

# Multidimensional test coverage analysis: PARADIGM-COV tool

Ana C. R. Paiva<sup>1</sup>  · Liliana Vilela<sup>1</sup>

Received: 6 November 2014 / Revised: 17 October 2016 / Accepted: 4 January 2017 / Published online: 16 January 2017  
© Springer Science+Business Media New York 2017

**Abstract** Currently, software tends to assume increasingly critical roles in our society so assuring its quality becomes ever more crucial. There are several tools and processes of software testing to help increase quality in virtually any type of software. One example is the so called model-based testing (MBT) tools, that generate test cases from models. Pattern Based Graphical User Interface Testing (PBGT) is an example of a MBT new methodology that aims at systematizing and automating the Graphical User Interface (GUI) testing process. It is supported by a Tool (PBGT Tool) which provides an integrated modeling and testing environment for crafting test models based on User Interface Test Patterns (UITP) using a GUI modeling Domain Specific Language (DSL) called PARADIGM. Most of the MBT tools have a configuration phase, where test input data is provided manually by the tester, which influences the quality of the test suite generated. By adding coverage analysis to MBT tools, it is possible to give feedback and help the tester to define the configuration data needed to achieve the most valuable test suite as possible and, ultimately, contribute for increasing the quality of the software. This paper presents a multidimensional test coverage analysis approach and tool (PARADIGM-COV), developed in the context of the PBGT project, that produces coverage information both over the PARADIGM model elements and during test case execution (to identify the parts of the model that were actually exercised). It also presents a case study illustrating the benefits of having multidimensional analysis and assessing the overall test coverage approach.

**Keywords** Software testing · Test coverage analysis · Model-based testing · GUI testing

## 1 Introduction

As software complexity grows, it becomes increasingly difficult to assure its quality. Adding this to the increasing role of software in our daily lives, it becomes clear the ever-growing importance of software testing. Today, we have a wide range of software testing tools available that automate many aspects of the quality assurance process, which facilitates the generation of a larger quantity of test cases. Still, it is widely known that the number of tests executed does not assure system quality by itself. Rather, more than the number of tests generated, it is the quality of those tests that matters.

For one to conclude that the system has been tested thoroughly, not only must it pass the defined test cases, but one must assure that test suites themselves have met certain criteria. These criteria are usually based on metrics that allow one to know the extent to which a system (or model) is exercised by the test suites (coverage criteria), or, according to Weyukers definition: The purpose of testing is to uncover errors, (...) the purpose of an adequacy criterion is to assess how well the testing process has been performed” [1].

Although coverage criteria and test case generation have generally been well-documented in literature, El-Far and Whittaker noted a lack of in-depth studies regarding the coverage of models in particular [2]. In fact, they are not the only authors to point coverage metrics as one of the drawbacks of model-based testing (MBT). Eslamimehr has also indicated that coverage metrics prevents the MBT from being more widely applied, claiming that common metrics are insuffi-

---

✉ Ana C. R. Paiva  
apaiva@fe.up.pt

<sup>1</sup> INESC TEC and Faculty of Engineering, University of Porto, Porto, Portugal

cient, as stated previously, and that counting test cases is not enough, especially if these are automatically generated [3].

Model and code coverage information is particularly important for MBT tools, as most of them require a test configuration phase in which the tester manually provides input data that influences the quality of the test suite generated. With coverage feedback, the tester is capable of adapting the test input data to achieve the highest degree of coverage as possible, so as to improve the quality of the final test suite generated and, ultimately, contribute for better software.

This work presents an approach and tool (PARADIGM-COV) developed in the context of a wider project called Pattern Based GUI Testing (PBGT) that aims to provide a new MBT approach to test software applications through their GUI without needing access to their source code. PARADIGM-COV performs test coverage analysis which provides diversified coverage information both over the PARADIGM graphical model elements (to assess if input data is adequate to cover the test goals and assess if preconditions are achievable), and during test case execution (to calculate the parts of the model that were effectively exercised). Besides this information, the tool also provides code coverage analysis (when the code is available) and test script analysis to assess the quality of a manually built test script regarding a PARADIGM model.

This paper extends the work published in [4] by

- providing more information about the PBGT project in which context this work is developed. In particular, the PBGT process is described in more detail and the test case generation algorithm explained along a theoretical example with all the steps described (Sect. 3);
- explaining how the PARADIGM-COV tool fits within the overall PBGT process (in Sect. 4);
- describing an additional functionality of the PARADIGM-COV tool regarding test script analysis (in Sect. 4.4);
- extending the case study in Sect. 5 with the complete model (both levels of the model), the configuration data provided, a more detailed description about the test generation process and the total coverage analysis results obtained, in particular, the coverage over the second level of the model, the result of constraint analysis and script analysis were added.

This paper is structured into six sections. After the introduction, Sect. 2 describes the state of the art regarding coverage analysis. The following section gives an overview of the PBGT project. Section 4 describes the developed coverage analysis tool. Section 5 demonstrates the functionalities of the tool through case-studies. The last section presents conclusions and future work.

## 2 State of the art

Modeling is a technique fundamental for software engineering and models can be found in several contexts (e.g., [2,5,6]). Model driven engineering is a software development methodology in which abstractions (or models) are built and used to generate code. These models are abstract entities that are used to handle the application complexity and automate the software engineering process [7]. The goal is to increase productivity and promote communication. In software testing models can be used to describe the systems to test or the testing goals and from there test cases are generated automatically. This is called Model Based Testing (MBT). In this case, the coverage analysis of the test cases may be performed over the model instead of the code.

There are not many research papers about MBT model coverage in which the coverage analysis itself is the main goal. Often times, model coverage criteria are used merely as a means to guide automatic test generation [8,9]. Other times, instead of analyzing the model directly, the test coverage of a model is found by generating code from said model, making then a coverage analysis on that code [10]. We consider this to be code coverage at its core, and not a coverage analysis purely based on a model. Still, there exist some tools and methods that attempt to approach this problem.

The work of Weißleder is based on semantic preserving state machine transformations [11,12]. Instead of trying to achieve coverage for a complex model, he tries to achieve coverage for a simpler, but equivalent model, i.e., the stronger coverage criteria can be simulated through the use of a weaker one. 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 [13]. Coverage Simulator is a tool prototype that intends to demonstrate this approach [12].

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 [14], we believe it can be adapted to other uses, for instance, as a facilitator to analyze the coverage of any model in general. However, in such case, new transformation rules may be needed and the impact of the transformations is ultimately linked with the coverage criteria being used, which may not always be positive. In addition it is also important to consider the trade-off in what concerns the performance of a tool applying this technique and to consider the scalability of such approach in regards to the complexity growth in the original models.

Regarding industrial practices in coverage in MBT tools, we found it hard to differentiate between the techniques used

to evaluate coverage and the ones used to generate the test cases automatically. The focus was on using coverage criteria as test generation guiding mechanisms. 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) [8]. A similar case can be found with Spec Explorer. Here, what is defined as coverage criteria is effectively used as test selection criteria, being Transition Coverage the most used one [9, 15].

Simulink Design Verifier tool 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, such as Branch Coverage, but also reports coverage on metrics like test objectives or constraints [10]. Other tools that seem to use the coverage criteria to generate test cases include Conformiq Qtronic and Smartesting CertifyIt [16].

In what concerns the analysis of the model itself, the most common approaches rely on exploring the state-machine derived from the model. The process, by which this is achieved, however, varies. For instance, Conformiq Qtronic tools make use of coverage checkpoints in the model [17]. With SCADE Suite, coverage analysis is done by verifying the activation of each element in the model as the system is executed. Andrade et al. use algebraic specifications, expressed in ConGu, to generate an Alloy model that obeys said specifications [18]. This technique was expanded further to support the generation of test cases for Java generic classes in particular [19].

While most commercial MBT tools found at this point do specify the basic type of functionalities provided, including coverage, they do not make it clear in what way exactly is the criterion being analyzed. Such we have found to be the case with IBMs Rational Rhapsody for instance [20]. 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 Labiches report on the subject [16].

One of the problems we found with coverage analysis tools is that they offer a certain set of coverage criteria and only some of them offer various distinct criteria. A common tool seems to provide a relatively small array of options regarding coverage criteria, and when they offer larger choice, the criteria themselves seem to implement similar base approaches.

