 [CS12]:

- Source code instrumentation: as the name itself indicates, it is 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 is usually very dependent on the architecture and compiler used.

Though instrumentation allows an effective way to monitor the behaviour of a program, it is 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 [YLW09].

This additional overhead problem is particular significant when wanting to add coverage to large industrial systems [Kim03], and even more so when it comes to embedded systems, in which the execution time is a critical factor [WLWL07]. 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 [TH02].

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 [MCR⁺05].

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 [NMS09].

As for how this is achieved, Ammann presents several examples on how instrumentation for coverage analysis can be accomplished through graphs [AO08]. 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 are executed. By the end of the execution, we will 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. [AO08].

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 will 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 coverage analysis tool we intend to develop is geared specially towards web interface testing, we will 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 [YLW09].

In regards to the PHP code coverage tools available, we find software such as PHPCoverage¹, 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 [Pac]. Another tool, Xdebug², 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 PHPUnit³ [Ber] and the library PHP_CodeCoverage⁴. As with PHPCoverage, probes are placed as to track the execution of each line.

¹<http://sourceforge.net/projects/phpcoverage/>

²<http://xdebug.org/>

³<https://github.com/sebastianbergmann/phpunit/>

⁴<https://github.com/sebastianbergmann/php-code-coverage>

As for Javascript, JSCover⁵ 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 [too]. Saga⁶ on the other hand, provides coverage information for both standalone and inline scripts. Other examples of tools providing code coverage for this language include Istanbul⁷, and the Blanket library⁸.

In what concerns the Java programming language, there is a wide choice in tools. For example, Emma⁹ 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, Cobertura¹⁰, 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 Clover¹¹, CodeCover¹² and Coverlipse¹³, with these last two being available as Eclipse plugins.

For the .NET platform, we have available standalone tools such as NCover¹⁴, OpenCover¹⁵, and dotCover¹⁶, being that code coverage is also supported natively by IDEs such as Visual Studio¹⁷ [Mic]. 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, PurifyPlus¹⁸ 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 [IBMa, IBMb]. 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

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/>

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/>

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 [YLW09].

2.4 Coverage Analysis in Models

As we have 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 [Pre05]. Other times, model coverage itself is not even considered, since coverage can be used as merely a means to guide automatic test generation, and not an end by itself [Sto05, GKSB11].

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 among several types of design models, being UML ones amongst the most used. 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 is 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 is 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 [Wei10a, Wei10b]. 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 [WR11]. Coverage Simulator is a tool prototype that intends to demonstrate this approach [Wei].

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 [Wei10b], 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 is not 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 is 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 isolate the techniques used to evaluate coverage from 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) [Sto05].

A similar case can be found with Spec Explorer²⁰. Here, what is defined as coverage criteria is effectively used as test selection criteria, being Transition Coverage the most used one [GKSBB11, VCG⁺08]. A third MBT tool we looked into, Simulink Design Verifier²¹, uses formal methods to identify design errors in the models. In this one, model coverage is once again used to generate additional tests. In regards to coverage criteria though, it supports not only commonly used criteria,

²⁰ <http://visualstudiogallery.msdn.microsoft.com/271d0904-f178-4ce9-956b-d9bfa4902745>

²¹ <http://www.mathworks.com/products/sldesignverifier/>

such as Branch Coverage, but also reports coverage on metrics like test objectives or constraints [Mat]. Other tools that seem to use the coverage criteria to generate test cases include Conformiq Qtronic²² and Smartesting CertifyIt²³ [SL10].

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 do not seem to offer a vast array of choice on this matter. Qtronic is the only one we have found to supply All-Paths coverage for acyclic graphs for instance [too08, too11], along with some other additional coverage criteria not mentioned in this report [SL10].

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 [GKSB11, VCG⁺08]. Other tools make use of *coverage checkpoints* in the model, also known as *coverage items*, of which Conformiq Qtronic is also an example [Hui07]. With SCADE Suite²⁴, the coverage analysis is done by verifying the activation of each element in the model as the system is executed. It is 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 [RdAFLP12].

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 [AFP11]. This technique was inclusively further explored to provide support in generating test cases for Java generic classes in particular [RdAFLP12].

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 are the criteria being analysed. Such we have found to be the case with IBM's Rational Rhapsody²⁵ for instance [IBM13]. 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 [SL10].

²²<http://www.conformiq.com/>

²³<http://www.smartesting.com/index.php/cms/en/product/certify-it>

²⁴<http://www.esterel-technologies.com/products/scade-suite/>

²⁵<http://www.ibm.com/developerworks/rational/products/rhapsody/>

2.6 Conclusions

In this chapter it was presented the current state of the art regarding coverage analysis.

There are several distinct coverage criteria, being these usually divided into two main branches: structural based, and data-flow based [AO08]. Coverage tools also abound, and each offers its own array of capabilities and implements a given set of coverage criteria. And yet, we have not found many that offer several types of distinct criteria. The common tool seems to provide a relatively minor array of options regarding criteria, and when they do offer a larger choice, the criteria themselves seem to implement similar base approaches. With the development of the PARADIGM-COV tool, this is something we intend to address: to provide the tester with the most varied set of coverage analysis approaches possible, and attempt to bring together the strengths of each.

State of the Art

Chapter 3

Test Coverage Tool

The more complex a model gets, the harder (i.e., error prone and expensive time-wise) it is to manually keep track of all the configurations inserted. PARADIGM-COV intends to automate such task, freeing the tester to focus on other aspects of the quality process. As such, PARADIGM-COV tool has four main goals:

- *Perform model coverage analysis*: to provide feedback on whether the test configurations defined are considered 'adequate' and the restrictions are 'realizable';
- *Script Analysis*: to evaluate whether a given test script is able to cover an existing PARADIGM model;
- *Execution analysis*: to provide feedback on the actual test execution process through dynamic coverage analysis;
- *Code analysis*: to provide feedback on the degree to which the actual SUT code is exercised during the test execution process.

Throughout this chapter we will expand on the means used to achieve these goals.

3.1 Context: PBGT

Before delving into the developed tool itself, it matters to supply a brief exposition into the project in which this work is integrated, the Pattern-Based GUI Testing (PBGT) project.

The goal of Pattern Based GUI Testing [MP13] is to provide a means to test graphical user interfaces (GUIs) by providing generic test strategies for testing recurrent behaviour. This approach relies on the fact that most GUIs end up having similar elements and behaviour (the so called UI Patterns), which can be reused across several applications. PBGT defines generic test strategies for UI Patterns able to test slightly different implementations of these after a configuration phase. Those generic test strategies are called UI Test Patterns (UITP). A UI Test Pattern defines a test strategy as a set of Test Goals (TG) with the form

$$\langle Goal;V;A;C;P \rangle \tag{3.1}$$

where *Goal* is the name of the test strategy, *V* is a set of pairs relating the input variables with test input data, *A* is the sequence of actions to perform during test case execution, *C* is a set of checks to perform in order to evaluate the test results, and *P* is a Boolean expression defining the states in which the sequence of actions (*A*) ought to be executed.

These UI Test Patterns are defined within a graphical Domain Specific modelling Language (DSL) called PARADIGM. This DSL has structural elements (to structure the model in different levels of abstraction), behavioural elements (UI Test Patterns), Init and End elements (to mark the start and end points of a model), and connectors to establish relations among the elements and define their sequencing (Table 3.1).

Table 3.1: Types of elements in PARADIGM.

Element Type	Element
Structural Elements	Group
	Form
Behavioural Elements	Login
	Input
	MasterDetail
	Sort
	Find
	Call
Other	Init
	End

PARADIGM-ME [CPFA10, MP13] is the modelling environment that supports building models written in PARADIGM, that generates test cases from those models, establishes the mapping

of model elements with GUI controls, and executes test cases over the system under test (SUT). This is the tool into which PARADIGM-COV is integrated.

To generate test cases from a PARADIGM model, the tester needs to go through a manual configuration step in which he selects the TGs for each UI Test Pattern within the model, and for each of those TGs, provides test data: enters test input data, selects the checks to perform, and defines the preconditions. A Configured Test Goal (TGConf) is an instance of a TG in the sense that it has the test data already defined. It is possible to assign the same TG more than once for a UITP by providing different test data.

In order to generate test paths, and since a PARADIGM model can be structured into different levels of abstraction, PARADIGM-ME starts by flattening the model, i.e., it replaces recursively all structural elements (Forms) within a hierarchical level of a model by their internal elements, present at the next level, until all Forms are discarded. After this flattening process, PARADIGM-ME calculates the set of paths (SPaths) that go from the Init to the End elements within the model. Following, it expands every path within SPaths into test cases according to the TGConf defined for the UITP within each path. To conclude, it analyses the preconditions to determine which steps that can be performed from said test cases.

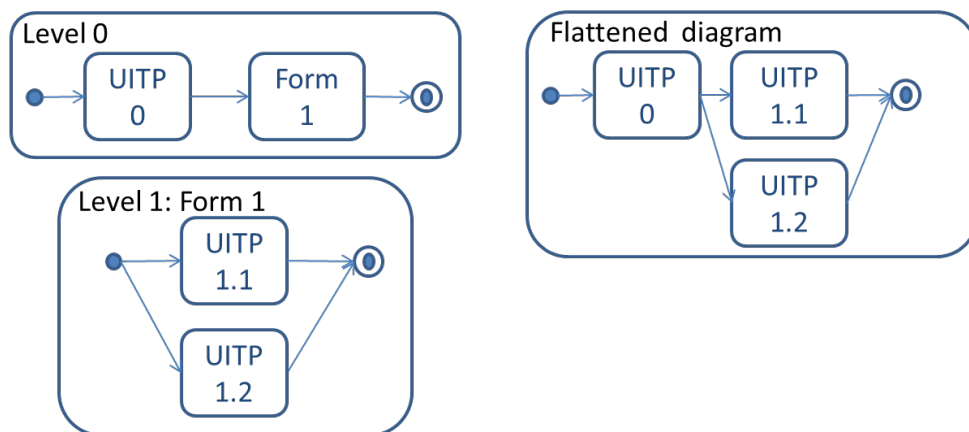


Figure 3.1: Diagram depicting PARADIGM-ME's path generation process.

As an example of the test generation process, Fig. 3.1 illustrates a model written in two levels of abstraction (Level 0 and Level 1) in the left side and the corresponding flattened diagram in the right side. In this case SPaths has two paths:

```
Path1: Init -> UITP0 -> UITP1.1 -> End
Path2: Init -> UITP0 -> UITP1.2 -> End
```

Let's consider that UITP1 has two configurations defined ($c01, c02$), UITP 1.1 has one configurations defined ($c111$) and UITP 1.2 has two configurations ($c121, c122$), hence the final test suite would be constituted by the following test cases:

Test Coverage Tool

```
tc1: Init -> UITP0(c01) -> UITP1.1(c111) -> End
tc2: Init -> UITP0(c02) -> UITP1.1(c111) -> End
tc3: Init -> UITP0(c01) -> UITP1.2(c121) -> End
tc4: Init -> UITP0(c02) -> UITP1.2(c121) -> End
tc5: Init -> UITP0(c01) -> UITP1.2(c122) -> End
tc6: Init -> UITP0(c02) -> UITP1.2(c122) -> End
```

, where $UITP_x(cy)$ is the instance (or TGConf) of the UI Test Pattern number x with configuration values defined in cy . Above, the first two TGConfs are generated by expanding the first test path (*path1*) being the final four generated by expanding the second test path (*path2*).

Afterwards, the test cases are analysed regarding their preconditions. Considering a test case (tc) as a sequence of N steps ($Init - S1 - \dots - SN - End$), if a precondition of a step i (such that $1 \leq i \leq N$) does not hold, the tc becomes $Init - S1 - \dots - Si - End$. This happens because if a step cannot be executed, the following steps cannot also be executed. This diminishes the length of the final test cases generated, and, in other words, ensures all preconditions are true.

We say that a test case n belongs to the same family of another test case m if they were obtained by expanding the same test path p . In the example given, $tc1$ and $tc2$ are of the same family, while $tc3$, $tc4$, $tc5$, and $tc6$ belong to a different family.

3.2 Architecture and Integration

Since the coverage features were to be integrated seamlessly into the PARADIGM-ME environment, we devised a modular coverage plugin that would deliver the intended functionalities. However, in order to gather the necessary coverage information, some additions were also made to several aspects of the host system itself (Fig. 3.2), namely:

- Implementation of constraints' support (package *paradigm*);
- Added path information to generated scripts (*scriptGenerator*, *testExecution*);
- Altered test execution as to provide failure-detection mechanisms (*testExecution*);
- Added the necessary dialogue and model customization to display the coverage analysis results (*diagram*, *edit.ui*);
- Improved functionalities related to test path generation mechanisms (*scriptGenerator*, *testExecution*).

Of these, the addition of model constraints in particular will be further addressed in the section 3.3.2.

Since the coverage tool was to be integrated into the existing system, the technologies chosen followed the ones used by PARADIGM-ME itself. As such, the PARADIGM-COV was developed in Java as an Eclipse plugin, and makes use of the Eclipse Modelling Framework (EMF).

Test Coverage Tool

As for the integration with the existing application, the relation of PARADIGM-COV with the remainder of the system can be seen in Fig. 3.2. The plugin, here represented by the *coverage* package, imports the necessary model and element data from the *paradigm* and *diagram* packages. These are, for all effects, part of the 'core' packages of PARADIGM-ME, being generated by the EMF framework and holding all data about the models and their data structures. In turn, the *coverage* plugin provides functionality to the package *testExecution*, enabling the tracking of the executed tests and, as a result, allowing to realize dynamic coverage analysis.