The goal of this research work is to provide a new approach in which testers can benefit from multidimensional coverage analysis (i.e., analysis from several viewpoints) to improve the quality of their tests and, ultimately, the quality of the final software product. The PARADIGM-COV tool imple-

ments such coverage analysis providing the tester with the most varied set of coverage analysis approaches possible, and attempting to bring together the strengths of each. With such diversity of coverage metrics, the tester is better prepared to assess the quality of the test suite and implement the necessary actions to increase the quality of such tests.

### 3 PBGT—Pattern Based GUI Testing

PARADIGM-COV is developed in the context of the PBGT project [21]. The goal of PBGT is to provide a means to test Graphical User Interfaces (GUIs) by providing generic reusable test strategies for testing recurrent behavior. This approach relies on the fact that most GUIs end up having similar elements and behavior (the so called User Interface Patterns), which can be reused across several applications. PBGT defines generic test strategies able to test slightly different implementations of User Interface Patterns after a configuration phase. Those generic test strategies are called User Interface Test Patterns (UITP). A UITP defines a test strategy as a set of test goals (TG), each with the form

$$\langle \textit{Goal}; V; A; C; P \rangle \quad (1)$$











where *Goal* is the name of the test goal, *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 UITPs are defined within a graphical Domain Specific modeling Language (DSL) called PARADIGM. This DSL has (Table 1) structural elements (to structure the model in different levels of abstraction—Form element—, and to define a set of elements that can be exercised in any order—Group element); behavioral elements (UI Test Patterns—UITP); Init and End elements (to mark the start and end points of a model); and connectors, that establish relations among elements and define their sequencing.

PBGT tool [21] is the modeling environment that supports the PGBT testing approach (Fig. 1). It supports

- building models (from scratch) written in the PARADIGM graphical notation [5],
- building part of the model by a reverse engineering process and complete it manually afterwards [22],
- the configuration phase where the tester selects the testing goals (TGs) and provides test input data,
- establishing the mapping between model elements and GUI controls to identify the controls in which to act during test case execution,

**Table 1** PARADIGM's syntax

| Type        | Element                    | Symbol  |
|-------------|----------------------------|---|
| Structural  | Group                      |  |
|             | Form                       |  |
| Behavioural | Login                      |  |
|             | Input                      |  |
|             | MasterDetail               |  |
|             | Sort                       |  |
|             | Find                       |  |
| Other       | Call                       |  |
|             | Init                       |  |
|             | End                        |  |
| Connectors  | Dependency                 | ->>>  |
|             | Sequence                   | >>  |
|             | Sequence with Data Passing | []>>  |
|             | Sequence with Moved Data   | o[]>>   |

- generating test cases from PARADIGM models [23],
- execution of the test cases over the application under test (AUT),
- building reports with the test cases execution results.

PARADIGM-COV is an additional component that works within this environment to provide coverage analysis to help the tester during the configuration phase and during the analysis of the test results [4]. The way this tool integrates into the overall PBGT process can be seen as dashed activities, artifacts and arrows within Fig. 1.

The PBGT process has several steps which start with building the PARADIGM model. A PARADIGM model can be constructed from scratch by the tester or with the help of the reverse engineering tool. This reverse engineering tool identifies the UI patterns present on the GUI under test and builds a PARADIGM model with the UI Test Patterns able to test the identified UI patterns. In order to generate test cases from a PARADIGM model, the tester must go through a configuration step in which he selects the *TGs* for each UI Test Pattern within the model and provides test configuration data (TGConf):

1. the test input data,
2. the checks to perform, and

3. the preconditions defining the states in which a TGConf may be executed.

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 (resulting in different *TGConfs*). *TGConfs* are used in conjunction with the PARADIGM model itself for test case generation. Firstly, the tool generates test paths that guarantee full transition coverage (connectors within PARADIGM model); following, those test paths are expanded in order to exercise all *TGConfs* defined for the UITPs within the model.

A PARADIGM model can be structured into different levels of abstraction, thus, to generate test cases, the algorithm starts by (1) flattening the model. It does this by replacing recursively all structural elements (Forms and Groups) within a hierarchical level of a model by their internal elements, present at the next level, until all Forms and Groups are discarded. After this flattening process, (2) it calculates the set of paths (*SPaths*) that go from the Init to the End elements within the model and guarantees full connector coverage. Considering that some UITPs can be optional (as is the case of *ID[2][True]* inside Fig. 2), some additional paths may be created ignoring such optional UITPs. Then, (3) it expands every path within *SPaths* into test cases according to the *TGConf* defined for the UITP within each path. At the end, the algorithm (4) discards all the configurations that cannot be executed because their precondition can never be True. More details about this generation process can be found in [21].

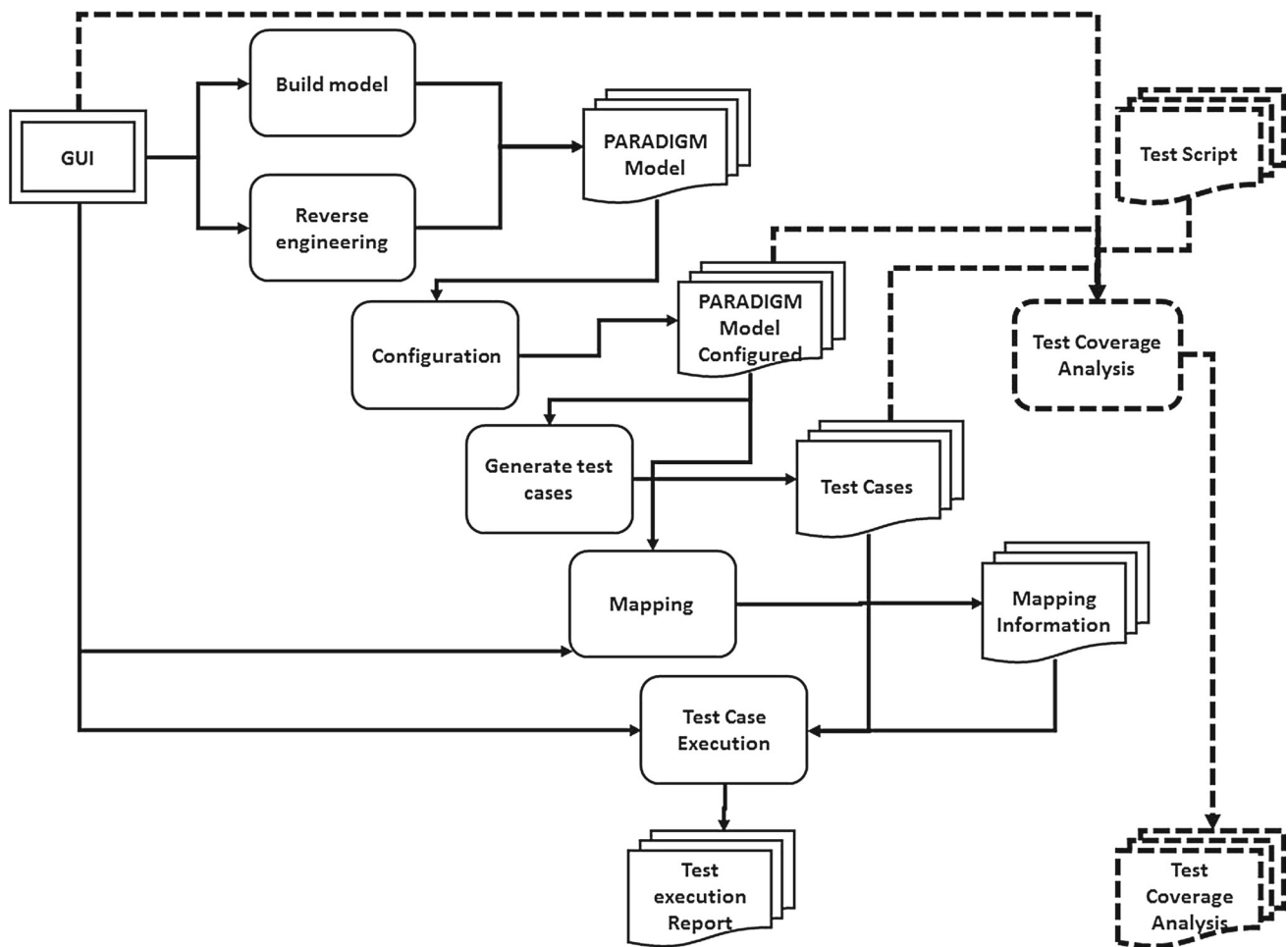
Figure 2 illustrates a model written in two levels of abstraction (*Level0* and *Level1*) on the left side and the corresponding flattened diagram on the right side. The UITPs within the model have a graphical representation (its icon as listed in Table 1), and a label with the format:

*str[id][optional]*

where *str* is the name of the UITP, *id* is a unique number identifying the UITP, and *optional* is a Boolean value indicating if the UITP is mandatory or optional.