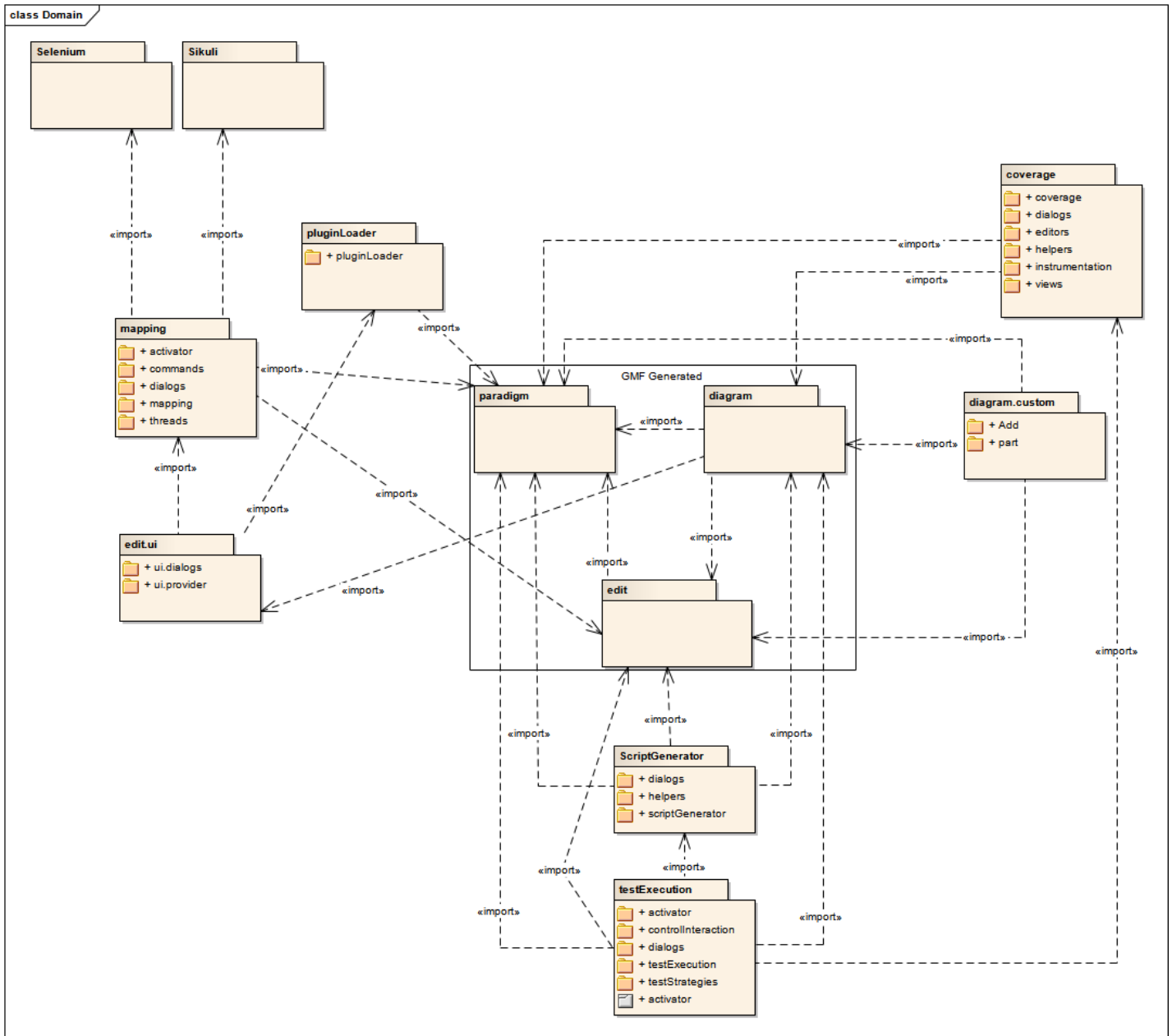


Figure 3.2: Final PARADIGM-ME package diagram, showcasing the relations between the coverage plugin and the remainder of the system.

Test Coverage Tool

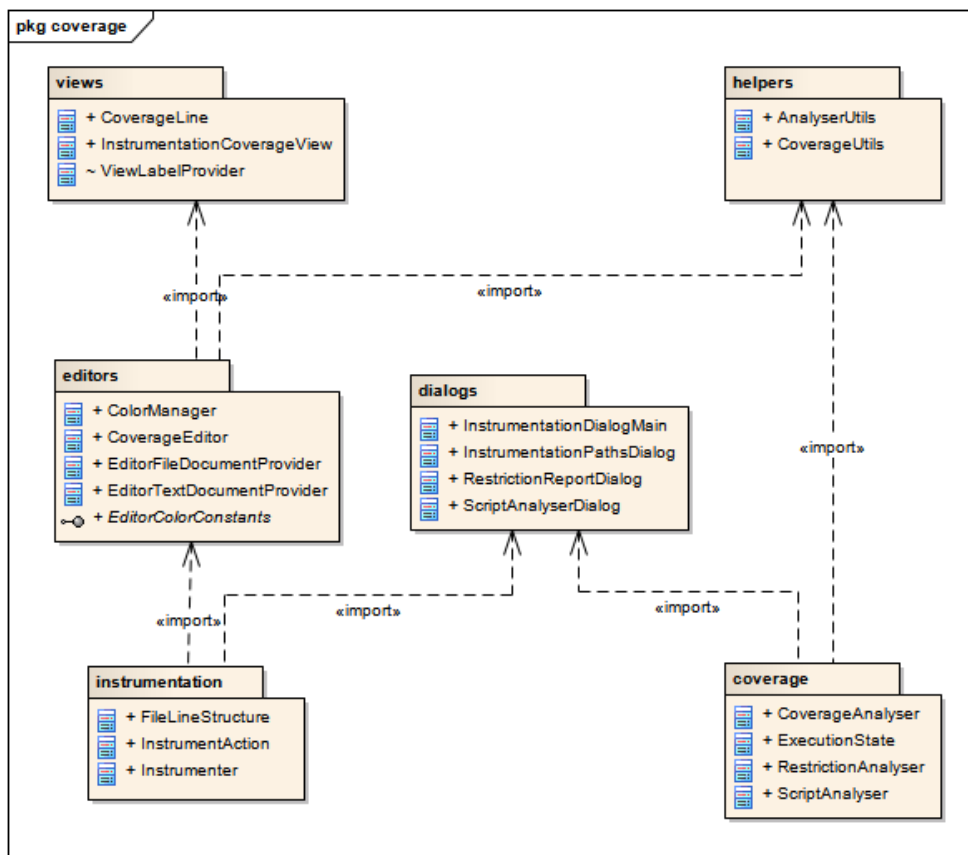


Figure 3.3: Package diagram of the *coverage* plugin (PARADIGM-COV).

Regarding the architecture of the PARADIGM-COV plugin itself, its package structure, as well as their dependencies, is shown in Fig. 3.3. As can be seen, PARADIGM-COV is divided into six packages: *coverage*, *instrumentation*, *dialogs*, *editors*, *views*, and *helpers*.

The package division scheme and their naming were made taking into account the standards already being used by PARADIGM-ME. As such, *coverage* and *instrumentation* contain the higher-level functionalities, implementing the respective Eclipse handlers for each coverage analysis type available. The package of name *dialogs* on the other hand, contains all Java SWT dialogs used throughout the plugin. As for *editors* and *view*, as their very name indicates, they contain the custom Eclipse views and editors designed specifically for the developed plugin. Finally, *helpers* contains general functional methods used throughout that are not tied to a particular feature.

We will now expand on each of the developed plugin's functionalities in particular.

3.3 Model Analysis

The first goal of PARADIGM-COV is then to provide model coverage information. The model coverage analysis functionality is grounded on two main premises: (1) analysis of the test goals (test requirements), as to define whether all Test Goals (TGs) associated with a given UI Test Pattern element are being configured (*Test Goal Analysis*); and (2) analysis of preconditions in order to evaluate if it is possible to execute all the Configured Test Goals (TGConf), i.e., evaluate if the model configurations allows reaching states where precondition holds (*Constraint Analysis*).

3.3.1 Test Goal Analysis

Test Goal Analysis aims at checking if there are configurations (TGConf) defined for every TG within UITPs belonging to a PARADIGM model. As such, Test Goal coverage is defined differently for structural and behavioural elements.

Structural elements by themselves realize no test strategies, since their purpose is to allow the tester to structure the model in groups (Group element) or into different levels of abstraction (Form element). As such, for any structural elements Se , its goal coverage $gCov(Se)$ is considered to be the average of the goal coverage of the N elements e within it:

$$gCov(Se) = \frac{\sum_{i=1}^N gCov(e_i)}{N} \quad (3.2)$$

Behavioural elements on the other hand, are UI Test Patterns (UITP) defining a test strategy as a set of Test Goals (TG). To reach full coverage here, for every Test Goal within a UITP, the tester needs to provide, at least, one configuration (TGConf). A TGConf is an instance of a TG that the tester configured with test input data assigned to input variables, checks to perform, and test preconditions. As such, a behavioural element is fully covered in regards to its Test Goals when there are configurations defined for all its Test Goals (according to Formula 3.3). In turn, a PARADIGM model is fully covered if all its UITP are covered.

$$\forall tg \in TG, \exists e \in TGConf | c.Goal = tg \quad (3.3)$$

As an example, consider the Login behavioural element. This UI Test Pattern defines a test strategy with two Test Goals ($TG=G_LV, G_LINV$) for testing both successful and failed logins. Should the user test only successful logins, there is no guarantee the system will behave as expected when time comes to submit an invalid one. At worst, the user could find out later that even invalid logins gave access to the system. Acceptable coverage criteria for a Login element could then be for the tester to include configurations for both valid and invalid login specifications or, in other words, to exercise all its test goals. The information relating the coverage criteria as the set of test goals defined for each UITP can be consulted in Table 3.2.

Table 3.2: Coverage criteria for each PARADIGM UITP.

Test Type	Set of Test Goals
Call	{ <i>G_Call</i> } – test the call result.
Find	{ <i>G_Found</i> , <i>G_notFound</i> } – test when the 'find' returns values and when it does not find anything.
Input	{ <i>G_IV</i> , <i>G_IINV</i> } – test for input valid and for input invalid.
Login	{ <i>G_LV</i> , <i>G_LINV</i> } – test for login valid and for login invalid.
Sort	{ <i>G_SRTASC</i> , <i>G_SRTDESC</i> } – test the sort for ascending and descending.
MasterDetail	{ <i>G_MD</i> } – test the detail values for two specific master values.

Anytime that, through the configuration process, the tester defines configurations for every TG related to a UITP, meeting the criteria described in Table 3.2, the element is classified as being *Covered*. This is but one of three distinct coverage states an element may take, being these: *Covered*, *Partial*, and *Uncovered*. As such, the UITP is considered *Uncovered*, when there are no TGConfs defined, and *Partial*, in all the other cases. A fourth, special state, *Undefined*, indicates a coverage analysis has yet to be performed and, as such, no coverage information regarding that element may be presented.

As a way to make the assessment of the model more intuitive, the approach here taken to display coverage data relies heavily on 'colour codes'. For every coverage state listed was defined its own representative colour. As such, and keeping with the usual conventions for depicting coverage information, the colours used are green, yellow and red, as can be seen in Table 3.3. Thus, the several model elements are coloured according to each coverage state: elements with full coverage will gain a green background, elements with no defined tests will be marked in red, and all the remaining elements between these two states will be coloured in yellow, to note that they reach only partial coverage. This colour association is used consistently throughout the several types of coverage analysis provided, as will be seen further ahead.

3.3.2 Constraint Analysis

In testing real-world applications, it is relatively common to come across elements that can only be executed in certain circumstances. As examples, one could consider any system that required a valid login prior to executing certain tasks, or a functionality that is only accessible if a given checkbox has been selected. As such, and as a second component of the Model Coverage Analysis, it is then necessary to consider the existence of preconditions to executing certain test cases.

3.3.2.1 Constraint Definition

Taking into account this need, we added support for the insertion and management of test configuration (TGConf) preconditions to PARADIGM-ME. A precondition (also known as constraint or restriction) defines the states in which a given TGConf may be executed. In other words, a TGConf may be executed only when all its preconditions are evaluated as being True.

Test Coverage Tool

Table 3.3: Definition of the coverage classes used.

Colour Code	Coverage State	Definition
Green	Covered	The element is considered to reach full coverage: $cov = 1.0$
Yellow	Partial	The element has tests defined, but not enough to reach a full coverage state; in other words, the coverage percentage of the element is between 0% and 100%: $1.0 > cov > 0.0$
Red	Uncovered	The element has no tests associated with it, hence, it is considered to have 0% coverage: $cov = 0.0$
White	Undefined	<i>Default state</i> : there is no coverage information to present for this element. $cov = ?$

Syntax

First off, it mattered to establish the syntax used to express a precondition. As seen in Chapter 3.1, each TG, and thereby TGConf, contains a set of input pairs V . Each pair associates an input variable (in PARADIGM-ME, *field*) with its test input data. For instance, a TGConf's V pairs for a Login UITP could be the following:

$$V = \{(username, "johndoe"), (password, "12345")\} \quad (3.4)$$

Different TGConf have different values (input test data) assigned to its input variables.

Thus, at its basic form, a constraint consists in the relation of a TGConf's V pair to either another V pair or a literal, through the use of restriction operators. Restriction wise, a field *tgcField* is expressed as:

```
tgcField := <UITP>.<vField> ,
```

where the UITP is the element to which the target TGConf belongs, and *vField* is the target TGConf's field to be used with that constraint.

The constraint itself is expressed through the format:

```
<tgcField>[.<method>] <operator> <tgcField|literal>
```

A description of the several operators allowed, as well as some usage examples, are shown in Table 3.4.

Table 3.4: Constraint operators.

Operator	Description	Constraint Example
<code>==</code>	equal	<code>Login.username == Input.field</code>
<code>!=</code>	different	<code>Input.field != "pass"</code>
<code><</code>	smaller than	<code>Login.username.len() < 5</code>
<code>></code>	bigger than	<code>MD.countryName.len() > 2</code>
<code>contains</code>	contains	<code>MD.__details contains {d1, d2, d3}</code>

Additionally, and for sake of keeping constraint definitions more adaptable, we set aside some 'reserved' keywords. These keywords, or restricted fields, are always preceded by a double underscore ('__').

Some of these restricted fields provide a way to apply a restriction to all TGConfs within a UITP if needed. An example of this use is found, for instance, with `__fields`, which returns the set of all input variables associated with a TGConf.

A more complete exposition on the syntax defined for constraints, respective features implemented, as well as some usage examples, is made available in [Appendix A](#).

Constraint Expressions

Having exposed the format of a single constraint, it is important to note that these may additionally be grouped in 'constraint expressions' through the use of the logical operators AND and OR. These expressions may also impart priorities in the constraint analysis by the use of parentheses. When no parentheses are present, the expression will simply be processed from left to right. At its most basic form, the format for a constraint expression can then be expressed as:

```
(constraint <AND|OR> constraint)
```

These expressions are processed in the form of a binary restriction tree, or RTree. In a RTree, each leaf is a constraint, while all non-terminal nodes represent a logical operator. In this tree, nodes with bigger depth have higher priority. Thus, let us consider n_i as a generic RTree node, where $i \geq 0$ and represents the node depth in the tree. Let us then take the nodes n_x and n_y : if $x > y$, the node n_x has higher priority, and will be processed before the node n_y . In order to better illustrate this, as well as to demonstrate the overall conversion between an expression and a RTree, we give an example of the processing for two similar constraint expressions, `cexpr1` and `cexpr2`, where c_i represents a single constraint.

Test Coverage Tool

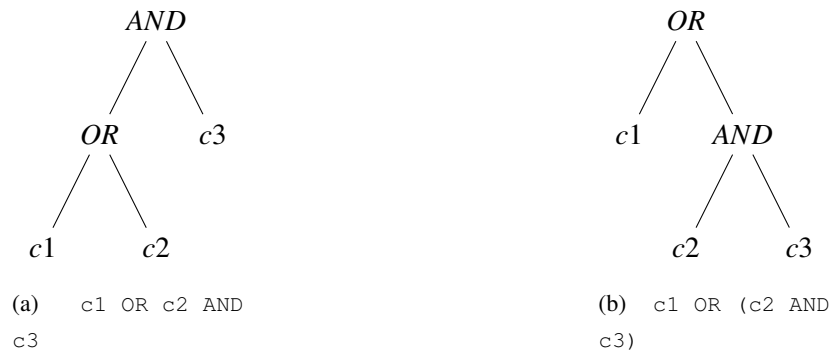


Figure 3.4: Example of the conversion of constraint expressions to RTrees.

Let us now assume that, by analysing the constraints, the values returned for each of them are as follows:

$c1 = \text{True}$
 $c2 = \text{False}$
 $c3 = \text{False}$

Thus, the final result for both these constraint expressions is:

(a) $c1 \text{ OR } c2 \text{ AND } c3 = \text{True OR False AND False}$
 $= \text{True AND False}$
 $= \text{False}$

(b) $c1 \text{ OR } (c2 \text{ AND } c3) = \text{True OR (False AND False)}$
 $= \text{True OR False}$
 $= \text{True}$

The same schematic holds for more complex expressions, some examples of which are also present in [Appendix A](#).

In order for the tester to be able to supply the desirable preconditions, a Restriction Table was provided for each TGConf in the model. An example of such table is presented in [Fig. 3.5](#).

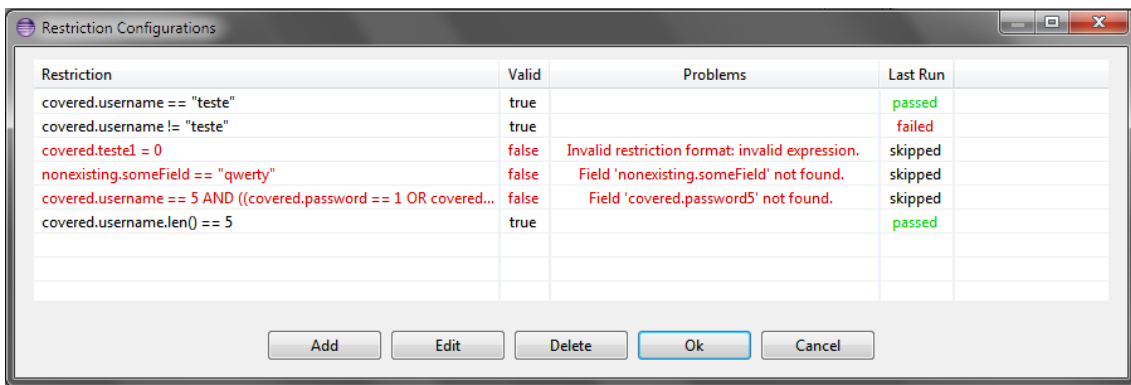
Each entry of the table contains either a constraint or a constraint expression. For all effects, constraint entries have an AND relation, i.e., all entries must evaluate to True for the respective TGConf to be executed. It is up to the tester to divide the several expressions into table entries as he sees fit, given that the main goal this division is ultimately to increase readability for the end-user.

Besides containing a constraint expression, each table entry displays any errors that the expression might have. This verification for errors is made when the user inserts or edits a table

Test Coverage Tool

entry, time at which both the error column and the column concerning the entry's validity is automatically updated. A non-valid table entry is therefore one whose constraints contain errors, be they syntax related or due to referencing a non-existent field.

Finally, the very last column shows the result of the last Constraint Analysis made on the model. The constraint expression is evaluated as *failed* if its result is False, i.e., there is never a state in which the given preconditions hold. Such case is indicative that the associated TGConf will never be executed on the system. On the other hand, when the model contains at least one state which verifies the precondition, this one is tagged as *passed*. The entry tables that contain errors at the time of the analysis are labelled as being *skipped*.



Restriction	Valid	Problems	Last Run
covered.username == "teste"	true		passed
covered.username != "teste"	true		failed
covered.test1 = 0	false	Invalid restriction format: invalid expression.	skipped
nonexisting.someField == "qwerty"	false	Field 'nonexisting.someField' not found.	skipped
covered.username == 5 AND ((covered.password == 1 OR covered...	false	Field 'covered.password5' not found.	skipped
covered.username.len() == 5	true		passed

Figure 3.5: Example of a restriction table associated with a TGConf.

3.3.2.2 Analysis of Constraints

Having added constraint support, one can now evaluate them regarding their coverage. Analysing only if there are TGConfs defined for every TG inside a UITP is not enough to guarantee that the corresponding test cases will be exercised. Every TGConf may have preconditions that must hold in order for it to be executed, so it is necessary to check whether the TGConf will be executed at least once, i.e., whether the preconditions for TGConf are ever met.

Ergo, an element is considered valid restriction-wise if, for every configuration TGConf belonging to said element, there is a state (variables and corresponding input data) in which its associated precondition holds (Formula 3.5). This precondition checking can be especially useful as a way to help providing test input data (data needed to cover all possible paths of the model) and to restrict the total number of test cases generated by PARADIGM-ME.

$$\forall c \in TGConf, \exists s : State | c.P(s) = true \quad (3.5)$$

The information about the elements containing preconditions that do not hold is presented using the same colour codes and coverage terminology used before, but this time as a box around

Test Coverage Tool

the label of the corresponding UITP element. As such, an element is *Covered* (green) when all its restrictions are complied with, *Partial* (yellow) where only some are met, and *Uncovered* (red) when no preconditions are ever True.

It is possible to see the result of Test Goal Analysis and Constraint Analysis at the same time over the model in order to get a broader picture of the coverage analysis. This allows to combine the results from both approaches and obtain a better overview of the coverage on that element. The potential outcomes of such combination between Test Goal Coverage and Constraint Analysis can be consulted in Table 3.5.

Table 3.5: Possible outcomes on criss-crossing the results of *Test Goal Analysis* and *Constraint Analysis*.

{Test Goal Analysis, Constraints Analysis}	Result
{ <i>Covered</i> , <i>Covered</i> }	All test goals for this UITP used, all constraints met: all test configurations TGConfs for this UITP will be executed.
{ <i>Covered</i> , <i>Partial</i> }	All test goals used, some constraints met: only TGConfs that meet their preconditions will be executed.
{ <i>Covered</i> , <i>Uncovered</i> }	All test goals used, no constraints met: test paths will not include any TGConfs belonging to this element.
{ <i>Partial</i> , <i>Covered</i> }	Some test goals for this element used, all constraints met: all TGConfs defined for this element will be executed.
{ <i>Partial</i> , <i>Partial</i> }	Some test goals used, some constraints met: only TGConfs that meet their preconditions will be executed.
{ <i>Partial</i> , <i>Uncovered</i> }	Some test goals used, no constraints met: test paths will not include any TGConfs belonging this element.
{ <i>Uncovered</i> , <i>Covered</i> }	<i>Invalid</i> : there are no TGConfs defined, hence, there can be no constraints defined for this element.
{ <i>Uncovered</i> , <i>Partial</i> }	<i>Invalid</i> : there are no TGConfs defined, hence, there can be no constraints defined for this element.
{ <i>Uncovered</i> , <i>Uncovered</i> }	No test goals defined, no constraints met: no tests will be generated for this element.

Implementation

Test Goal Analysis

With regards to the implementation details, the attribute *Coverage*, concerning the definition of the coverage classes (*Covered*, *Partial*, *Uncovered*), was created inside the *paradigm* package

through the EMF.ecore files and, as such, is generated automatically. This class merely implements the four coverage classes exposed in 3.3, defining it as another property of a model element.

The Test Goal analysis on the other hand, is executed through methods defined in the coverage plugin. In the *coverage* package, the class *CoverageAnalyser* handles the request for executing Test Goal analysis, and runs through the EMF model analysing each element in the diagram one-by-one. The verification of the coverage criteria themselves however, is made by the helper class *CoverageUtils*. This one contains the method *isCovered* which, given an element, returns its coverage status regarding Test Goal coverage.

Constraint Analysis

As was the case with *Coverage*, *Restriction* was defined in the package *paradigm*. The simplified class structure for this functionality can be seen in Fig. 3.6. As stated, each UITP (behavioural element) can have several Test Goals configured (TGConfs) (represented by an *Entry* in PARADIGM-ME). In turn, each TGConf can have more than one precondition (*Restriction* object).

Following, we give a brief exposition on each of the classes related to restrictions themselves.

Restriction representing a single restriction expression.

RestrictionState contains information about the state of the restriction expression the last time this one was analysed (*undefined, skipped, failed, passed*).

RNode represents an abstract node of the binary restriction tree.

RNodeLogical an non-terminal RNode, i.e., an RNode that establishes a boolean relation AND/OR between its two children.

LogicalOperator representing a AND or OR relation.

RNodeExpression a leaf of the RTree, holding a single constraint expression.

RestrictionOperator operator used to relate the two fields of a constraint (*equal, different, smaller than, bigger than contains*).