As it can be seen in Fig. 2, the flattened model does not have Groups nor Forms. Because a Group means that their inner elements can be executed in any order, it will be possible to exercise Country[1] followed by ID[2] or ID[2] followed by Country[1], so we have four possible paths along the flattened model.

*Path1* : Init → UITP1 → UITP2 → UITP3.1 → End  
*Path2* : Init → UITP1 → UITP2 → UITP3.2 → End  
*Path3* : Init → UITP2 → UITP1 → UITP3.1 → End  
*Path4* : Init → UITP2 → UITP1 → UITP3.2 → End



**Fig. 1** PBGT process

However, since UITP2 is optional, we will have two additional paths:

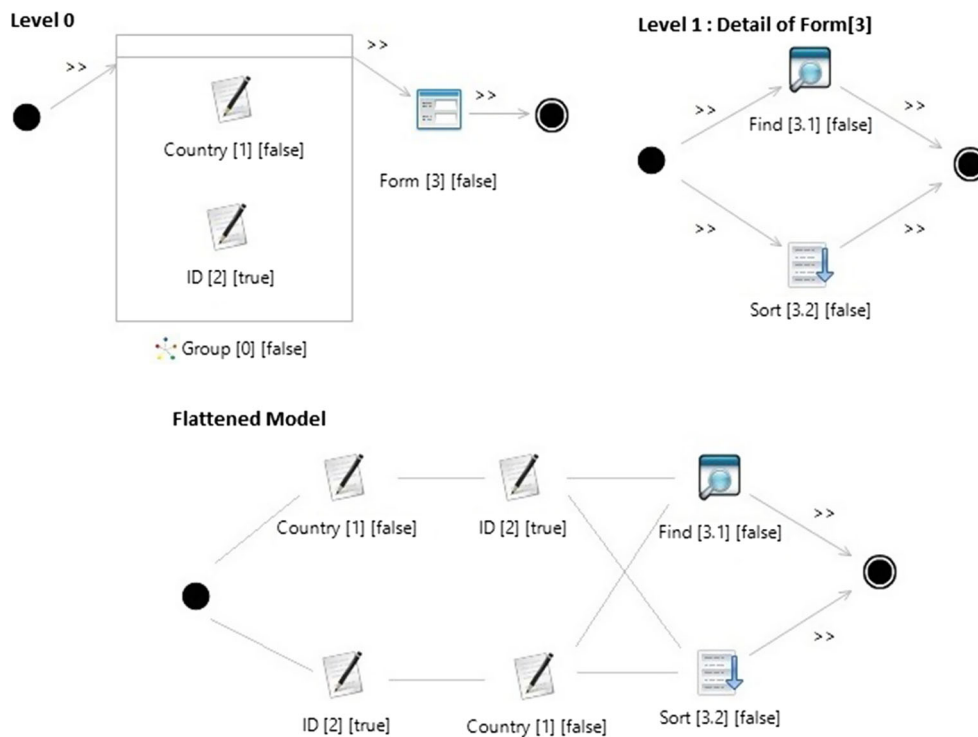
*Path5* : *Init* → *UITP1* → *UITP3.1* → *End*  
*Path6* : *Init* → *UITP1* → *UITP3.2* → *End*

After calculating the set *SPaths*, it is time to calculate test cases. *Test cases* include input values, execution preconditions and expected results. For that, we need to consider the configurations provided by the tester for each UITP in the model. Let us consider that *UITP1* has two configurations defined (*c11*, *c12*), *UITP2* has one configuration defined (*c21*), *UITP3.1* has one configurations (*c311*) and *UITP3.2* has two configurations (*c321*, *c322*), hence the final test suite would be constituted by the following test cases:

*tc01* : *Init* → *UITP1*(*c11*) → *UITP2*(*c21*) → *UITP3.1*(*c311*) → *End*  
*tc02* : *Init* → *UITP1*(*c12*) → *UITP2*(*c21*) → *UITP3.1*(*c311*) → *End*

*tc03* : *Init* → *UITP1*(*c11*) → *UITP2*(*c21*) → *UITP3.2*(*c321*) → *End*  
*tc04* : *Init* → *UITP1*(*c11*) → *UITP2*(*c21*) → *UITP3.2*(*c322*) → *End*  
*tc05* : *Init* → *UITP1*(*c12*) → *UITP2*(*c21*) → *UITP3.2*(*c321*) → *End*  
*tc06* : *Init* → *UITP1*(*c12*) → *UITP2*(*c21*) → *UITP3.2*(*c322*) → *End*  
*tc07* : *Init* → *UITP2*(*c21*) → *UITP1*(*c11*) → *UITP3.1*(*c311*) → *End*  
*tc08* : *Init* → *UITP2*(*c21*) → *UITP1*(*c12*) → *UITP3.1*(*c311*) → *End*  
*tc09* : *Init* → *UITP2*(*c21*) → *UITP1*(*c11*) → *UITP3.2*(*c321*) → *End*  
*tc10* : *Init* → *UITP2*(*c21*) → *UITP1*(*c11*) → *UITP3.2*(*c322*) → *End*  
*tc11* : *Init* → *UITP2*(*c21*) → *UITP1*(*c12*) → *UITP3.2*(*c321*) → *End*  
*tc12* : *Init* → *UITP2*(*c21*) → *UITP1*(*c12*) → *UITP3.2*(*c322*) → *End*  
*tc13* : *Init* → *UITP1*(*c11*) → *UITP3.1*(*c311*) → *End*





**Fig. 2** Diagram depicting PARADIGM-MEs path generation process

$tc14 : Init \rightarrow UITP1(c12) \rightarrow UITP3.1(c311) \rightarrow End$   
 $tc15 : Init \rightarrow UITP1(c11) \rightarrow UITP3.2(c321) \rightarrow End$   
 $tc16 : Init \rightarrow UITP1(c11) \rightarrow UITP3.2(c322) \rightarrow End$   
 $tc17 : Init \rightarrow UITP1(c12) \rightarrow UITP3.2(c321) \rightarrow End$   
 $tc18 : Init \rightarrow UITP1(c12) \rightarrow UITP3.2(c322) \rightarrow End$

where  $UITPx(cy)$  is the instance (or  $TGConf$ ) of the UI Test Pattern number  $x$  with configuration values defined in  $cy$ . Above, the first two test cases ( $tc01$ ,  $tc02$ ) are generated by expanding the first test path ( $Path1$ ), test cases ( $tc03$ ,  $tc04$ ,  $tc05$  and  $tc06$ ) are generated by expanding the second test path, test cases  $tc07$  and  $tc08$  are obtained by expanding test path 3, test cases  $tc09$ ,  $tc10$ ,  $tc11$ , and  $tc12$  are obtained by expanding test path 4, and so forth.

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,  $tc01$  and  $tc02$  are of the same family, while  $tc03$ ,  $tc04$ ,  $tc05$ , and  $tc06$  belong to another family.

Now consider that configuration  $c21$  has a precondition that is only True when configuration  $c11$  is executed previously. In this case, the final set of test cases would be  $tc01$ ,  $tc03$ ,  $tc04$ ,  $tc13$ ,  $tc14$ ,  $tc15$ ,  $tc16$ ,  $tc17$ ,  $tc18$ .

To execute the generated test cases over a GUI, we need to establish a mapping between UITPs within the model and GUI controls in which the corresponding test strategy will be executed. This is done by a point and click process (similar to

the one presented in [24]). The user selects first the UITP to map and then points and clicks over the GUI controls where the corresponding test strategy will be executed. During this process some properties of the controls are saved for allowing their identification during test case execution.

During test case execution, the tool looks for the GUI controls related to the current configuration (the current execution step of the test case) based on the previous gathered properties and acts on those controls according to the sequence of actions (A) defined within such configuration. At the end of the test process, a report is produced with the errors found (all the checks evaluated to False).

The PBGT overall testing approach has some steps that are fully automatic and others that need data provided manually by the tester. This data influences the quality of the generated test cases. So, a coverage analysis approach that could help assess the quality of the generated test cases according to different perspectives would help to define adequate test input and increase the quality of the test cases. The overall multidimensional coverage analysis approach and tool (PARADIGM-COV) are described next.

#### 4 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 defined. In addition, it may be hard to analyze, after test case execution, which configurations were executed and the checks that returned False. One of the goals of PARADIGM-COV is to provide a diversified array of coverage information to aid the tester during the test configuration phase, in order to help him define test data that may result in the generation of test cases with the maximum level of coverage as possible, and to help him analyze the results achieved after test case execution. The way PARADIGM-COV fits within the overall PBGT process is illustrated in Fig. 1. It provides coverage analysis over the model, over the test cases, over the code of the GUI (when such code is available) and over a test script. As such, the PARADIGM-COV tool has four main goals:

1. *Model coverage analysis* to provide feedback on whether the test configurations defined by the tester are considered "adequate" according to the test strategy defined for each UITP;
2. *Execution analysis* performed during test case execution to provide information about the configurations that were effectively executed and the corresponding checks that passed or failed;
3. *Code analysis* to provide feedback on the degree to which the code of the actual GUI under test was exercised during the test case execution process (only possible when code is available);
4. *Script coverage analysis* to check the extent to which a manually built test script is able to execute the UITPs inside an existing paradigm model.

#### 4.1 Model coverage analysis

The first goal of PARADIGM-COV is to provide model coverage information. The model coverage analysis functionality is based on two main premises:

1. *Test goal analysis*—to assess whether all *TGs* within the test strategies defined for UI Test Patterns within the model were configured by the tester;
2. *Constraint analysis*—to evaluate if the configurations provided by the tester allow reaching states where the preconditions defined hold.

Test Goal Analysis aims at checking if there are configurations (*TGConf*) defined for every *TG* within the test strategy of the UITPs of a PARADIGM model. Despite the fact that structural elements alone do not have test strategies associated, since their purpose is simply to allow the structuring of the model (in groups or form levels), the tool also provides goal coverage analysis on them. As such, for

any structural element *Se*, its goal coverage,  $gCov(Se)$ , is the average of the goal coverage of the *N* elements *e* inside it:

$$gCov(Se) = \frac{\sum_{i=1}^N gCov(e_i)}{N} \quad (2)$$

Behavioral elements on the other hand, are UITPs defining a test strategy as a set of test goals (*TGs*). So, according to Test Goal Analysis, a UITP is fully covered if it has configurations (*UITP.TGConf*) defined for all its *TGs* (*UITP.TG*).

$$\forall tg \in UITP.TG, \exists c \in UITP.TGConf \cdot c.Goal = tg.Goal \quad (3)$$

For example, the Login UITP defines a test strategy with two test goals (*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. In the worst case scenario, 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 to include configurations for both valid (*G\_LV*) and invalid (*G\_LINV*) login specifications to exercise all its *TGs*. The information relating the set of *TGs* defined by the test strategies of each UITP within the PARADIGM DSL can be consulted in Table 2.

The result of the Test Goal Coverage Analysis is represented on the model by painting each UITP element with different colors (Fig. 3). Fully Covered: green; Uncovered (i.e., no *TGConfs* defined): red; and Partial (i.e., *TGConfs* defined for some *TGs*): yellow. With this information, the

**Table 2** Coverage criteria for each PARADIGM UITP

| UITP         | Set of test goals   |
|--------------|---|
| Call         | <i>G_Call</i> —test the result of a call  |
| Find         | <i>G_Found</i> , <i>G_notFound</i> —test searches returning and not returning values  |
| Input        | <i>G_IV</i> , <i>G_IINV</i> —test for valid and invalid inputs                        |
| Login        | <i>G_LV</i> , <i>G_LINV</i> —test for valid and invalid authentications               |
| Sort         | <i>G_SRTASC</i> , <i>G_SRTDESC</i> —test the sort for ascending and descending order  |
| MasterDetail | <i>G_MD</i> —test if changing the value of the master, the detail updates accordingly |



**Fig. 3** Test goal coverage analysis (Color figure online)



**Fig. 4** Constraint coverage analysis (Color figure online)

tester can easily analyze the UITP painted in red or yellow to complete the configuration process to generate higher quality test cases. Of course he can also opt to leave some UITPs with an incomplete configuration because he does not want to exercise some of the testing goals at a specific moment of the testing process. In order to proceed to test case generation, it is not mandatory to have all UITPs fully configured. However, with coverage analysis over the model it is easy to evaluate which configurations are missing whenever needed.

Analyzing 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* has a precondition that must hold in order for it to be executed, so it is useful to provide information about preconditions that never hold because the test input data provided by the tester does not allow it. This is the goal of Constraint Analysis.

Ergo, an element is considered valid constraint-wise if, for every configuration belonging to said element (*UITP.TGConf*), there is a state (variables and corresponding input data) in which its associated precondition, *P*, holds (Formula 4).

$$\forall c \in \text{UITP.TGConf}, \exists s : \text{State} \cdot c.P(s) = \text{true} \quad (4)$$

This information is depicted as a colored box around the label of the corresponding UITP element (Fig. 4). The box is green when all its constraints may hold, yellow when only some are met, and red when no preconditions are ever True.

One can see the result of Test Goal Analysis and Constraint Analysis simultaneously over the model, and get a broader picture of the coverage information. Analyzing both *Test Goal Analysis* and *Constraint Analysis* allows you to get a broader view of the quality of your configuration data (*TGConfs*). For example, if for one UITP you get *Test Goal Analysis = Partial* and *Constraint Analysis = Uncovered*, it means that some test goals are configured but the constraints are never met so test cases will not execute those configurations.

## 4.2 Execution analysis

PARADIGM-COV provides information concerning the detection of discrepancies between the tests cases generated from the model and the ones effectively executed on the AUT. This dynamic analysis allows the tester to differ-

entiate between a test that fails (i.e., the checks defined are False) from the one that was not executed (e.g., the AUT crashed unexpectedly, or a target web page element was not found).

While the model coverage analysis evaluates if test cases completely cover the model, execution coverage analysis evaluates if the test cases were completely executed on the AUT. So, it is possible to reach full execution coverage even when test cases do not reach full model coverage.

The execution report built by PARADIGM-COV displays several metrics pertaining to both test check results (*cr*) and test execution (*er*) coverage data. While check results indicate whether checks within a *TGConf* passed or failed, execution result simply reports whether that given *TGConf* was executed or not. The report is divided into two distinct data sections:

- element statistics and
- test case statistics.

### 4.2.1 Element statistics

For each UITP within the PARADIGM model, element statistics displays the percentage of *TGConfs* whose checks passed, as well as the percentage of *TGConfs* that were executed. The check result coverage can thus be formalized by formula 5 where *cr* is the check result for a specific test goal configuration (*tc* : *TGConf*), and *N* is the total number of configurations (*TGConf*) within all *TGs* specified for a UITP.

$$\frac{\sum_{i=1}^N (cr(tc_i) = \text{passed})}{N} \quad (5)$$

$$\frac{\sum_{i=1}^N (er(tc_i) = \text{executed})}{N} \quad (6)$$

Likewise, the percentage of executed *TGConf* for each UITP is given by formula 6 where *er* is the execution result of the test goal configuration (*tc*: *TGConf*), and *N* is still the total number of configurations associated with the *TGs* of the UITP in question.

### 4.2.2 Test case statistics

A test case is a sequence of steps, and each step is a configuration (*TGConf*) for a specific UITP. So, the tool presents the percentage of the overall configurations within a test case that were executed, and the percentage of the corresponding checks that passed/failed. This is Test Case Statistics.

Being *N* the total number of *TGConfs* within a test case *tc*, the percentage of passed tests within a test case is given by formula 7 and the percentage of *TGConfs* within a test case that were executed (*er*) is given by formula 8:



$$\frac{\sum_{i=1}^N (cr(tgc_i) = passed)}{N} \quad (7)$$

$$\frac{\sum_{i=1}^N (er(tgc_i) = executed)}{N} \quad (8)$$

Besides presenting overall percentages of executed and passed *TGConfs* within a test case, the tool also signals the *TGConfs* that were not executed and, additionally, for every *TGConf* executed, it reports information whether its checks passed or failed. In the latter case, it also provides a reason for failure based on exceptions that are caught by the tool during the test case execution (e.g., a UI element where an action should occur could not be found within the web page of the AUT). Ultimately, the aim with showing information under element and test case statistics is to ease the analysis of the test results, allowing to fix detected failures more easily.