Test Coverage Tool

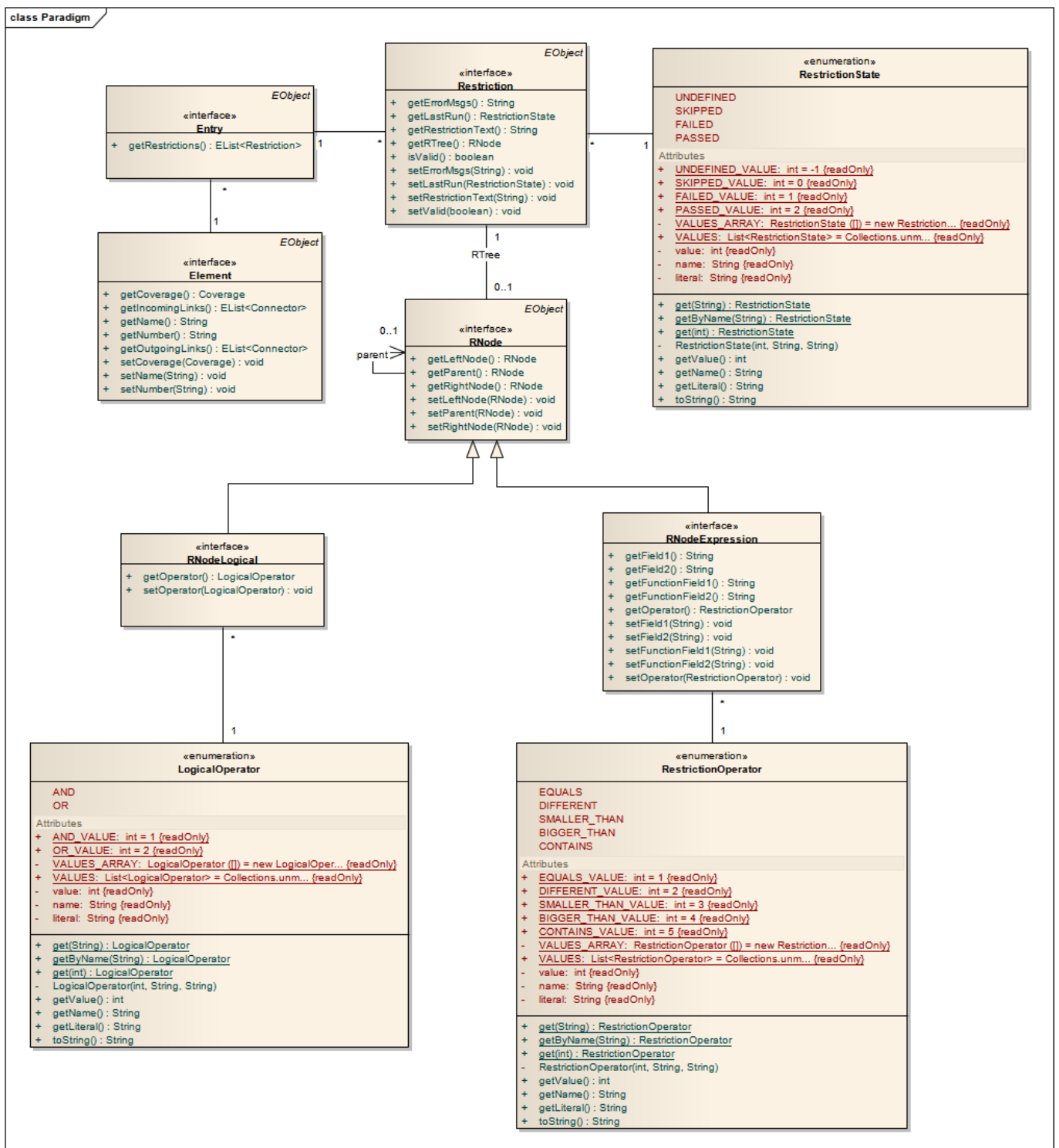


Figure 3.6: Simplified diagram of the constraints' support that was added to PARADIGM-ME.

To simply allow the insertion of constraint expressions alone is not enough however – as seen before, they must also be verified for correct syntax, as well as whether the TGConf fields referred

to actually exist. To this purpose, an extra package was added to the *paradigm* one, *utils.custom*, which contains utilities to parse a RTree and analyse a restriction on its syntax.

The *utils.custom* has then three classes:

generalUtils providing general accessibility methods to the data contained in a PARADIGM model;

restrictionUtils which contain functions to parse constraints, build RTrees, and analyse constraint syntax;

InvalidRSyntaxException constituting the exception thrown by *RestrictionUtils* if proper syntax is not met, or if a TGConf field used in the constraint does not exist in the model.

3.4 Script Analysis

Another functionality made available by the developed PARADIGM-COV tool is script coverage analysis, which allows a user to check to what extent a custom-made script is able to exercise an existing PARADIGM model's UITPs.

The format for these scripts derives from the existing tool itself, since PARADIGM-ME creates test scripts automatically when generating test cases from the model. In fact, it is with basis on those very scripts that the test cases are ultimately executed on the SUT. This allows the tester to override the automatically generated tests and provide PARADIGM-ME with his own test specifications if he so wishes.

At the moment, the script coverage analysis is focused on comparing a custom test script with an existing model for test goal coverage, although it should later be expanded to include constraints' analysis as well. The scripts themselves follow the XML specification identical to the format already used by PARADIGM-ME itself when generating its test scripts. An example of the format used is presented in Listing 3.1.

```
1 <?xml version="1.0" encoding="UTF-8" standalone="no"?><Script><Path value=
  "[2.0, 2.3, 2.2, 2.1]"><Init/><Call check="StayOnSamePage" flag="true"
  number="2.3" result=""><Field name="call_2.3"/></Call><Login check="
  StayOnSamePage" flag="false" message="" number="2.2" validity="Valid"><
  Value field="password_2.2" value="pass"/><Value field="username_2.2"
  value="johndoe@something.com"/></Login></Path></Script>
```

Listing 3.1: Example snippet demonstrating the format of a XML test script.

A script contains then a sequence of TGConfs, sorted by order of execution. Each TGConf is, as addressed before, a *Test Goal* Configured, and hence necessarily associated with a given UITP. Given so, each TGConf in the script is associated with its respective UITP type and ID

number. These 'attributes' allow the identification of which TGConfs belong to the same UITP, and therefore enables to evaluate whether that UITP is covered regarding its test goals.

Likewise, the model is made up several UITP elements, each of them containing in turn several TGConfs. The script coverage analysis relies on comparing the set of UITPs present in the test script, with the set of UITPs present in the model.

A test script is considered to cover a model's Test Goals if two conditions are upheld:

- (a) the set of UITP expressed in the script ($\{UITP_S\}$) contains the set of UITP present in the model ($\{UITP_M\}$):

$$\{UITP_S\} \supseteq \{UITP_M\}; \quad (3.6)$$

- (b) the test script contains configurations (TGConf) for all Test Goals associated with all the UITP within the model. For example, if there is only one TGConf for a Login UITP, its goal coverage will be only of 0.5, since this UITP has two Test Goals associated with it (G_{LV} , G_{LINV}). In other words, all UITP in the test script must achieve full Test Goal coverage:

$$gCov(UITP_M) = 1.0. \quad (3.7)$$

The results of this test goal analysis when comparing a test script to a model are presented using the same approach taken by Test Goal Analysis in Model Coverage (seen in section 3.3.2). More precisely, the elements in the model are coloured in green, yellow, and red backgrounds. A model element is covered in green if the test script contained TGConfs that encompassed all that element's test goals (thereby meeting the UITP coverage criteria presented back in Table 3.2); coloured in yellow if the script contains TGConfs for that element, but not enough to exercise all its test goals; and finally the model element is coloured in red whenever the script has no tests defined for that UITP element whatsoever.

This approach is used because uploading tests from a custom-made test script effectively overrides the automatically generated tests on execution time. In a way, one could say the model's test configurations (TGConfs) are discarded in favour of those provided by the script, and the coverage analysis regarding test scripts reflects precisely that fact.

Implementation

In what concerns the implementation of this feature, this one functions much as the model's Test Goal Analysis. While the previous strategy depended on *CoverageAnalyser*, this type of coverage evaluation is done by the class *ScriptAnalyser* instead, which performs the mapping between the script's UITPs and the ones belonging to the model. Regardless, the analysis on whether an UITP fulfils its coverage criteria is done by the same methods that were presented previously, namely, by the *CoverageUtils* class.

3.5 Execution Analysis

Another capability of PARADIGM-COV concerns the detection of discrepancies between the tests cases generated from the model and the ones actually executed on the system. This dynamic analysis allows the tester to differentiate between a test that fails (i.e., the checks defined are False) from one that was not executed (for example, due to the SUT crashing unexpectedly, or a target webpage element not being found).

Additionally, it matters to mark the distinction between model coverage analysis and execution coverage analysis. While the model coverage analysis evaluates if test cases cover completely the model, execution coverage analysis evaluates if the test cases were completely executed on the SUT. In other words, while model coverage asks "*is the model covered by the test strategies defined?*", execution coverage attempts to answer the question "*are all the defined test configurations executed over the SUT?*". So, it is possible to reach full execution coverage even when test cases do not reach full model coverage, and both metrics are important in order to fully understand to what extent the testing process exercises the SUT.

The execution report built by PARADIGM-COV displays several metrics pertaining to both test check results (*cr*) and test execution result (*er*) coverage data. Check results indicate whether a check passed or failed, i.e., whether if the executed test returned the expected outcome regarding the SUT's behaviour. Execution results however simply report whether a given test case was executed or not. A non-executed test case indicates that PARADIGM-ME could not complete the checks associated with a given test, hence, there are no assurances on whether the system is behaving as expected.

The report itself is divided into three distinct data sections: element statistics, test case statistics, and test trace.

3.5.1 Element Statistics

Element statistics is the report section that presents the test execution results organized by behaviour element. In other words, for every UITP executed, it is displayed the percentage TGConfs whose check passed (check result, or *cr*), and the percentage of TGConfs that were executed (execution result, or *er*).

The check result coverage can thus be formalized as

$$\frac{\sum_{i=1}^N (cr(tgc) == passed)}{N}, \quad (3.8)$$

where *cr* is the check result for a particular *tgc* : *TGConf*, and *N* is the total number of configurations (TGConf) within all TGs specified for a UITP.

Likewise, the percentage of executed TGConf for each UITP is given by:

$$\frac{\sum_{i=1}^N(er(tgc) == executed))}{N}, \quad (3.9)$$

where er is the execution result of the $tgc : TGConf$, and being N still the total number of configurations associated with the TGs of the UITP in question.

3.5.2 Test Case Statistics

Test case statistics follows the same approach as element statistics, but applied to test cases. A test case is a sequence of steps, and each step is a configuration (TGConf) for a specific UITP. So, the coverage tool presents two measurements for each test case: the percentage of the overall configurations within a test case that were executed; and the percentage of the corresponding checks that passed/failed.

Being N the total number of $tgc : TGConf$ s within a test case tc , one can define the percentage of passed tests (cr) within a test case as

$$\frac{\sum_{i=1}^N(cr(tgc) = passed))}{N}, \quad (3.10)$$

Similarly, the percentage of steps with a test case that were executed (er) is

$$\frac{\sum_{i=1}^N(er(tgc) = executed))}{N}. \quad (3.11)$$

Ultimately, the aim with displaying information under element and path statistics is to present the tester with a quicker overview in regards to where lay the errors found.

3.5.3 Test Trace

Test trace section displays results obtained by each TGConf by order of execution. It signals the TGConfs that were not executed and, for every TGConf executed, the report presents information whether its check passed or failed. In the latter case, it also provides a reason for failure (e.g., a UI element where an action should occur could not be found within the webpage of the SUT).

Implementation

In order to keep track of the tests being executed by PARADIGM-ME, a singleton class dubbed *ExecutionState* was created. This one was plugged into the already existing test execution mechanisms, registering all tests meant to be executed, their result, and listening for any error exceptions that might occur. The process is illustrated in the simplified sequence diagram shown at Fig. 3.7.

In the existing PARADIGM-ME structure, the test execution process began by reading all the test steps from the generated script, action that occurred in *ExecuteTests*. Each test step would

Test Coverage Tool

then be executed according to the test strategy of the UITP it belonged to. The actions of the UITP are in turn realized by a set of classes implementing the actions A associated with each UITP (in Fig. 3.7, these were all grouped together under *TestStrategy* for the sake of simplicity).

Test Coverage Tool

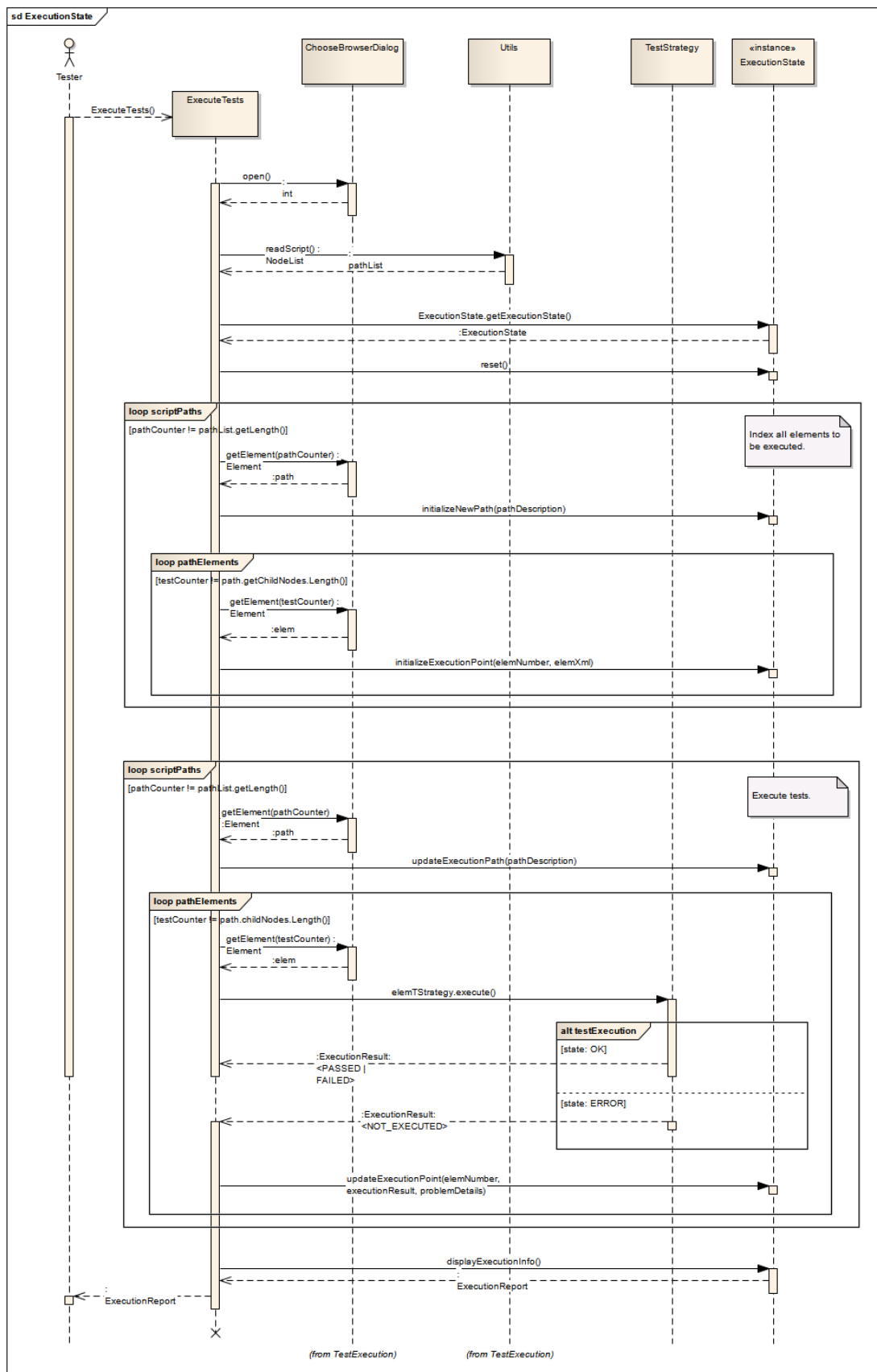


Figure 3.7: Simplified sequence diagram illustrating the test execution tracking process.