### 4.3 Code coverage

Code coverage is not the main goal of PARADIGM-COV because PBGT aims to test AUTs through their GUI without needing access to the source code. Even so, when the code is available, code coverage analysis is also possible and may be helpful to identify which parts of the code were not exercised and need additional tests. In simple terms, code coverage consists in tracking which parts of the code are executed while exercising the software AUT.

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 AUT. Code coverage information is useful to inspect the parts of the code that were not exercised and to help to 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 cri-

teria. Block coverage does not indicate the lines that were executed but runs faster, since the number of probe insertions is vastly inferior. In any case, the code coverage ratio for a source file is calculated using the following formula:

$$\frac{\text{number of probes executed}}{\text{total number of probes}} \quad (9)$$

PARADIGM-COV produces a list with the files that were exercised and a report painting the files in green, yellow or red if they are fully, partially, or not exercised at all. Inside these files, code lines/blocks executed are painted in green, while the others are painted in red.

### 4.4 Script coverage

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.

The test scripts generated by PBGT tool are XML files that can be updated manually by the tester or even built from scratch and then executed by the PBGT tool. However, for executing such manually updated script, the tool needs a PARADIGM model and all the information captured during the mapping phase where UITPs are linked to real GUI controls. It is not possible to execute a manually built test script without a corresponding PARADIGM model. An example of a test script is illustrated in Fig. 5.

As you can see in Fig. 5, the test script contains two test cases (<Init/> marks the beginning of a test case) one which tests an invalid login (validity="Invalid") checking if the GUI stays on the same page (check="StayOnSamePage") and other to test a valid login (validity="Valid") and checks if the GUI changes the page (check="changePage").

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

**Fig. 5** Example of a test script

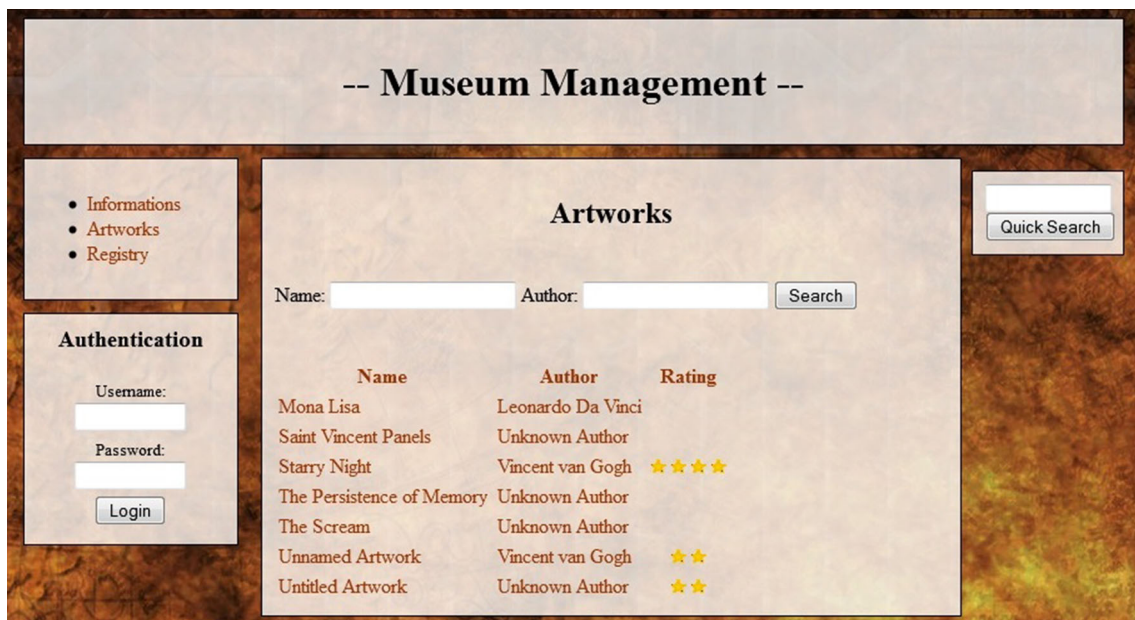


Fig. 6 Screenshot illustrating the main page of the museum management website

During the manual built test suite execution, the tool walks along the PARADIGM model and at the end it indicates which UITPs of the model were covered in the execution report.

## 5 Case study

After the implementation of the PARADIGM-COV tool, we conduct 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;
- assess the value of tool usage;
- gather information about possible tool improvements.

This case study is based on a real website, with source code available. The testing goals were described by a model written in PARADIGM with corresponding configurations. PARADIGM-COV was used to help the configuration activity and to assess the quality of the generated test cases.

### 5.1 Sample website museum management website

This website is an artwork museum management system (Fig. 6). It allows registry new users, authenticate, insert artwork data in the database according to permission rules, manage donations, search for artworks and sort the results obtained.

### 5.2 PARADIGM model

A PARADIGM [5] model describes the tests to perform over the AUT. In this case, the aim is to test

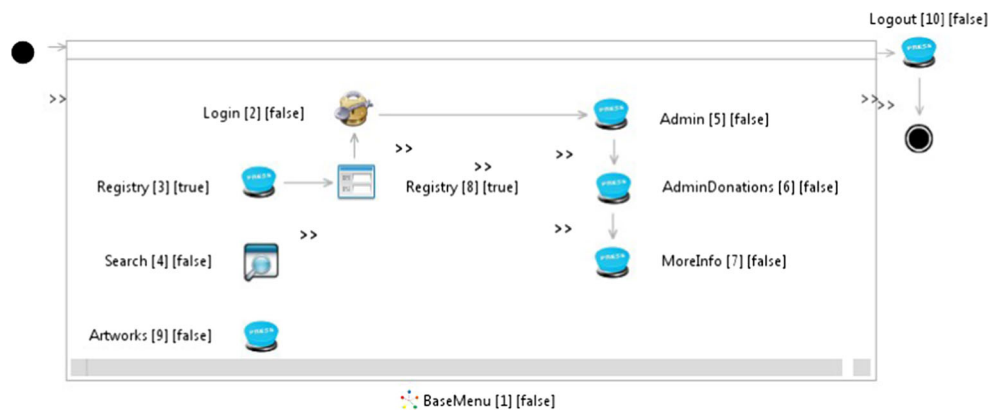
- the registration functionality,
- the authentication,
- the search mechanism,
- and a subset of the administrative functions accessible only for logged-in users.

The model built for testing the museum management website has two abstract levels and is illustrated in Figs. 7 and 8. The model of the first level contains one BaseMenu[1] Group (meaning that its inner elements can be accessed in any order) followed by a Logout[10] Call. The Registry[3] Call gives access to the Registry Form, detailed in another level of abstraction.

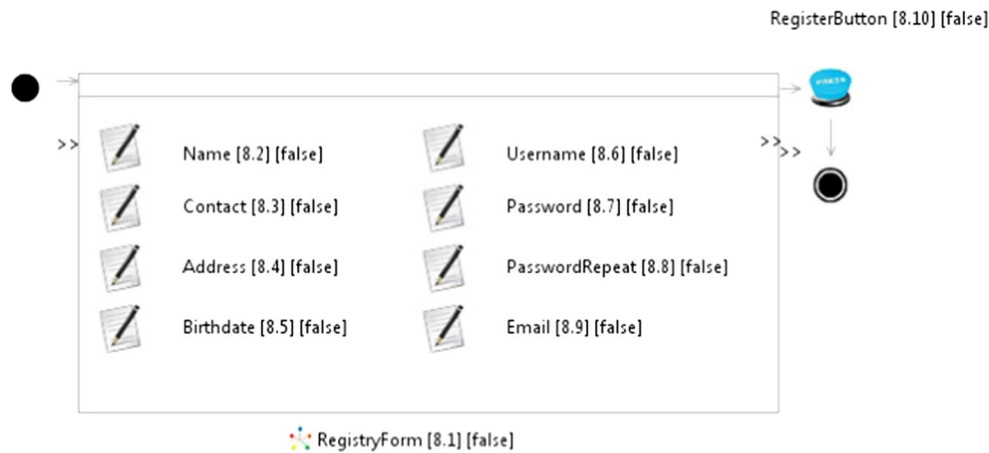
When two elements A and B are linked by a connector it means that configurations defined for B can only be executed after those defined for A. This is the case of the connection between Login[2] and Admin[5].

Both Registry[3] Call and Registry[8] Form are optional (i.e., there will be test paths that go from Init directly to Login[2]), as registration is only available for non-registered users (i.e., not available for authenticated users after Login[2]).

To conclude, Logout[10] will test if such Call closes the current session and returns the website to its initial state, where the user may register or login. The configurations



**Fig. 7** First level of the PARADIGM model for the artwork museum management system



**Fig. 8** PARADIGM model describing the test goals for the form Registry (8)

**Table 3** Configurations defined for the elements within 1st level model

| Id | Name         | Goals   | Input values                          | Checks  |
|----|--------------|---------|---------------------------------------|---|
| 2  | Login        | G_LV    | username="admin";<br>password="admin" | Text present in page =<br>="Logout"               |
|    |              | G_LINV  | username="abc";<br>password="abc"     | Stay on the<br>same page                          |
| 3  | RegistryCall | G_Call  | None                                  | Text present in page=<br>="Register Account"      |
| 4  | Search       | G_Found | search="starry night"                 | Number of results =1                              |
| 5  | AdminCall    | G_Call  | None                                  | Text present in page=<br>="Administration Module" |
| 6  | Donations    | G_Call  | None                                  | Text present in page=<br>="donations"             |
| 7  | MoreInfo     | G_Call  | None                                  | Text present in page=<br>="donations"             |
| 9  | Artworks     | None    | None                                  | None  |

defined for the UITP within the 1st level of the model can be seen in Table 3. For illustration purposes we left Artworks without configurations (None).

There are no preconditions defined, except for the configurations related to elements accessible only to logged-in users (Admin[5]; Donations[6]; MoreInfo[7], Logout

**Table 4** Configurations defined for the elements within 2nd level of the model

| Id   | Name           | Test goals | Input values                | Checks      |
|------|----------------|------------|-----------------------------|-------------|
| 8.2  | Name           | G_IV       | input = 'John Doe'          | None        |
| 8.3  | Contact        | None       | None                        | None        |
| 8.4  | Address        | None       | None                        | None        |
| 8.5  | Birthdate      | None       | None                        | None        |
| 8.6  | Username       | G_IV       | input="myusername"          | None        |
| 8.7  | Password       | G_IV       | input="mypass"              | None        |
| 8.8  | PasswordRepeat | None       | None                        | None        |
| 8.9  | Email          | G_IV       | input="john@unkownuser.com" | None        |
| 8.10 | RegisterButton | G_Call     | None                        | Change page |

[10]). The precondition defined for those elements is:

*Login.username == 'admin' and Login.password == 'admin'*

The second level of the model describing the Registry[8] Form is illustrated in Fig. 8. It has several Input UITPs for testing valid and invalid input data. These can be executed in any order, so they are within a Group element (Registry-Group[8.1]). The RegisterButton[8.10] corresponds to the final submit button of said web page.

The configurations defined for each UITP within model in Fig. 8 are in Table 4.

None of the UITP within the second level of the model have preconditions. Leave blank preconditions is the same as writing True, which means there is no restriction and, as such, the corresponding configurations will always be executed.

### 5.3 Test cases

The test case generation from the model in Figs. 7 and 8 generates many test paths since RegisterForm[8.1] is a group with eight elements that can be executed in any order. Just for the second level of the model, there are  $8! = 40320$  different test paths. Considering that the first level has 20 test paths because there are two optional elements, the overall set of test paths would be 483848 [21]. To restrict such set, the test case generation algorithm allows to define an upper bound to the number of generated permutations when the models have Groups. Some examples of generated test paths are the following:

Path1:[Init,3,8.2,8.3,8.4,8.5,8.6,8.7,8.8,8.9,8.10,2,5,6,7,4,9,10,End]

Path2:[Init,4,9,3,2,5,6,7,10,End]

Path3:[Init,9,4,2,5,6,7,10,End]

where the elements depicted by the number 8.n, are those corresponding to the second hierarchical abstraction level detailing Form Registry[8]. Path2 ignores the optional element number 8 (Registry[8] Form) and Path3 ignores both optional elements (Registry[3] Call and Registry[8] Form). After calculating the test 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 (G\_LV and G\_LINV in Table 3) the third path (Path3) would be transformed into two test cases:

tc1:[Init; 9, 4, 2(G\_LV), 5, 6, 7, 10, End]

tc2:[Init; 9, 4, 2(G\_LINV), 5, 6, 7, 10, End]

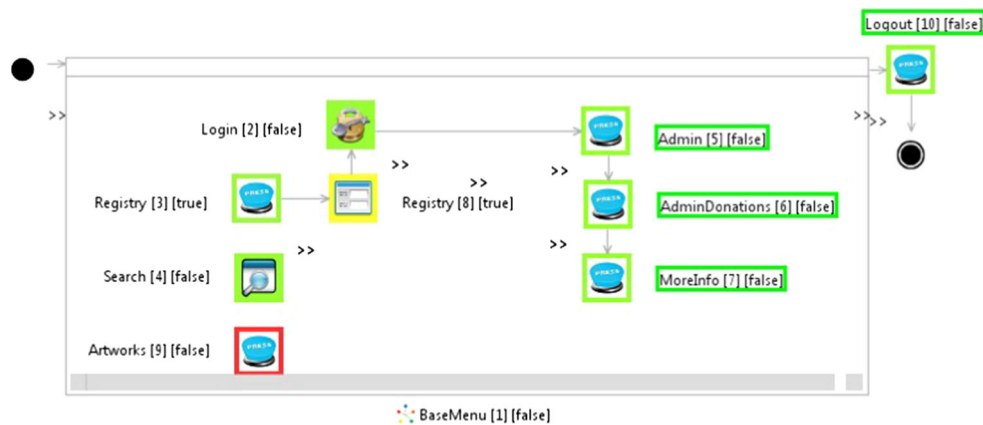
However, since elements 5, 6, 7 and 10 have a precondition demanding a valid login, the second test case will be instead [Init; 9, 4, 2(Login\_LINV), End]. This explains why preconditions are useful as a way to limit the final test cases to perform. The same expanding process is performed for the other UITP within such path (and for the other paths) in order to obtain the final test cases.

### 5.4 Coverage tool output

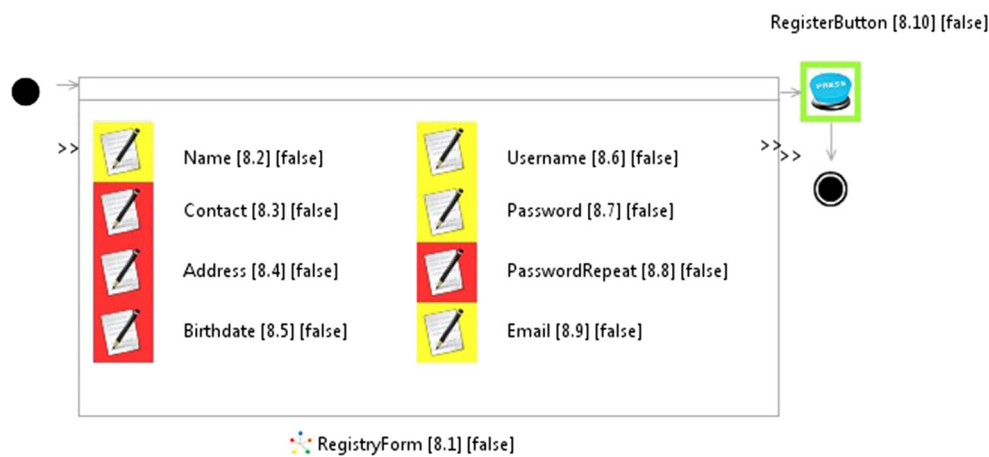
Figures 9 and 10 show the coverage information obtained by Test Goal Analysis and Constraint Analysis. Elements in red have no TGs configured; elements in yellow have some TGs defined; and elements in green have all TGs defined.

The green boxes around the labels of the elements indicates that there is at least one state in which the preconditions hold. Labels without a surrounding box mean that there is no precondition defined for corresponding element.