Test Coverage Tool

The first step in order to implement dynamic coverage analysis was to make sure we could obtain all the necessary data from the test cases being executed. Since PARADIGM-ME stores all the data on the generated tests directly into the test script, the contents of the test script dictate the information that is available at test execution time.

In the initial version of the test script, only the data strictly necessary to the test execution process was recorded, i.e., the generated script contained solely the array of test steps to be executed. However, in order to later report feedback on test execution, we needed to be able to collect additional information - for example, it was necessary to distinguish the test case to which a test step (TGConf) belonged, as well as being able to identify its respective test path. As such, the test script being created was upgraded to a new format that would include this information, and its generation process was adapted accordingly. The difference between both formats is displayed in Listings 3.2 and 3.3, where in the second version it is possible to gather information on test paths ([1,2,3]), and the element `Init` marks the beginning of each individual test case (in this case, [1,2(Login_Valid),3] and [1,2(Login_Invalid),3]).

```
1 <?xml version="1.0" encoding="UTF-8" standalone="no"?>
2 <Script>
3   <Login check="StayOnSamePage" flag="true" message="" number="2" validity="Invalid
4     "><Value field="username_2" value="not_signed_in"/>
5     <Value field="password_2" value="pass"/></Login>
6   <Login check="changePage" flag="true" message="" number="2" validity="Valid"><
7     Value field="username_2" value="visitor"/>
8     <Value field="password_2" value="visitor"/></Login>
9 </Script>
```

Listing 3.2: Example of previous test script format.

```
1 <?xml version="1.0" encoding="UTF-8" standalone="no"?>
2 <Script>
3   <Path value="[1, 2, 3]">
4     <Init/>
5     <Login check="StayOnSamePage" flag="true" message="" number="2" validity="
6       Invalid"><Value field="username_2" value="not_signed_in"/>
7       <Value field="password_2" value="pass"/></Login>
8     <Init/>
9     <Login check="changePage" flag="true" message="" number="2" validity="Valid"><
10      Value field="username_2" value="visitor"/>
11      <Value field="password_2" value="visitor"/></Login>
12 </Path>
13 </Script>
```

Listing 3.3: Example of the updated test script format.

Having access to all the needed data during the test execution process, the coverage analysis

mechanisms were then added to *ExecuteTests*. To obtain feedback on the execution of each single test step, we created the attribute *ExecutionResult*, which can take one of the following values: *passed*, *failed*, or *not executed*. By default, the execution of a TGConf will return the result of its test checks (*true* : *passed* or *false* : *failed*). The addition of an error handling mechanism guarantees that PARADIGM-ME is able to recover from any errors occurred during test execution.

Regarding error handling in particular, this one is done by catching and logging any exceptions occurred during test execution. PARADIGM-ME executes its test cases by using Selenium¹ and Sikuli Script². As such, the exceptions caught can be roughly of three types: a Selenium or a Sikuli Script exception, which are thrown when each of these frameworks is not able to perform a test action, or a PARADIGM-ME exception, usually originating due to defective mapping data, or when the execution of a test action hangs and results in a *timeout*. Each time an exception is caught, this event is relayed to the *TestExecutionClass*, along with details about its localization and cause of failure.

As a result of this addition, an error occurrence no longer implies that the whole testing process is interrupted, but rather, the error is logged and the next test case is executed. As for the test step where the error occurred, this one is marked as *not executed*, and recorded in the Execution Analysis report with further information on what might have provoked said error.

All this processing is done through *ExecutionState*, which contains the following three inner structures (Fig. 3.8):

ExecutionPoint Represents a single test step or TGConf; it contains data on the TGConf being executed, its final result, as well as a description of problems that might have prevented its execution.

ExecutionPath Stands for a single test case and, in that light, is considered as a collection of *execution points* (TGConfs). Performs the required calculations for obtaining the Test Case Statistics regarding *check results* and *execution results*.

ExecutionState Represents the possible results of a test execution: *passed*, *failed*, and *not_executed*.

¹<http://docs.seleniumhq.org/>

²<http://www.sikuli.org/>

Test Coverage Tool

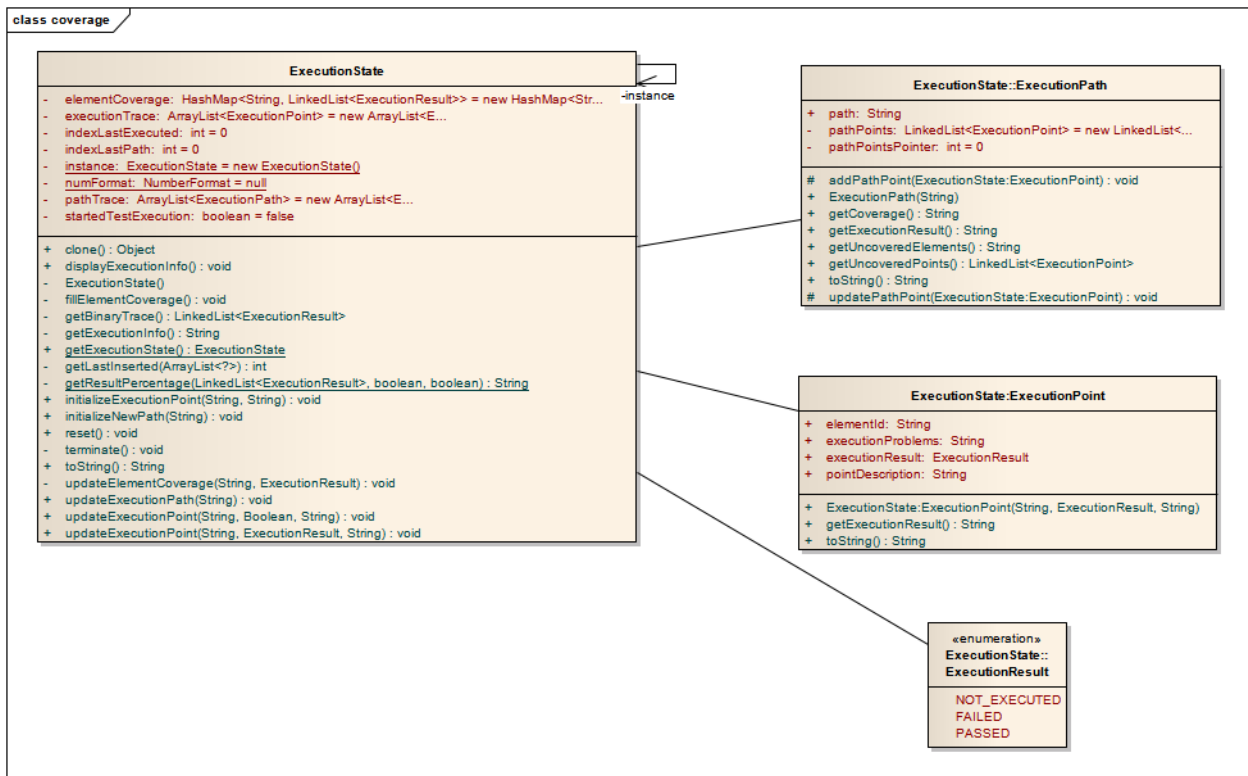


Figure 3.8: Structure of the *ExecutionState* class.

3.6 Code Coverage

It is not always true that the code of a SUT is available but, when such is the case, code coverage may be an additional metric useful to evaluate the quality of test cases. In simple terms, code coverage consists in tracking which parts of the code are executed while exercising the software application under test (SUT).

Indeed, while it is possible to reach full model coverage, nothing assures that full code coverage is achieved, for instance, because the model does not describe fully the SUT. Code coverage information is useful to inspect the parts of the code that were not exercised and design additional test cases in order to cover them.

In what concerns the coverage criteria used, the PARADIGM-COV tool provides support for two different approaches: line coverage criteria, and block coverage criteria. Unlike what happens with the line coverage information, with block coverage data the user will not know the exact line source of the problem when the application fails in the middle of a block. The obvious advantage with block coverage is that probe insertion and execution is faster than with its counterpart, given that the number of probes inserted is vastly inferior. In any of these cases, the code coverage ratio for a file is calculated using the following formula:

$$\frac{\textit{number of probes executed}}{\textit{total number of probes}}. \quad (3.12)$$

PARADIGM-COV creates a working copy of the SUT for instrumentation. At the end of the code coverage analysis, the tool produces a list with the SUT files that were instrumented, and the degree to which they were exercised during the testing process. Still adhering to the same colour conventions, each file entry indicates in green, yellow or red if said file was exercised, partially exercised, or not exercised respectively. Additionally, we created a custom editor to provide a more in-depth analysis on the coverage of each file in particular, indicating exactly which code lines were covered. To achieve this, instrumented lines are marked in either green or red, being all others are left blank. By opting not to colour lines without assigned coverage information, we assure that the tester has always precise information on what has effectively been executed. In the event of an application failure (in the middle of a block for instance), non-executed lines will never be displayed to the tester in green, avoiding this way to provide him with erroneous information.

Implementation

All the main features related with the instrumentation process and code coverage analysis in general are implemented in the coverage plugin's *instrumentation* package. A diagram of its class structure can be seen in Fig. 3.9. Following, we give a brief exposition on the functions of each of its classes.

FileLineStructure Structure containing the final code coverage information for a single file. Alongside with the file's absolute path, it contains all its line numbers registered as covered and uncovered. This class is used mainly as a convenient way to pass the coverage information between different methods, since it discards all the information not directly relevant to the code coverage report.

Instrumenter Main engine behind the instrumentation process. It indexes the SUT's relevant source files, and manages the probe placement process, inserting probes at the relevant lines or code blocks accordingly; also responsible for retrieving the raw data gathered by the probes during execution, and process those results to obtain the final coverage data to present in the report.

InstrumentAction Abstract handler that detects the event fired when the user selects to perform coverage analysis, and ties all the instrumentation dialogues with their actions. Its primary functions include the launching of the instrumentation process, and the calls to the appropriate editors when time comes to present the final report.

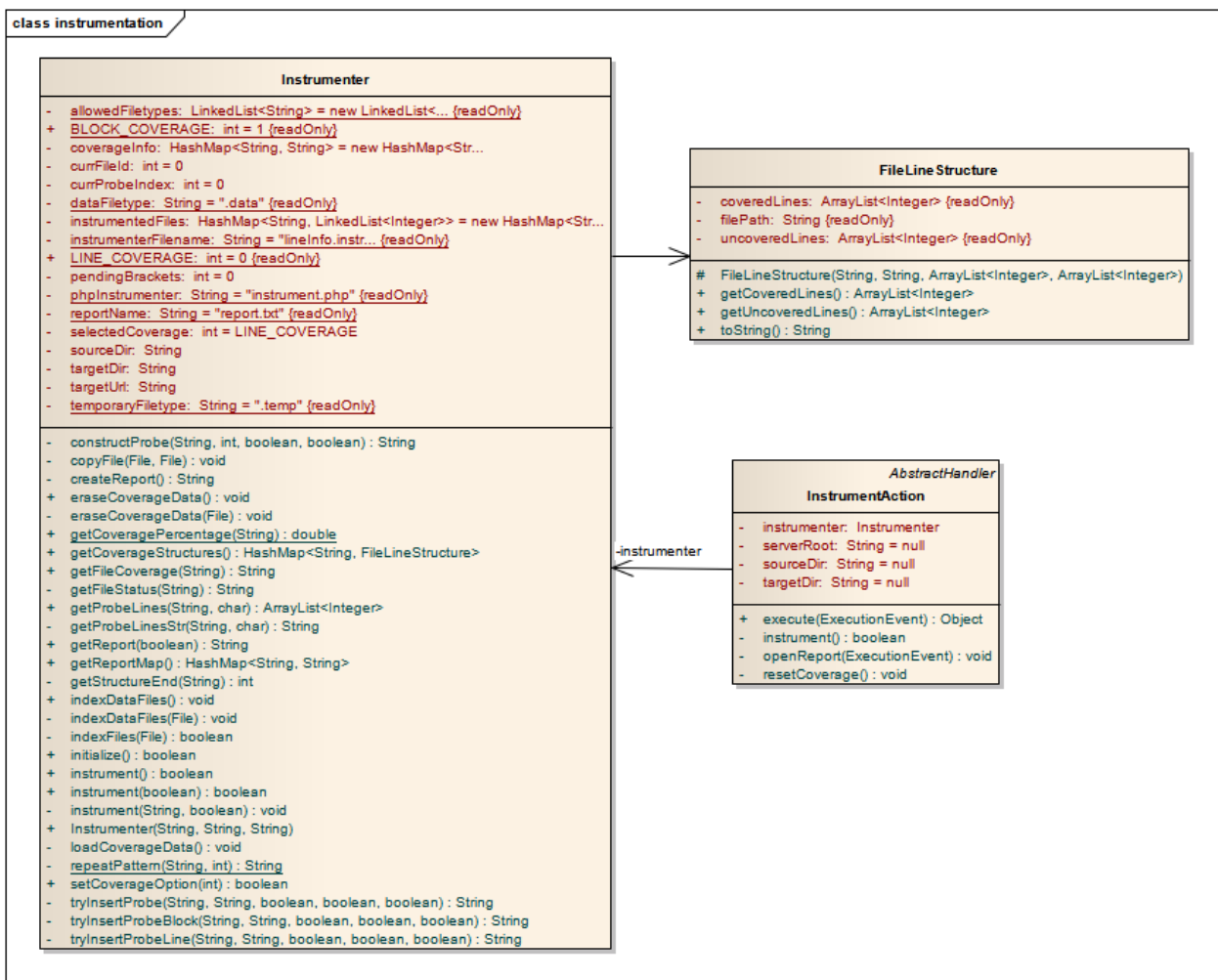


Figure 3.9: Class diagram of the main classes responsible for the instrumentation process.

3.6.0.1 Probe Placement

The first phase in the probe placement process is to index the target file, copying it to the directory that will hold the instrumented version of the application under test. Only then are the probes placed, be it in every code containing line (line coverage), or in the first line of every block (block coverage). This probe insertion is performed by the *instrument* and *tryInsertProbe* functions of the *Instrumenter* class. The probe locations themselves are found by doing a line-by-line analysis on each relevant source file, and matching it with a set of regular expressions' patterns, which were created specifically with the PHP language syntax in mind.

At the end of this instrumentation process, the total number of probes inserted into each file is recorded. Such number is then used to insert a block of PHP code at the beginning of the instrumented file, making a call to the PHP module that will allow probe data to be gathered during execution. This PHP module, *FileInstrumenter*, was created to handle everything concerning the probe execution process itself.

Test Coverage Tool

An excerpt of a instrumented file, as well as the original source from which it was constructed, is presented in Listings 3.4 and 3.5.

```
1 <?php
2
3 $a = 2;
4 $b = 4;
5 \\...
6 if ($a == 2){doSomething();}
7 else
8     doNothing();
9
10 \\...
11 }
```

Listing 3.4: Excerpt of a source file, before instrumentation.

```
1 <?php include_once ($_SERVER["DOCUMENT_ROOT"] . "/eclipse_instrumentation/target/
    instrument.php");
2 $file2 = new FileInstrumenter();
3 $file2->initialize("teste2.php.data", 78); ?>
4
5 <?php
6 $file2->runProbe(0);
7
8 $a = 2; $file2->runProbe(1);
9 $b = 4; $file2->runProbe(2);
10 \\...
11 if ($a == 2){$file2->runProbe(12); doSomething();}
12 else {$file2->runProbe(13);
13     doNothing(); $file2->runProbe(14);}
14
15 \\...
16 }
```

Listing 3.5: Excerpt of an instrumented file.

3.6.0.2 Data Gathering

When executing the instrumented SUT, the PHP *FileInstrumenter* creates a *.data* file for each file instrumented. This data file contains the raw data for that file's coverage, stored in a binary format. For a concrete example, let us take an actual instrumented file, *listarobras.php*. After running a series of tests on the system, we could find *listarobras.php.data* with the following content:

```
111000001
```

Test Coverage Tool

This is the raw data from the probes' execution on that file. Each binary number represents a probe on the file: an executed probe is represented with a 1, and a non-executed one with a 0. Hence, from these contents we could conclude that *listarobras.php* had 9 probes placed, of which only 4 were executed (corresponding to approximately 44% coverage).

Yet, this does not tell us which lines that were actually executed. This is the reason why, during file indexing, *lineInfo.instrumenter* is created. This file registers all instrumented files, and, for each, it lists which lines contain probes. By taking the excerpt relevant to the file in question, we can infer the numbers of the executed lines on *listarobras.php*:

```
(...)  
listarobras.php:[0, 1, 2, 4, 5, 7, 8, 9, 10, 12, 13, 14, 15, 16, 18, 19]  
(...)
```

This crossing of information is realized on the Java *Instrumenter* class when creating the code coverage report.

Still concerning these auxiliary data files, it is relevant to note that they are persistent, i.e., they are not deleted automatically at the end of the code coverage analysis, unless the tester specifically chooses to do so. There are several reasons why we opted for this solution. Firstly, this fact allows the tester to consult the code coverage report relative to the testing session without having to run through the test execution process once more. A second reason is that, when the data files are present, a new test execution process will not override them - new coverage data is simply added to the existing one. If it were not for this, the tester would be forced to execute all the intended tests at a time, which may not be practical if the test suites themselves are extensive. By merging the probes' data over executions, the tester may divide a particularly large test script into several smaller ones (executing only a test path at a time, for example), and obtain the same results.

Chapter 4

Case Study

After the implementation of the PARADIGM-COV tool, we conducted a case study with the intent to:

- Showcase the basic functionalities of the tool;
- Illustrate how the tool can be used in a real-world scenario;
- Analyse the results obtained for assess the value of tool usage.

This case study is based on a real, crudely tested, website, for which the source code was available. The original website was populated with realistic data for this case study, and next the testing goals for this website were described by a model written in PARADIGM with corresponding configurations. PARADIGM-COV was then used to help the configuration activity and to assess the quality of the generated test cases.

4.1 Sample Website

The website used for this case study aims to serve as an artwork museum management system. As such, it allows to register new users, authenticate existing ones, edit the artwork data in the database according to permission rules, manage artwork donations made by users, consult and search for artworks, as well as sort the results obtained from said search. A screenshot of the site is shown in Fig. 4.1.

4.2 PARADIGM-ME Model

The first step to test this website is to represent the system through a PARADIGM model, which describes the tests to perform over the SUT. In this case, the aim is to test the registration functionality, the authentication, the search mechanism, and a subset of the administrative functions accessible only for logged-in users, namely, the artwork donation management system.

Case Study

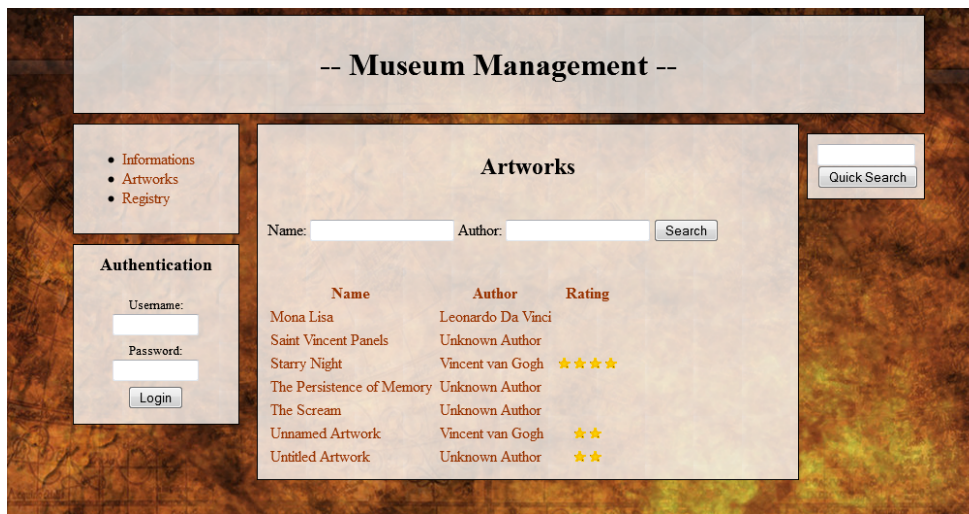


Figure 4.1: Screenshot illustrating the main page of the Museum Management website.

The model built (Fig. 4.2) contains one main Group, *BaseMenu*, whose inner elements can be accessed in any order. Namely, these functionalities are *RegistryCall*, *Search*, and *Artworks*, and represent links available through the website's navigation menu.

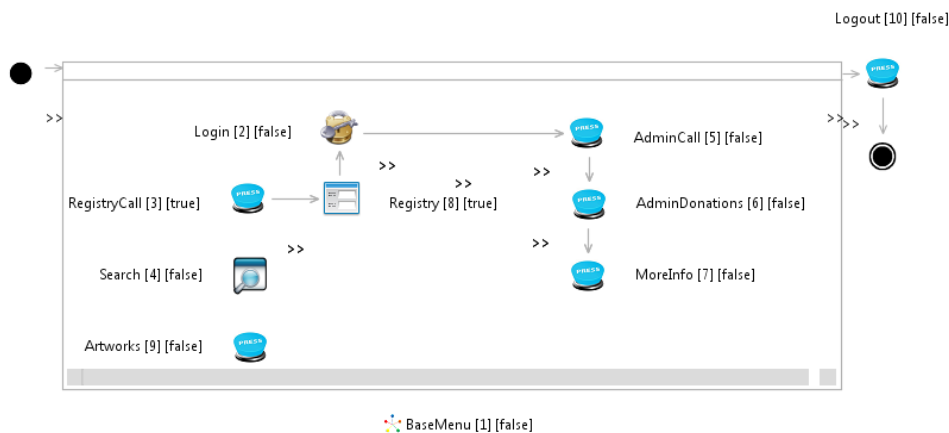


Figure 4.2: Example of the PARADIGM-ME model developed for the first 'level' of the website.

A Call element such as these is intended to access the respective link from the main menu, and subsequently subsequently test the resulting page.

Regarding element sequencing in particular, when two elements, A and B, are linked by a connector from A to B, this signifies that the configurations defined for element B can only be executed after A. In other words, the sequence connection ensures that an element can only be tested after the element that precedes it. Such is the case with the connection between *Login* (2) and *AdminCall* (5) for instance.

Case Study

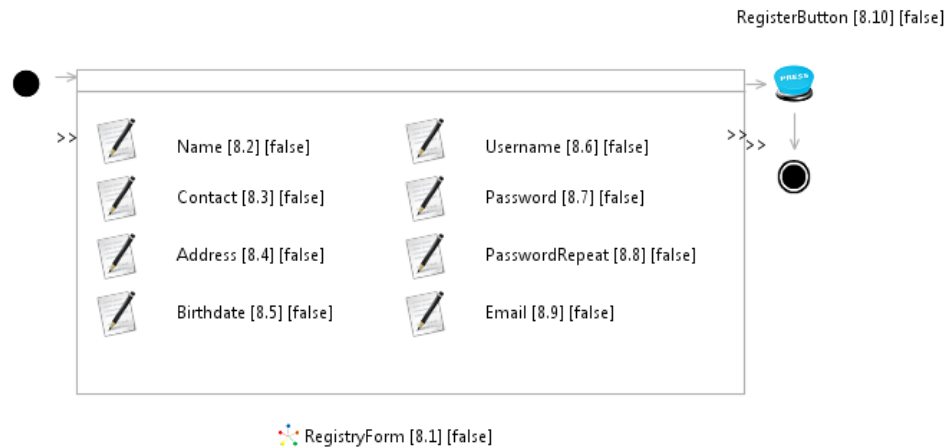


Figure 4.3: PARADIGM-ME's model created inside the form *Registry* (8).

PARADIGM also accounts for the possibility of optional elements. In this model, the *RegistrationCall* (3) and *Registry* (8) are marked as optional, since there will be test paths that go from *Init* directly to *Login* (2)). The reason for this choice is that the corresponding registration functionality is only available for non-authenticated users (i.e., registration is not available for authenticated user after the *Login* (2)).

The *Registry* Form in particular represents the set of inputs required from the user in order to create a new account. As such, inside the *Registry* Form (8) (Fig. 4.3) there are several Input Test Patterns to test the corresponding input fields of the registration page, for both valid and invalid input data. Once again, these tests can be performed by any order. As for *RegisterButton* (8.10), this element simply corresponds to the final submit button of said webpage form.

To conclude the PARADIGM model, a final call to the *Logout* (2) button on the first level makes sure that the webpage is always returned to its initial state upon completion of a test path.

As for the configurations defined for the UITP within the model, these can be seen in Tables 4.1 and 4.2. For illustration purposes we left *Artworks* and most of the elements within the *Registry* without configurations.

Regarding the definition of preconditions, all preconditions are true, except for the configurations of elements that test functionality which is accessible only to logged-in users, these being *AdminCall* (1.4), *AdminDonation* (1.5), *MoreInfo* (1.6), and *Logout* (2). As such, each of the mentioned elements contain a constraint expression, which consists in:

```
Login1_1.username == 'admin' AND  
Login1_1.password == 'admin'
```

4.3 Test Cases

The execution of the model above described automatically generates the following test paths:

Case Study

Table 4.1: Configurations performed on the model's first level elements.