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 Art-



**Fig. 9** Result of the model coverage analysis of the first level of the model (Color figure online)



**Fig. 10** Result of model coverage analysis on the second level of the model (*Registry[8]Form*) (Color figure online)

works element, which has no configurations, and inside the Registry Form, which does not guarantee that the goals for its containing elements are fully exercised. Indeed, taking a second look at the configuration tables (Tables 3, 4), 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.

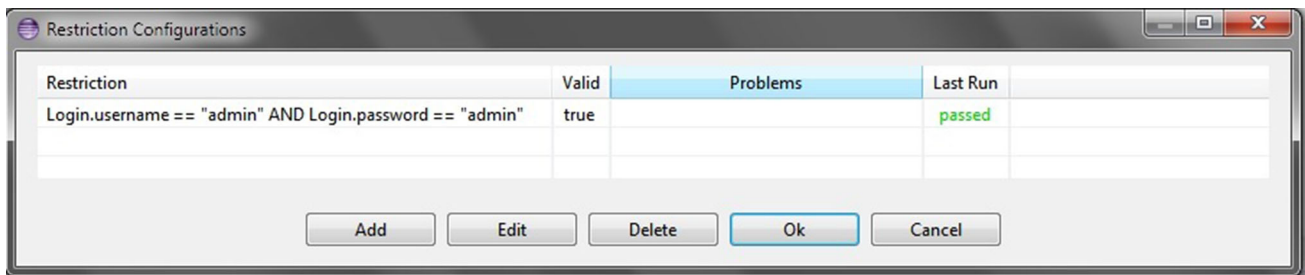
Regarding constraint analysis, it is possible to see the result in Fig. 11. It indicates that precondition defined can be True so the corresponding TGConf will be executed. Indeed there is a configuration defined for Login[2] in Table 3 that assigns “admin” to the username and “admin” to the password.

PARADIGM-COV also performs execution analysis. To illustrate the execution analysis, we forced a crash near the end of the test case execution process, as to showcase non-executed tests. Once the testing finishes, a tabbed view shows the results (Fig. 12). As shown, the report informs that

85.71% of all TGConfs passed and 92.86% of all TGConfs were executed. By analyzing the reports produced, the tester needs to understand why elements 10 and 8.10 did not reach 100% passed and executed. For element 10, it happens because we forced a crash. For element 8.10, all its configurations (TGConf) were actually executed, but none has passed (i.e., checks return False). So, either the AUT or the model is wrong. The tester must analyze the checks in the TGConfs defined for this element and decide if it is a bug within the AUT or a problem with the specification. This analysis is facilitated by the expanded Test Case Statistics, dubbed test trace in Fig. 12, which displays the input data associated with failed or non-executed TGConfs. In this particular case, there is a check which verifies if the web page changes when submitting data to the server. Yet, this check failure indicates that the SUT remains on the same page instead, indicating a possible bug in the SUT.

The result of code coverage analysis can be seen in Fig. 13. Element 3 is fully covered on the model, but does not reach





**Fig. 11** Constraint analysis

**Fig. 12** 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)

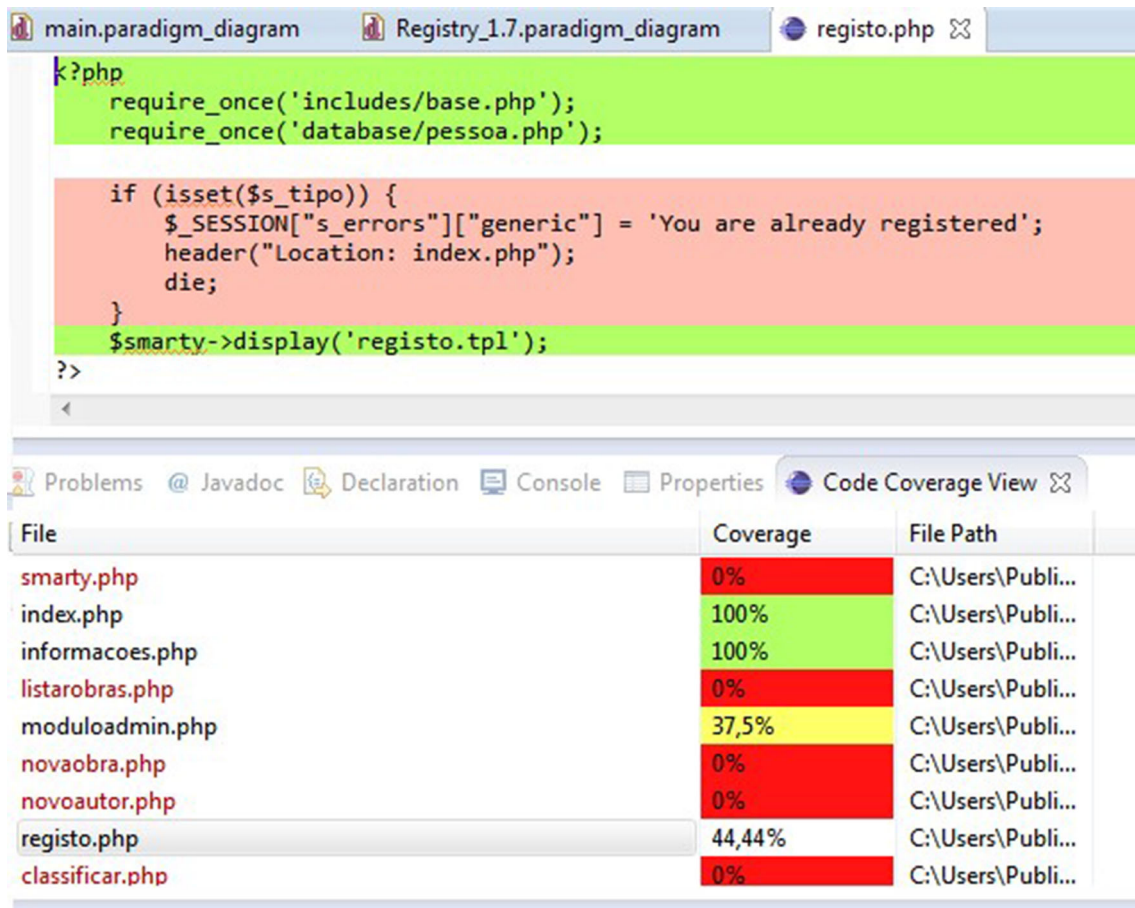
Test Case Coverage [<testcase>: <checkResult> | <executionResult>]:
* [0,1,9,4,2,11]: 100% passed | 100% executed
* [0,1,9,4,2,5,6,7,10,11]: 100% passed | 100% executed
* [0,1,9,4,3,8,8.0,8.1,8.6,8.7,8.8,8.9,8.2,8.3,8.4,8.5,...
* [0,1,9,4,3,8.0,8.1,8.6,8.7,8.8,8.9,8.2,8.3,8.4,8.5,...
  => uncovered at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