Id	Element	Input Values (V)	Test Goals	Checks
3	<i>RegistryCall</i>	not applicable	<i>G_Call</i>	Text present in page = "Register Account"
5	<i>AdminCall</i>			Text present in page = "Administration Module"
6	<i>Donations</i>			Text present in page = "Donations"
7	<i>MoreInfo</i>			Text present in page = "Name: "
9	<i>Artworks</i>		none	none
4	<i>Login</i>	username = "admin" password = "admin"	<i>G_LV</i>	Text present in page = "Logout"
		username = "abc" password = "abc"	<i>G_LINV</i>	Stay on the same page
4	<i>Search</i>	search = "starry night"	<i>G_Found</i>	Number of results = 1

Table 4.2: Configurations performed on the model's second level, the form *Registry*.

Id	Element	Input Values (V)	Test Goals	Checks
8.2	<i>Name</i>	input = "John Doe"	<i>G_IV</i>	not applicable
8.3	<i>Contact</i>	none	none	
8.4	<i>Address</i>			
8.5	<i>Birthdate</i>			
8.6	<i>Username</i>	input = "myusername"	<i>G_IV</i>	
8.7	<i>Password</i>	input = "mypass"		
8.8	<i>PasswordRepeat</i>	none	none	
8.9	<i>Email</i>	input = "johndoe@unknownusers.com"	<i>G_IV</i>	
8.10	<i>RegisterButton</i>	not applicable	<i>G_Call</i>	

```
{ [Init; 9, 4, 2, 5, 6, 7, 10, End],
  [Init; 9, 4, 3,
8.0, 8.1, 8.6, 8.7, 8.8, 8.9, 8.2, 8.3, 8.4, 8.5, 8.10, 8.11,
2, 5, 6, 7, 10, End] } ,
```

where the elements depicted by the number $8.n$, are the ones that correspond to the hierarchical abstraction level detailing Form *Registry* (8), illustrated in Fig. 4.3.

Once calculated the paths, these are expanded according to the TGConfs defined for each UITP. For instance, considering that *Login* (2) has two configurations, *Login_Valid* and *Login_Invalid* (Table 4.1), the first path would be transformed into:

```
[Init; 9, 4, 2(Login_Valid), 5, 6, 7, 10, End],
[Init; 9, 4, 2(Login_Invalid), 5, 6, 7, 10, End]
```

The same strategy is followed the the remaining path. However, since elements 5, 6, 7 and 10 have a precondition demanding a valid login, the second test case will be instead $[Init; 9, 4, 2(Login_Invalid), End]$. This explains why preconditions are useful as a way to limit the final test cases to perform.

The same expanding process is performed for the other UITP within such path in order to obtain the final test cases.

4.4 Coverage Tool Output

After executing the generated tests, the coverage tool developed was then used to evaluate their coverage. Figures 4.4 and 4.5 show the coverage information obtained by *Test Goal Analysis* and *Constraints' Analysis*. From the results obtained, one can easily see which ones are not exercising all their test goals. Elements marked in red background mean that no test goal was configured for the UITP, while elements in yellow mean that configurations were defined for only some of the test goals. Elements marked in green indicate that configurations were defined for every test goal of the respective UITP.

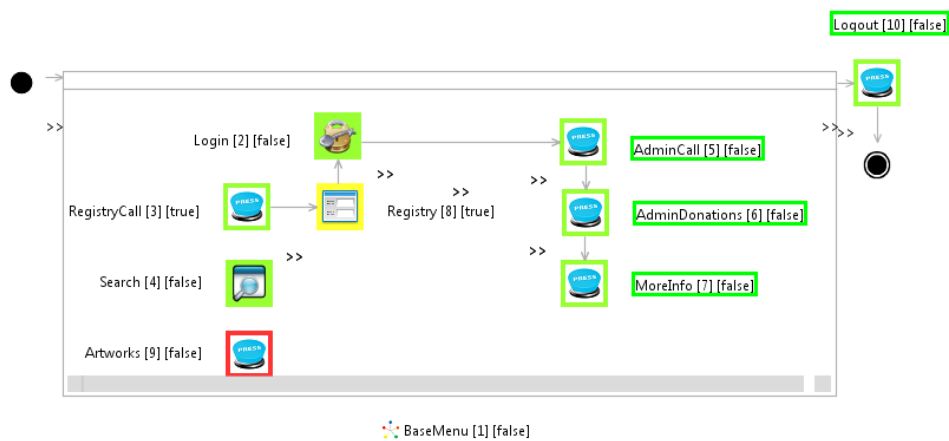


Figure 4.4: Result displayed by the model coverage analysis feature for this website’s model.

Concerning restrictions, their existence is marked by displaying a borderline on the respective element’s tag, as can be seen with elements number 5, 6, 7, and 10. The fact that this border is presented in green colour assures the tester that all test configurations are to be executed at least once, since there exists at least a state where the given restrictions are valid. One can inclusive confirm this fact by taking a look at the 'last run' field one of the element’s restriction table (Fig. 4.6).

This signalization allows the tester to quickly spot the problem areas and, if he so desires, to define additional test configurations until full model coverage is reached. In this case, the 'problematic' elements lie solely on the *Artworks* element, which has no configurations whatsoever, and inside the *Registry* Form, which does not guarantee that the goals for its containing elements are

Case Study

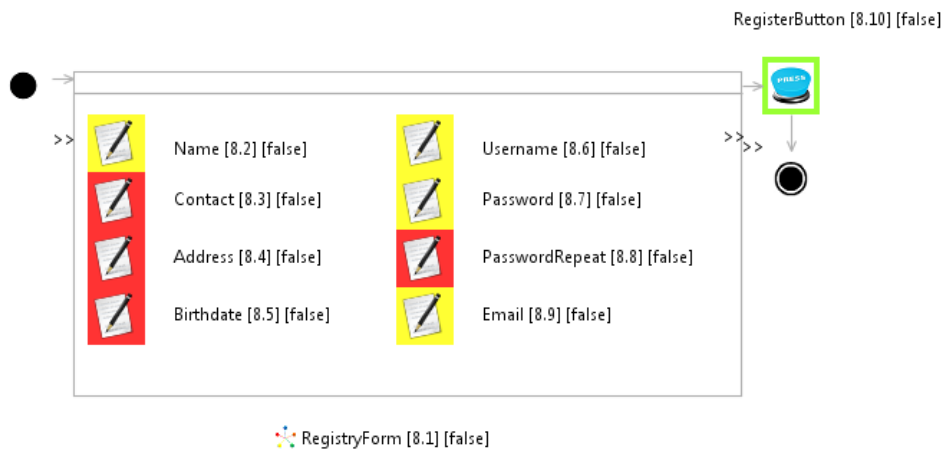


Figure 4.5: Result of model coverage analysis on the second level of the model, *Registry Form* (8).

The screenshot shows a window titled "Restriction Configurations" with a table containing one row of data. The table has four columns: Restriction, Valid, Problems, and Last Run. The data row shows the restriction "Login.username == 'admin' AND Login.password == 'admin'", which is valid (true), has no problems, and was last run successfully (passed).

Restriction	Valid	Problems	Last Run
Login.username == "admin" AND Login.password == "admin"	true		passed

Figure 4.6: Restriction table for a TGConf of the element *AdminCall* (5).

fully exercised. Indeed, taking a second look at the configuration tables (Tables 4.1 and 4.2), one can confirm that most of the elements on *Registry Form* possess insufficient tests, since they have only configurations for valid inputs, overlooking the invalid ones.

PARADIGM-COV also performed execution analysis. For illustration purposes, we forced a crash near the end of the testing process, to showcase non-executed tests as well. Once testing is complete, a tabbed view displaying the results is presented, as shown in Fig. 4.7. As can be seen, the report informs that 86% of all TGConfs passed and 92.86% of all TGConfs were executed.

Glimpsing at the first section in particular, statistics about element 10 and element 8.10 are noteworthy. None of these elements reached the ideal result (100% passed, 100% executed), yet, the reason for that is radically different.

In case of element 10, only 50% of the TGConfs were executed and passed. The goal of the tester here is therefore to try to understand why checks could not be fully exercised, which can happen because the associated web elements are not available. In this particular case, the failure to execute happens because we forced a crash on the SUT. Test case coverage information may

Case Study

```
Execution Report

Overall Results: 85,71% passed (92,86% executed)
* Elem. 3           : 100% passed (100% executed)
* Elem. 8.2        : 100% passed (100% executed)
* Elem. 2          : 100% passed (100% executed)
* Elem. 10         : 50% passed (50% executed)
* Elem. 7          : 50% passed (50% executed)
* Elem. 6          : 100% passed (100% executed)
* Elem. 5          : 100% passed (100% executed)
* Elem. 4          : 100% passed (100% executed)
* Elem. 8.10       : 0% passed (100% executed)
* Elem. 8.7        : 100% passed (100% executed)
* Elem. 8.6        : 100% passed (100% executed)
* Elem. 8.9        : 100% passed (100% executed)

Test Case Coverage [<testCase>: <checkResult> | <executionResult>]:
* [0, 1, 9, 4, 2, 11]: 100% passed | 100% executed
* [0, 1, 9, 4, 2, 5, 6, 7, 10, 11]: 100% passed | 100% executed
* [0, 1, 9, 4, 3, 8, 8.0, 8.1, 8.6, 8.7, 8.8, 8.9, 8.2, 8.3, 8.4, 8.5
* [0, 1, 9, 4, 3, 8, 8.0, 8.1, 8.6, 8.7, 8.8, 8.9, 8.2, 8.3, 8.4, 8.5
=> uncovered at elements 7, 10

Test Trace [<executionOrder>) <elementID>: <testResult>]:
(executed tests based on generated script)
1) 4           : passed
2) 2           : passed
3) 4           : passed
4) 2           : passed
```

Figure 4.7: Part of the execution report with check success and test execution measurements.

also be useful in these cases to help identify the problem, by allowing paths not fully covered to be inspected more deeply (in this case, test cases 3 and 4):

```
Test Case Coverage [<testCase>: <checkResult>|<executionResult>]:
* [0, (...), 11]: 100% passed | 100% executed
* [0, (...), 11]: 100% passed | 100% executed
* [0, (...), 11]: 87,5% passed | 100% executed
* [0, (...), 11]: 75% passed | 83,33% executed
```

Regarding element 8.10, all configurations (TGConf) associated with it were actually executed, but none passed (i.e., the target system did not behave as expected). One of two things could have happened: either the SUT is wrong, or the model itself is. By checking once more with the diagram, one can verify that element 8.10 corresponds to the submit button of the *Registry* form. This is a case where Test Trace section may provide additional helpful information regarding this failure, namely, the details on the input data provided for this test, as well as its respective check:

```
Test Trace [<executionOrder>) <elementID>: <testResult>]:
(...)
15) 8.10 : failed
```

Case Study

```
=> at <Call check="changePage"
      flag="false" number="8.10" result="">
```

(...)

In this particular situation, there is a check which verifies if the webpage changes when submitting the data to server. Yet, this check failure indicates that the SUT remains on the same page instead, indicating a possible bug in the SUT.

Regarding element 8.10, all configurations (TGConf) associated were actually executed but none passed (i.e., did not behave as expected). One of two things can happen, either the SUT or the model is wrong. Element 8.10 corresponds to the submit button of the *Registry* form. In this case *Test Trace* section provides may provide additional helpful information regarding this failure, namely, the input data provided for this test, as well as its respective check. In this particular case, there is a check which verifies if the webpage changes when submitting the data to server. Yet, this check failure indicates that the SUT remains on the same page instead, indicating a possible bug in the SUT.

Having previously instrumented the target systems, all tests were run on the instrumented version. Hence, the result of code coverage analysis are also available, and can be seen at Fig. 4.8.

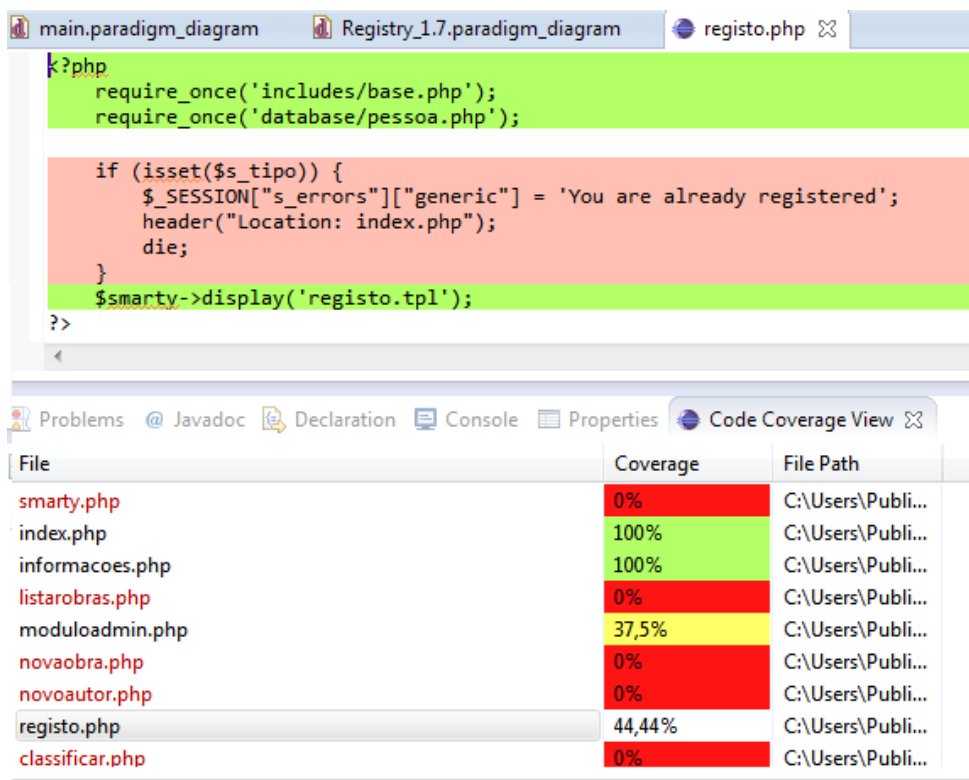


Figure 4.8: Code coverage for a single file and overall results for the list of analyzed files.

Fig. 7 shows the file invoked when a user attempts to access the registry page through the main menu. In the model, it is described by the *RegistryCall* element (3). Besides element 3 being fully

Case Study

covered, test cases did not reach full coverage of the code (as shown by the block signalled in red). The uncovered block pertains to a situation that occurs if a logged-in user attempts to access this page. This can happen when an authenticated user tries to access the corresponding functionality directly by typing the URL. Since this is not included in the model, the code is not covered by the generated test cases. However, by analysing the code coverage results, the tester can perform additional tests in order to reach this specific block of code.

Case Study

Chapter 5

Conclusions and Future Work

This work concerned the development of a Coverage Analysis tool (PARADIGM-COV) that produces model and code coverage information in the context of a model-based testing project called PBGT.

The main advantage of the PARADIGM-COV tool is its capability to perform a broad coverage analysis over different aspects of the SUT, presenting information about test requirements coverage (Test Goal Analysis), model coverage (Restriction Analysis – to check if the input data defined allows executing the corresponding test step), test case execution coverage (using *passed/executed* metrics through Execution Analysis), as well as code coverage analysis.

The case study illustrates the main functionalities of the tool and the value of combining several coverage metrics. Through this case study, we conclude that there is not ONE best coverage metric to evaluate the quality of test cases. Rather, the best approach is a combination of several distinct ones that allow the tester to merge the strengths of each to obtain fuller, more useful, coverage information.

Possessing the overall coverage feedback provided by PARADIGM-COV, the tester can now go back and add extra configurations to the original model to account for these problems and gaps, thereby trying to exercise more of the website functionalities under analysis and, hopefully, to achieve a better quality system.

5.1 Satisfaction of Objectives

Regarding the objectives set out, we consider that the main goal of providing PARADIGM-ME with several types of coverage analysis feedback was achieved successfully. As of now, the coverage analysis performed by PARADIGM-COV effectively provides the intuitive and easy to grasp coverage information display we hoped to achieve, by using a colour system that quickly allows to identify potential problem areas.

Conclusions and Future Work

Model coverage analysis is at this moment fully developed, and the tester is able to fine-tune his PARADIGM-ME test configurations in order to obtain the maximum coverage for the test patterns available.

A base version of script coverage analysis was also achieved, providing the tester with an automated way to compare a custom-made test script with a PARADIGM model, although this feature in particular has still much potentiality to grow in regards to functionality.

Dynamic analysis too was well succeeded, and now it is possible to detect the discrepancies between the model and the test execution process as intended. Where previously a non-executed test meant the whole testing process halted, now the system is able to recover from error states and provide information on these to the tester.

At last, the tester can now make use of source code available, since PARADIGM-COV has added instrumentation and code coverage analysis to PHP files, providing both block and line coverage criteria, and packing a custom code coverage viewer that permits to ascertain the exact location of non-exercised code.

Additionally, an article detailing the work here developed was also created and submitted for peer review, namely, "Test Coverage Analysis" by L. Vilela and A. C. R. Paiva, to *the 25th IFIP International Conference on Testing Software and Systems (ICTSS'13)*, to be held on Turkey, Istanbul on November 2013.

Throughout the way, the PARADIGM-ME system itself was also updated from its initial version. Among other things, it now is able to define and manage several types of test constraints. In turn, this addition enabled to perform constraint analysis, enabling the early detection of test cases that will never meet the necessary conditions to their execution.

5.2 Future Work

In the end, while we found the case study results encouraging, there is still vast room for improvement.

For future work, the extension of code coverage analysis to other programming languages besides PHP would be an important feature, allowing the tester to analyse a larger pool of applications. It would also be interesting to explore more upon coverage of test paths in particular, as well as finding ways of crossing the information from the various coverage sources in a more interrelated way.

Additionally, we consider that it is important to perform usability tests on the current PARADIGM-COV, in order to make it more user-friendly. For an example in particular, and while we consider colours to be a visually intuitive marker, we are aware that not everyone reacts to colour in the same way. We believe the reports would greatly benefit from added markers and alternative customization, that would allow each tester to adapt the application to their needs, and thereby enable easy model analysis to be more widely accessible.

References

- [AFP11] 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.
- [AO08] Paul Ammann and Jeff Offutt. *Introduction to Software Testing*. Cambridge University Press, January 2008.
- [Ber] Sebastian Bergmann. PHPUnit manual. Available at <http://www.phpunit.de/manual/3.8/en/code-coverage-analysis.html>. Accessed on February 2013.
- [CPFA10] 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 V1 – 202, 2010.
- [CS12] 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.
- [EFW02] Ibrahim K. El-Far and James A. Whittaker. Model-based software testing. In *Encyclopedia of Software Engineering*. John Wiley & Sons, Inc., 2002.
- [Esl08] M.M. Eslamimehr. The survey of model based testing and industrial tools. Master’s thesis, Master Thesis, Linköping University, 2008.
- [GKSB11] 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.
- [Hui07] 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.
- [IBMa] IBM. Data sheet: Rational PurifyPlus. Available at <ftp://public.dhe.ibm.com/software/rational/web/datasheets/version6/pplus.pdf>. Accessed on February 2013.
- [IBMb] IBM. Develop fast, reliable code with IBM Rational PurifyPlus. Accessed on February 2013.
- [IBM07] IBM. IBM information center masthead. Available at <http://pic.dhe.ibm.com/infocenter/clmhelp/v4r0/topic/com.ibm.help.common.resources.doc/banner/banner.htm>, May 2007. Accessed on February 2013.

REFERENCES

- [IBM13] IBM. Safety-related software development using a model-based testing workflow. Available at <http://www.ibm.com/developerworks/rational/library/safety-related-software-development/index.html>, January 2013. Accessed on February 2013.
- [IST12] ISTQB. ISTQB standard glossary of terms used in software testing. Available at <http://www.istqb.org/downloads/viewcategory/20.html>, 2012. Accessed on February 2013.
- [Kim03] 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*, pages 145–155. IBM Press, 2003.
- [Lip08] Lipsum. Lorem ipsum. Disponível em <http://www.lipsum.com/>, acessido a última vez em 19 de Maio de 2008, 2008.
- [Mat] MathWorks. Model coverage analysis - simulink design verifier. Available at <http://www.mathworks.com/products/sldesignverifier/description5.html>. Accessed on February 2013.
- [MCR⁺05] 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.
- [Mic] Microsoft. Code coverage basics with Visual Studio team system. Available at <http://msdn.microsoft.com/en-us/library/dd299398>. Accessed on February 2013.
- [MP13] T. Monteiro and A. C. R. Paiva. Pattern based gui testing modeling environment. In *TESTing Techniques & Experimentation Benchmarks for Event-Driven Software (TESTBEDS), 2013 4th International Workshop on*, 2013.
- [NMS09] 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.
- [Pac] Nimish Pachapurkar. PHPCoverage - an open-source code coverage measurement tool for php applications. Available at <http://phpcoverage.sourceforge.net/>. Accessed on February 2013.
- [Pre05] A. Pretschner. Model-based testing. In *27th International Conference on Software Engineering, 2005. ICSE 2005. Proceedings*, pages 722 – 723, 2005.
- [RdAFLP12] Francisco Rebello de Andrade, João Faria, Antónia 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.
- [SL10] 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.

REFERENCES

- [Sto05] Keith Stobie. Model based testing in practice at microsoft. *Electronic Notes in Theoretical Computer Science*, 111(0):5–12, January 2005.
- [Str] G. Struth. Software verification and testing. Available at <http://staffwww.dcs.shef.ac.uk/people/G.Struth/COM6854.html>. Accessed on February 2013.
- [TH02] Mustafa M. Tikir and Jeffrey K. Hollingsworth. Efficient instrumentation for code coverage testing. *SIGSOFT Softw. Eng. Notes*, 27(4):86–96, July 2002.
- [too] Jscover user manual. Available at <http://tntim96.github.com/JSCover/manual/manual.xml>. Accessed on February 2013.
- [too08] Conformiq Qtronic SG: Semantics and algorithms for test generation. Available at <http://www.verifysoft.com/ConformiqQtronicSemanticsAndAlgorithms-3>, 2008. Accessed on February 2013.
- [too11] Conformiq Designer 4.4 user manual. Available at <http://www.verifysoft.com/ConformiqManual>, 2011. Accessed on February 2013.
- [VCG⁺08] 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.
- [Wei] S Weißleder. Coverage simulator. Available at <http://covsim.sourceforge.net/>. Accessed on January 2013.
- [Wei10a] 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.
- [Wei10b] S. Weißleder. *Test models and coverage criteria for automatic model-based test generation with UML state machines*. PhD thesis, Humboldt-University Berlin, Germany, 2010.
- [Wey86] E.J. Weyuker. Axiomatizing software test data adequacy. *IEEE Transactions on Software Engineering*, SE-12(12):1128–1138, December 1986.
- [WLWL07] 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.
- [WR11] 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.
- [YLW09] 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.
- [ZHM97] 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.

REFERENCES

Appendix A

Constraint Syntax

In this appendix we present a more complete overview on the syntax used to define constraints.

A.1 Constraint Format

A constraint can take one of the following basic formats:

```
<UITP>.<field> <operator> <literal>  
<UITP>.<field> <operator> <UITP>.<field>
```

Where `UITP` is the name of a UITP element in the model, and `field` is an input variable belonging to a TG of said UITP.

Additionally, some UITPs such as `Sort` and `Find` allow to apply a restricted keyword to a field, in the case of that field itself possessing its own attributes.

```
<UITP>.<field>.<__reserved> <operator> <literal>
```

A constraint expression is formed by grouping two or more constraints through the use of a logical operator:

```
<constraint> <AND|OR> <constraint>
```

Priorities in analysing constraints are expressed through the use of parenthesis (see Examples in [A.3](#)).

A.2 Constraint Tables

Table A.1: Literal types allowed in a constraint.

Literal Type	Example
String	"abc"
Set	{a, b, c}

Constraint Syntax

Table A.2: Constraint operators.

Operator	Description	Constraint Example
==	equal	<code>Login.username == "abc"</code>
!=	different	<code>Input.field != "pass"</code>
<	smaller than	<code>Login.username.len() < 5</code>
>	bigger than	<code>MD.countryName.len() > 2</code>
contains	contains	<code>MD.__details contains {d1, d2, d3}</code>

Table A.3: Reserved fields in use for each UITP type.

UITP	Reserved Keyword	Return Type	Description
MasterDetail	<code>__master</code>	Set	Refers to the set containing all the input fields of type 'master' for this UITP (<code>UITP.__master</code>).
	<code>__details</code>	Set	Set containing all input data associated with the field 'details' for all TGConfs within a UITP (<code>UITP.__details</code>), or for all the TGConfs within a UITP which contain a specific master value (<code>UITP.mastervalue.__details</code>)
Find/Sort	<code>__fields</code>	Set	Set with all the field names defined for within a Find or Sort UITP element (<code>FindUITP.__fields</code>).
	<code>__check</code>	String	Returns the content of a Find or Sort's check (<code>UITPSort.__check</code>)
	<code>__position</code>	String	Returns the content of the field <i>position</i> , referent to a Find or Sort's TGConf (<code>UITPFind.findfield.__position</code>)
	<code>__result</code>	String	As with above, returns the data for field <i>result</i> (<code>UITPFind.searchfield.__result</code>)

Table A.4: Current methods supported by constraints.

Method	Description
<code>.len()</code>	If applied to a string literal, it returns its length; when applied to a set literal, it returns the number of elements within the set.

A.3 Examples

Here we present some examples on some constraint expressions and their respective results, for some specific UITP configurations.

Example of Element Configurations		
"MDCities_1"	UITP	MasterDetail
	Configurations	{"Portugal"={"Lisboa", "Guarda", "Braga", "Beja"}, "Spain"={"Madrid", "Barcelona"}}
"Login_2"	UITP	Login
	Configurations	{username="john_doe", password="mypass"}
"Find_3"	UITP	Find
	Configurations	{userSearch="david doe", country_origin="Peru"}

Constraint	Result
MDCities_1.Portugal contains {Lisboa, Beja}	True
MDCities_1.__master != {Portugal, Spain}	False
MDCities_1.__details contains {Guarda, Madrid}	True
MDCities_1.__master.len() == 2	True
Login_2.username == "janedoe"	False
Login_2.password.len() > 3	True
Find_3.__fields == {userSearch, country}	True
Login_2.username == Find_3.userSearch	False
MDCities_1.__master contains Find_3.country	False
Find_3.userSearch.len() > 0 AND Login_2.username == "admin" OR Find_3.country == "Peru"	False
(Login_2.username == "johndoe" AND (Find_3.country == "Peru" OR Find_3.country == "Spain"))	True