Test Trace [<executionOrder> <elementID>: <testResult>]:
(executed tests based on generated scripts)
1) 4      : passed
2) 2      : passed
3) 4      : passed
4) 2      : passed
...
15) 8.10 : failed <check="change page" flag="false"...
...

```

full code coverage (as shown by the code block signaled in red). The uncovered code block pertains to a situation occurring when logged-in users attempt to access this page. Such can happen if an authenticated user tries to access this functionality directly by typing the URL. The model built does not aim to test this functionality, so the test cases generated do not exercise it. However, by analyzing the code coverage results, the tester can build additional tests to execute and test this block of code if he wants.

To illustrate the test script analysis, we used the test script shown in Fig. 5 and assessed its quality over the model. The final report is shown in Fig. 14. As it can be seen in the report, just Elem 2 is considered fully executed. Despite the test scripts provides test input data different from the one shown in Table 3, the element 2 is considered fully executed (100% executed) because both TGs (G\_LV and G\_LINV) are executed. None of the other elements were executed (not executed).



**Fig. 13** Code coverage for a single file and overall results for the list of analyzed files (Color figure online)

#### Test Script Analysis Report

- \* Elem 2: 100% executed
- \* Elem 3: not executed
- \* Elem 4: not executed
- \* Elem 5: not executed
- \* Elem 6: not executed
- \* Elem 7: not executed
- \* Elem 9: not executed
- \* Elem 8.2: not executed
- \* Elem 8.3: not executed
- \* Elem 8.4: not executed
- \* Elem 8.5: not executed
- \* Elem 8.6: not executed
- \* Elem 8.7: not executed
- \* Elem 8.8: not executed
- \* Elem 8.9: not executed
- \* Elem 8.10: not executed

**Fig. 14** Test script analysis report

## 6 Conclusions

This paper presents a multidimensional coverage analysis approach and a tool (PARADIGM-COV) that is able to provide information according to:

- test requirements coverage (test goal analysis),
- model coverage (restriction analysis to check if preconditions ever hold),
- test case coverage (passed/executed metrics through Execution Analysis),
- code coverage (which parts of the code were exercised) and,
- test script analysis (assess the quality of a test script related to the coverage of a PARADIGM model).

The case study illustrated the main functionalities of the tool. The results of the experiment showed the usefulness of

having different coverage metrics over the model and over the code:

- during configuration of the model, the tester could check if all the test input data was provided;
- during test case execution it was possible to assess the percentage of test cases executed versus tests planned, and the test cases that passed versus the ones that failed;
- with code coverage analysis it was possible to detect situations where code coverage was not 100% despite complete model coverage.

Regarding the usability of the tool, along the experiments performed, the users appreciate having different ways to assess the quality of their configuration process. They recognized the benefit of having different coverage information over the model. The only comments less positive was linked to the configuration process within PBGT tool which is not directly related to the PARADIGM-COV tool. In particular, the users mentioned that the configuration process requires interaction with multiple windows which complicates the process. Suggestions for improvement were collected and will be analyzed in the future.

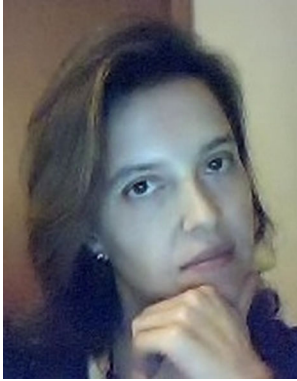
In the future, we also intend to implement code coverage analysis for other programming languages besides PHP and improve the usability of PARADIGM-COV tool. For an example in particular, and while we consider colors to be a visually intuitive marker, we are aware that not everyone reacts to color in 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.

Finally it is our belief that this coverage analysis approach can be applied in other contexts besides PBGT project and that may be useful to help the testers during the configuration of the testing process within MBT tools and promote the usage of MBT tools in industry.

## References

1. Weyuker, E.J.: Axiomatizing software test data adequacy. *IEEE Trans. Softw. Eng.* **12**, 1128–1138 (1986)
2. El-Far, I.K., Whittaker, J.A.: Model-based software testing. In: Marciniak, J.J. (ed.) *Encyclopedia of Software Engineering*. Wiley, New York (2002)
3. Eslamimehr, M.M.: The survey of model based testing and industrial tools. Master's Thesis, Linköping University (2008)
4. Vilela, L., Paiva, A.C.R.: PARADIGM-COV—a multidimensional test coverage analysis tool. In: *CISTI 2014—9 Conferencia Ibrica de Sistemas y Tecnologas de Informacin*, Barcelona, 18–21 Junio 2014
5. Moreira, R.M.L.M., Paiva, A.C.R.: A GUI modeling DSL for Pattern-Based GUI Testing PARADIGM. In: Maciaszek, L.A., Filipe, J. (eds.) *Proceedings of the 8th International Conference on Evaluation of Novel Approaches to Software Engineering (ENASE)*. SciTePress, Lisbon (2014)
6. Batchu, R., Dandass, Y.S., Skjellum, A., Beddhu, M.: MPI/FT: a model-based approach to low-overhead fault tolerant message-passing middleware. *Clust. Comput.* **7**(4), 303–315 (2004)
7. Stankovski, V., Petcu, D.: Developing a model driven approach for engineering applications based on mOSAIC. *Clust. Comput.* **17**(1), 101–110 (2014)
8. Stobie, K.: Model based testing in practice at Microsoft. *Electron. Notes Theor. Comput. Sci.* **111**, 5–12 (2005)
9. Grieskamp, W., Kicillof, N., Stobie, K., Braberman, V.: Model-based quality assurance of protocol documentation: tools and methodology. *Softw. Test. Verif. Reliab.* **21**, 55–71 (2011)
10. Pretschner, A.: Model-based testing. In: *Proceedings of 27th International Conference on Software Engineering, 2005 (ICSE 2005)*, pp. 722–723 (2005)
11. Weißleder, S.: Simulated satisfaction of coverage criteria on UML state machines. In: *2010 Third International Conference on Software Testing, Verification and Validation (ICST)*, pp. 117–126 (2010)
12. Weißleder, S.: Coverage Simulator. <http://covsim.sourceforge.net/>. Accessed Oct 2016
13. Weißleder, S., Rogenhofer, T.: Simulated restriction of coverage criteria on UML state machines. In: *2011 IEEE Fourth International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, pp. 34–38, March 2011
14. Weißleder, S.: Test models and coverage criteria for automatic model-based test generation with UML state machines. PhD Thesis, Humboldt-University Berlin (2010)
15. Veanes, M., Campbell, C., Grieskamp, W., Schulte, W., Tillmann, N., Nachmanson, L.: Model-based testing of object-oriented reactive systems with spec explorer. In: Hierons, R.M., Bowen, J.P., Harman, M. (eds.) *Formal Methods and Testing*, p. 3976. Springer, Berlin (2008)
16. Shafique, M., Labiche, Y.: A systematic review of model based testing tool support. Technical Report SCE-10-04. Department of Systems and Computer Engineering, Carleton University, Ottawa (2010)
17. Huima, A.: Implementing conformiq Qtronic. In: Petrenko, A., Veanes, M., Tretmans, J., Grieskamp, W. (eds.) *Testing of Software and Communicating Systems*, p. 112. Springer, Berlin (2007)
18. Andrade, F.R., Faria, J.P., Paiva, A.C.R.: Test generation from bounded algebraic specifications using alloy. In: *ICSOF* (2), pp. 192–200 (2011)
19. Andrade, F.R., Faria, J.P., Lopes, A., Paiva, A.C.R.: Specification-driven unit test generation for Java Generic Classes. In: Derrick, J., Gnesi, S., Latella, D., Treharne, H. (eds.) *Integrated Formal Methods*, pp. 296–311. Springer, Berlin (2012)
20. IBM: Safety-related software development using a model-based testing workflow. <http://www.ibm.com/developerworks/rational/library/safety-related-software-development/index.html>. Accessed Oct 2016
21. Moreira, R., Paiva, A.C.R., Memon, A.: A pattern-based approach for GUI modelling and testing. In: *The 24th IEEE International Symposium on Software Reliability Engineering—ISSRE* (2013)
22. Sacramento, C., Paiva, A.C.R.: Web application model generation through reverse engineering and UI pattern inferring. In: *Proceedings of the 9th International Conference on the Quality of Information and Communications Technology—QUATIC*, Guimaraes, Portugal, 23–26 Sept 2014
23. Nabuco, M., Paiva, A.C.R.: Model-based test case generation for web applications. In: *14th International Conference Computational Science and Its Applications—ICCSA 2014*, pp. 248–262 (2014)

24. Paiva, A.C.R., Faria, J., Tillmann, N., Vidal, R.: A model-to-implementation mapping tool for automated model-based GUI testing. In: 7th International Conference on Formal Engineering Methods—ICFEM, vol. 3785, pp. 450–464, UK, 1–4 Nov 2005



**Ana C. R. Paiva** is Assistant Professor at the Informatics Engineering Department of the Faculty of Engineering of University of Porto (FEUP) where she works since 1999. She is a Researcher at INESC TEC in the Software Engineering area. She has a PhD in Electrical and Computer Engineering from FEUP with a thesis titled “Automated Specification Based Testing of Graphical User Interfaces”. Her expertise is on the implementation and automation of the model

based testing process. She is a member of the PSTQB (Portuguese Software Testing Qualification Board) board general assembly, member of TBok, Glossary, and Examination Working Groups of the ISTQB (International Software Testing Qualification Board).



**Liliana Vilela** is a graduate from FEUP’s Informatics Engineering masters course, where she specialized on the topic of testing coverage. Since 2015 she’s been working in the role of DevOps–Software Security Specialist at I2S, where she combines the two things she likes best: software quality and automation. In that role, she obtained the GIAC’s Security Essentials Certification, and is currently working in her masters in Information Security at FCUP since 2016.